ED 033 868 SE 007 681

TITLE	Introduction to an Algorithmic Language (Basic).
INSTITUTION	National Council of Teachers of Mathematics, Inc., Washington, D.C.
Pub Date	68
Note	53p.
Available from	National Council of Teachers of Mathematics, 1201 Sixteenth Street, N.W., Washington, D.C. 20036
EDRS Price	EDRS Price MF-$0.25 HC Not Available from EDRS.
Descriptors	*Computer Assisted Instruction, *Computer Science, Development, Instruction, Instructional Materials, Mathematics, *Mathematics Education, *Problem Solving

Abstract

This booklet was written to help the mathematics teacher introduce computers through an easy, problem-oriented language. In Section I, a problem is selected and solved in a manner that builds up the use of the language. In Section II, the language is applied to three sample problems to illustrate further the programming techniques presented in Section I. These problems illustrate three different mathematical settings that lend themselves to computer analysis. The first is from number theory and forces the student to recall precise definitions of certain key mathematical concepts in order to write the necessary computer program. The second is a problem from advanced algebra. The last is from the area of statistics and represents a useful program for students (or teachers) for analyzing data. Related exercises for the reader are given at the end of each sample problem. (RP)

INTRODUCTION ALGORITHMIC LANGUAGE

NATIONAL COUNCIL OF
TEACHERS OF MATHEMATICS
1201 Sixteenth Street N.W.
Washington, D.C. 20036

Library of Congress
Catalog Card Number
68-26706

Table of Contents

Introduction

COMPUTERS have great promise as a tool in the teaching of mathematics. Many applications of the computer as a tool for learning through problem solving have been found, and many programs in computer education are in existence throughout the United States. It can be anticipated that the use of computers will become an integral part of the secondary school mathematics curriculum. Indeed, such is already the case in many school systems.

This booklet aims to help the mathematics teacher introduce computers through an easy, problem-oriented language. In Section I a problem is selected and solved in a manner that builds up the use of the language. The statements of the language are few in number and can be clearly understood and readily learned. Then in Section II the language is applied to three additional mathematics problems.

All the problems in the booklet can be solved on any computer that can accept the language called "BASIC." It is expected that more and more computer facilities will make BASIC available to users. For facilities that do not accept this language, transfer to other usable languages is not difficult. After students are introduced to concepts of programming and can produce logical solutions to problems, the language becomes secondary. The knowledge transfer to other languages is very high.

I *oducing* BASIC

IN INTRODUCING the programming language BASIC, a reasonable approach is to consider an elementary but interesting mathematical setting. One interesting problem for the application of computers is to develop a computer algorithm for finding a good approximation to $\sqrt{n}$ for $n \geq 0$—in computer typography, to $\sqrt{\text{N}}$ for N $> = 0$.

Introductory Problem

Normally, a satisfactory approach to a problem such as this is to consider first a special case for N, say 7. We will write a computer program to enable us to use the computer to find a good approximation to $\sqrt{7}$.

A note describing a computer program is appropriate here. A program is simply an ordered sequence of statements or steps. For example, if you were to write out the rules of the traditional steps for long division, this could be thought of as a program even though it is not a formal computer program.

In writing computer programs it is necessary to use a language or vocabulary that a particular computer "understands." There are many computer languages. In this booklet we have selected one language, BASIC, over the others because of the small number of commands needed, its easy application in solving problems, and its practicality in our evolving educational setting. At the present time there are many languages with much more extensive commercial use than BASIC. For example, FORTRAN and ALGOL are such languages.

Now back to our initial problem—how to use the computer to find

$\sqrt{7}$. Our first programs will involve very few programming statements and hence will be easy to write. Then, since these programs will be inefficient in their use of the machine, we will attempt to utilize more of the elementary BASIC language statements to write "better" programs. The improved programs will introduce you to additional statements of BASIC, and the sequence of program development should suggest a technique appropriate for teaching other mathematical topics with a computer. The ability to develop a general algorithm is, in fact, one of the broad goals of mathematics education and is considered by many to be the highest level of problem solving.

Before approaching the given problem a brief review note is necessary: A positive number x is the square root of N if $x^2 = N$. Using this notion, we can start with our first computer program. Initially we shall use the computer as a computational device, similar to a calculator. In the language BASIC *the computer can be instructed to do computations within a statement called a* PRINT *statement.* A sample program will help here. A preliminary estimate by mental arithmetic indicates that $\sqrt{7}$ is between 2 and 3 and is in fact greater than 2.5, since $(2.5)^2 = 6.25$. Thus consider 2.6 as an approximation to $\sqrt{7}$. One way to check the accuracy of this approximation is to square 2.6. The following BASIC program tells the computer to do this computation and print the result.

Program A

`1 PRINT 2.6 * 2.6`	This command, or statement, instructs the computer to compute and then print the result of multiplying 2.6 by 2.6. (Note that the symbol * is used for multiplication and stands for "times," so that 2.6 * 2.6 means "2.6 times 2.6.") The symbols for addition and subtraction are the usual ones, namely, + and −. A slash, /, is used for division. The notation for an exponent is a vertical arrow; for example, 2.6 ↑ 2 means "2.6 squared." Thus Statement 1 could be written, alternatively, as "1 PRINT 2.6 ↑ 2."
`2 END`	This statement tells the machine that this is all there is to this program. All programs in BASIC must end with an END statement.
`RUN`	After a program has been read into the computer the operator types RUN when he wishes the computer to execute that program.

One further note about our first program is in order. You will have noticed that each program statement, up to and including END, has a number in front of it. These numbers tell the computer the order in which the statements are to be executed—and this is always done from smallest to largest. The END statement must *always* have the largest program number. The numbers in our program did not have to be 1 and 2; they might better have been 10 and 20. Using these numbers would have enabled us to insert steps in between, if necessary, merely by calling the new steps 15, 17, etc. The steps can be typed in any order in the program, since the computer will always execute in numerical order; that is, the machine will automatically order the program statements by number. This is a characteristic of many computer languages.

Now let us consider the output or "printout" from the computer for our first program. This is given below. (Each program written in this section will be repeated when the computer output is given for your information. In addition, the programming statements introduced up to that point will be listed.)

Program A

```
1 PRINT 2.6 * 2.6
2 END
RUN
```

Output

```
6.76
```

BASIC *programming statements introduced thus far:* PRINT . . . and END

The approximation of 2.6 appears to be a reasonable guess, but is this as good an approximation as we can get? We can repeat the procedure for 2.7 or 2.8, if we wish. What about approximating to hundredths? Could we write the program to do some further calculations? One handy feature of the PRINT . . . statement is the fact that many different expressions can be evaluated within the same statement, as long as they are separated by commas. Thus our next program could look like the following.

Program B₁

```
 5 PRINT  2.60 ↑ 2,  2.61 ↑ 2,  2.62 ↑ 2,  2.63 ↑ 2,  2.64 ↑ 2,  2.65 ↑ 2
 6 PRINT  2.66 ↑ 2,  2.67 ↑ 2,  2.68 ↑ 2,  2.69 ↑ 2,  2.70 ↑ 2
10 END
```

Note that the two PRINT . . . statements are needed, since each line must be preceded by a command.

The program is repeated below, and the output is given.

Program B_1

```
 5 PRINT  2.60 ↑ 2,  2.61 ↑ 2,  2.62 ↑ 2,  2.63 ↑ 2,  2.64 ↑ 2,  2.65 ↑ 2
 6 PRINT  2.66 ↑ 2,  2.67 ↑ 2,  2.68 ↑ 2,  2.69 ↑ 2,  2.70 ↑ 2
10 END
RUN
```

Output

```
6.76      6.8121    6.8644    6.9169    6.9696    7.0225
7.0756    7.1289    7.1824    7.2361    7.29
```

BASIC *programming statements introduced thus far:* PRINT . . . (with commas) and END

Note that we have to refer back to our list of calculations in the given program to determine which x and x^2 go together. Also, if we were to repeat this type of calculation for a large number of approximations, it would get pretty tiresome to keep writing ↑ 2. Since we are squaring every time, it would be handy to have some way to read a value from a list and then have a single command that instructs the computer to square the number. In BASIC we have a pair of commands that enable us to read data from a list. These are appropriately called the READ . . . and DATA . . . statements. Read the following program and discussion carefully.

Program B_2

```
10 READ X
```

This instructs the computer to find a DATA list and set X equal to the first value on the list—in this case, 2.60 would be the first number in the DATA list. In BASIC any letter or any letter and a single digit is acceptable for the name of a variable *(hence we could have used* S, S1, Q, Q2, *or any other such symbol for a variable).*

```
20 DATA  2.60, 2.61, 2.62, 2.63, 2.64, 2.65, 2.66, 2.67, 2.68, 2.69, 2.70
```

The DATA statement can fall any place in the program. This is the list of possible replacements for X. *The computer goes to this list only when instructed to do so in a* READ . . . *statement.* Other wise, the computer pays no attention to the statement. For each successive READ . . . command the computer selects the next piece of data from the list until out of data, and then the computer

	automatically goes to END—this is a distinct characteristic of BASIC. (Note: *The data must be in decimal or integer form; fractions and radicals are not accepted.*)
30 PRINT X ↑ 2	This instructs the computer to print the value of x squared.
40 GO TO 10	At this point it is necessary either to repeat the earlier READ . . . and PRINT . . . instructions or, if possible, go back in the program and use the earlier statements over again. We use a GO TO . . . statement to jump from one statement to another in a program. In our program the GO TO 10 sends the computer back to Statement 10, the READ instruction. The computer repeats this process until out of data in 20 and then goes to END (the highest numbered statement).
50 END	

The above program is repeated below, and the output is given.

Program B_2

```
10 READ X
20 DATA 2.60, 2.61, 2.62, 2.63, 2.64, 2.65, 2.66, 2.67, 2.68, 2.69, 2.70
30 PRINT X ↑ 2
40 GO TO 10
50 END
RUN
```

Output

```
6.76
6.8121
6.8644
6.9169
6.9696
7.0225
7.0756
7.1289
7.1824
7.2361
7.29
OUT OF DATA IN 10
```

BASIC *programming statements introduced thus far:* PRINT . . . , END, READ . . . , DATA . . . , GO TO . . .

There may be some question as to whether program B_2 is really better than B_1. To illustrate the superiority of the type of program illustrated by B_2, we need only consider the case of evaluating a polynomial such as $3x^3 + x^2 - x$ for successive values of x, say 2.60, 2.61, . . . , 2.70. The PRINT . . . statement for this problem would be a long monster. Try it! This illustrates the necessity for using variables in our programs.

At this point let us stop and see what we can use the computer to do. We can actually write a program to instruct the computer to do *any* particular computation for a given set of data (as long as we can use an algebraic form to write the computation to be performed). This is a useful technique; however, there may be times when we will not want to print the results of all the computations. For example, in program B_2 the computations 2.66^2, 2.67^2, . . . , 2.70^2 are all unnecessary, since the $\sqrt{7}$ is bounded between 2.64 and 2.65. It would be useful to be able to have the computer do a calculation for a given value for x and then stop and let us select a second approximation on the basis of the first results. For example, if you wished to find $\sqrt{1{,}433}$ to the nearest tenth, you might try 35.0, and then on the basis of the computation of 35.0^2 you would try a value larger or smaller. It would be a lot of useless work to do the calculations of 30.0^2 up to 40.0^2 (or some other guessed upper and lower bounds), stepping by tenths. There is a BASIC statement that allows the programmer (you) to decide what value you want to substitute for the variable. This is the INPUT . . . command. The following program shows how the INPUT . . . command can be used in finding $\sqrt{7}$.

Program B_3

```
10 INPUT X
20 PRINT X ↑ 2
30 GO TO 10
40 END
```

After reading (and storing internally) the program, the computer stops and waits for the programmer or operator to put in a value for x. After it receives this value for x, it prints the result of squaring x. And then what do you think happens? Note the use of the GO TO . . . statement. Without the GO TO . . . command in this program the computer would stop and the program would be completed. Then to do a second approximation, the programmer would have to start at the beginning of the program again. In this case, however, the program instructs the computer to go back to Statement 10. The machine will return to this instruction and stop and wait for the operator to select another value for x. The following output shows a running of this program. (The values to be substituted were arbitrarily selected.)

Program B_3

```
10 INPUT X
20 PRINT X ↑ 2
30 GO TO 10
40 END
RUN
```

Output

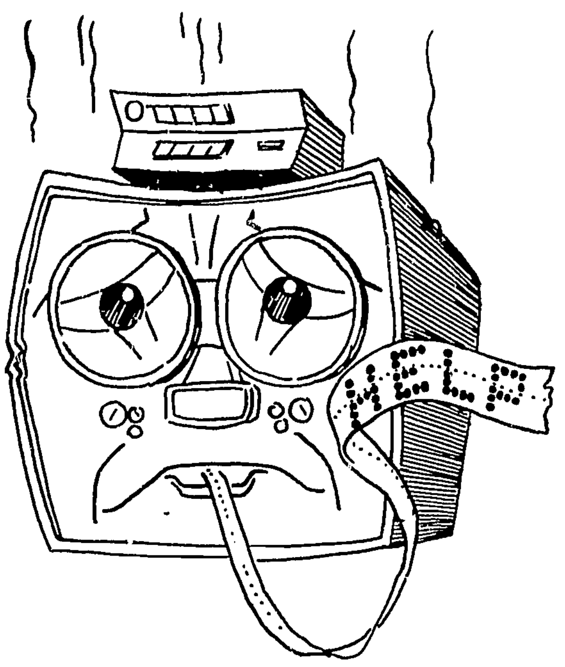
```
? 2.6
  6.76
? 2.7
  7.29
? 2.65
  7.0225
? 2.64
  6.9696
? 2.645
  6.996025
? STOP
```

The computer typed the question marks. We typed the values of x. The computer printed the values of x^2.

BASIC *programming statements introduced thus far:* PRINT . . . , END, READ . . . , DATA . . . , GO TO . . . , INPUT . . .

Notice that we needed to use an additional comment, STOP, in the actual running of our program. This particular program might run "forever" unless we tell the computer to stop. We can type one of two possible words to tell the machine that we have done enough work on this problem. The word used in the sample running of the program was STOP. After STOP it is possible to start the computer on a new program. The reader should refer to a BASIC manual to familiarize himself with the actual operation of the teletype and the input capabilities and commands of the facility. The words RUN and STOP do not appear in the computer program but, rather, are used to tell the computer to start or stop. These are a part of the actual running of the equipment, but they are not considered a part of the program.

There is one additional advantage in our last program, B_3, over the previous ones. Do you see what it is? We also checked 2.645^2. We were not able to do this in any of our previous programs as they were written. This would be possible if we revised them to include this particular value. The INPUT . . . statement enables us to make decisions on the basis of the most recent information and adjust our approximations accordingly. We are not bound by preconceived ideas.



We now have a very good computer program for approximating the square root of any given number N except that the program, while it enables us to get a good approximation, is quite inefficient. The investigator must sit at the console and actually communicate with the computer. This presents problems if the computer facility is not immediately accessible. Also, with this last program, the time spent on-line with the computer is excessive. Actually, the computer could do hundreds of calculations just during the time you spend typing in numbers. It would be valuable if we could have the computer check how close a particular approximation is and adjust the next trial value accordingly. To do this we need to have a statement that is referred to as a "decision" command. In addition, we want the machine then to change the trial value for X by some predetermined increment. The following program makes use of these ideas and introduces two more elementary BASIC statements.

Program C

10 LET X = 2.0	The LET . . . command enables us to set some variable X at a predetermined value. In this case we arbitrarily selected 2.0.

```
20 IF X ↑ 2 > 7 THEN 50
```

The IF . . . THEN . . . statement is the "decision" statement. In BASIC the computer action at this point is just what is implied. The computer compares the value of x^2 with 7. If, as in our first case when $x = 2$, the value of x squared is *not* greater than 7 (that is, it is less than or equal to 7), the computer proceeds with the next step in the program, 30. If, on the other hand, x squared is greater than 7, then the computer is told to go to a particular statement number, in this case 50. Thus the computer is asked to check a given condition. If the condition is false, the computer continues with the program; and if the condition is true, then the computer goes to the statement number given in the command. A further note is necessary: The condition to be checked can involve any one of the symbols that are shown in the box on page 11, together with those introduced earlier. It is up to the programmer to decide which condition is of interest.

```
30 LET X = X + .1
40 GO TO 20
```

Statements 30 and 40 illustrate a very interesting technique. The computer in this situation is told to replace x by the value $x + .1$. Since x is 2.0 prior to Statement 30, x is changed to 2.1, and then Statement 40 sends the computer back to Statement 20 to repeat the comparison of x^2 with 7. Note that in Statement 30 we do not have a true application of the concept of equality. Rather, the equality symbol can be thought of as being a symbol for the notion "is replaced by." *This is a technique that is used in all of the computer languages.* The technique is often referred to as a "counter." Note that if the increment on x were 1, and x started at 1, this would in fact make x equal to 2, 3, 4, and so on, for each "loop" of the program. *(A section of a program that is repeated a number of times is referred to as a loop.)*

```
50 PRINT X - .1, (X - .1) ↑ 2, X, X ↑ 2
```

When x squared first becomes greater than 7 the condition $x^2 > 7$ in Statement 20 is true, and thus the computer goes to Statement 50 given in the

THEN . . . part of the condition. In Statement 50 we might have decided to print only x^2, but actually we do not know what the corresponding value of x is at this point (all the computations and comparisons have been done in the machine). Therefore it is desirable to print x also. But what is true about x and x^2? They are the first values that are too large. Thus it would be useful, for this program, to print also the last value of x which still satisfied the condition that $x^2 < 7$. Then we can decide which approximation to use for a specific purpose and whether to seek other approximations. Program procedures such as this are actually up to the ingenuity of the programmer. Care should be taken to be sure that the computer output is in a readily applicable and usable form. In this case notice that we have bounded the true value of $\sqrt{7}$ to the nearest tenth.

60 END

The above program is given again, below, and the actual computer output is shown. The actual running time of the computer for this program is such that the final answer is typed almost immediately after the program has been put into the machine. The calculations are all done in less than half a second. This program is considerably more efficient than the previous program, B_3.

Program C

```
10 LET X = 2.0
20 IF X ↑ 2 > 7 THEN 50
30 LET X = X + .1
40 GO TO 20
50 PRINT X − .1, (X − .1) ↑ 2, X ↑ 2
60 END
RUN
```

Output

```
2.6    6.76    2.7    7.29
```

BASIC *programming statements introduced thus far:* PRINT . . . , END, READ . . . , DATA . . . , GO TO . . . , INPUT . . . , LET . . . , IF . . . THEN . . .

Meaning of Computer Symbols

+	plus
−	minus
*	multiply by
/	divide by
↑	exponent
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal to (or, is replaced by)
<>	not equal to (this symbol is used rather than ≠ or =/)

We might wish to stop now—or maybe you have an idea for a further refinement of the program for this problem. What if we wanted an approximation to the nearest .0001? Would you start at, say, 2.6000 and increment by .0001? How many calculations would be involved? (In this case there would be 458.) What if we weren't sure about 2.6 and started at 2.0 as in our previous program?—Then 6,458 calculations would be necessary. Thus the problem facing us is as follows: Can we reduce this number of computations and comparisons in some reasonable way and still have the computer make all the computations and decisions necessary? How about changing the increment?

Study the following program carefully. This program illustrates one possible procedure for getting an extremely accurate approximation with a small number of calculations.

Program D_1

```
10 LET X = 2.0
20 LET K = .1
30 IF X ↑ 2 > 7 THEN 55
40 LET X = X + K
50 GO TO 30
55 IF K < .001 THEN 90
60 LET X = X − K
70 LET K = K/10
80 GO TO 30
90 PRINT X − K, (X − K) ↑ 2, X, X ↑ 2
100 END
RUN
```

Output

```
2.6457    6.99973    2.6458    7.00026
```

Do you see how this program gets a closer and closer approximation to $\sqrt{7}$? Note that we have let K be our increment; as soon as $(X + K)^2$ becomes greater than 7, we drop back to the value of X just prior to the last incrementing and increase this value now by K/10. Thus we have *at most* ten calculations for each of the increments of .1, .01, .001, .0001. When K becomes .0001, what happens? We have now bounded the approximation to the nearest ten-thousandth; and, rather than go further, our decision statement (55) instructs the computer to go to Statement 90, the PRINT statement. Read through the program carefully until you understand all parts, for this program represents as good a refinement of our original problem as we will attempt in this booklet. No new statements are introduced in Program D_1, but it is somewhat longer and more complicated than any previously attempted.

At this point in our discussion we have noted all but two of the elementary BASIC statements. Addition of these two new statements, the FOR . . . and NEXT . . . statements, allows us to replace many of the statements in the previous programs. The new statements set up the "counter" referred to in our discussion of loops. The FOR . . . statement enables us to set a variable equal to some value and increment it for each successive loop in the computer program. Consider the program statement given below:

```
5 FOR X = 0 TO 48 STEP 2
```

This statement, in which STEP means to add an amount other than 1 (in this case, 2) should be read as though there were a comma separating the "48" and the "STEP." It says to start with X = 0, proceed with the rest of the program, and increment by 2 each time the computer returns to this statement until $X > 48$. At some point in the program there needs to be a statement like the following:

```
25 NEXT X
```

This tells the computer to return to the FOR . . . statement, increment (that is, increase X by 2 as specified), and continue with the program again. Thus in Statement 5 the next X value will be 2, then 4, and so on. The machine will automatically check each statement and go to the statement that follows the NEXT X statement as soon as $X > 48$. We can change the increment to whatever we wish. For example, Statement 5 could be as follows:

```
5 FOR X = 0 TO 48 STEP .5
```

or

```
5 FOR X = 0 TO 48 STEP 7.25
```

If we wish to increment by 1, then the STEP part of the statement can be left off, and the statement becomes

```
5 FOR X = 0 TO 48
```

Remember that each FOR . . . statement in a program must have an accompanying NEXT. . . . It is also possible to write the FOR . . . statement with variables—say A, B, and C—and then let A, B, and C take on selected values. In this case the FOR . . . statement would look like the following:

```
5 FOR X = A TO B STEP C
```

The two new FOR . . . and NEXT . . . statements can be used to replace portions of Programs C and D_1. An example follows, after we repeat Program C for the sake of ready comparison.

Program C

```
10 LET X = 2.0
20 IF X ↑ 2 > 7 THEN 50
30 LET X = X + .1
40 GO TO 20
50 PRINT X − .1, (X − .1) ↑ 2, X, X ↑ 2
60 END
RUN
```

Output

```
2.6     6.76     2.7     7.29
```

Program D_2

(*Program* C *with* FOR . . . , NEXT . . .)

```
10 FOR X = 2.0 TO 3.0 STEP .1
20 IF X ↑ 2 > 7 THEN 40
30 NEXT X
40 PRINT X − .1, (X − .1) ↑ 2, X, X ↑ 2
50 END
RUN
```

Output

```
2.6     6.76     2.7     7.29
```

Note that in Program D_2 we did not want to process the entire range of values set up in the FOR . . . statement. Instead, we gave the computer a rule for determining whether or not it was useful to continue processing values by means of the FOR . . . statement. We did this by inserting an

IF . . . THEN . . . statement between the FOR . . . statement and its associated NEXT . . . . *In this way, we "jumped out of" the* FOR . . . *loop before it would ordinarily end,* thereby saving computer time. It is important to note that although we could "jump out of" the FOR . . . loop, we must start at the beginning in order to reenter the loop.

BASIC *programming statements introduced thus far:* PRINT . . . , END, READ . . . , DATA . . . , GO TO . . . , INPUT . . . , LET . . . , IF . . . THEN . . . , FOR . . . NEXT . . .

In this new computer program notice how we use the FOR . . . and NEXT . . . statements to replace three statements in the original program. Program D_1, reprinted below from page 11, offers even more of a challenge—you might wish to try to write it, using FOR . . . and NEXT . . . statements, before going on to the next section, so that you can compare your results with Program D_3, a sample program using FOR . . . and NEXT. . . .

Program D_1

```
10 LET X = 2.0
20 LET K = .1
30 IF X ↑ 2 > 7 THEN 55
40 LET X = X + K
50 GO TO 30
55 IF K < .001 THEN 90
60 LET X = X − K
70 LET K = K/10
80 GO TO 30
90 PRINT X − K, (X − K) ↑ 2, X, X ↑ 2
100 END
RUN
```

Output

```
2.6457    6.99973    2.6458    7.00026
```

Program D_3

(*Program* D_1 *with* FOR . . . , NEXT . . .)

```
10 LET X = 2.0
20 FOR J = 1 TO 4
30 LET K = 10 ↑ (− J)
40 FOR X = X TO 3 STEP K
50 IF X ↑ 2 > 7 THEN 70
```

```
60 NEXT X
70 LET X = X − K
80 NEXT J
90 PRINT X, X ↑ 2, X + K, (X + K) ↑ 2
100 END
RUN
```

Output

```
2.6457    6.99973    2.6458    7.00026
```

Note that in Programs D_1 and D_3 we actually are telling the computer to do the same computations; however, use of the FOR . . . and NEXT . . . statements often shortens a program. Look at the program that uses the FOR . . . and NEXT. . . . Do you see what is happening? First X is set to a given value, 2, and then K is set at .1 (Statements 10, 20, and 30 in Program D_3 actually do this for K). Then X is incremented by .1 until X^2 is greater than 7. At this point the program goes to Statement 70, which sets X equal to the value just preceding the point at which the squared quantity exceeded 7. Then the increment is changed from .1 to .01 (this is actually 10^{-J}, which is now 10^{-2}). Then the squaring and comparing process is again done until X^2 is greater than 7 and we jump from the loop on X to the "outside" loop on K. *These two loops are referred to as "nested loops," which actually means a loop within a loop.*

With the addition of these two new statements it is possible to program many of the problems or algorithms in the secondary mathematics curriculum.

A list of the elementary BASIC statements and the program in which each was introduced is given in the table below. Some of the statements are listed together, since they must *both* appear in the program.

Statement(s)	*Sample Program*
PRINT . . .	A, B_1
END	A
READ . . . and DATA . . .	B_2
GO TO . . .	B_2
INPUT . . .	B_3
LET . . .	C
IF . . . THEN . . .	C
FOR . . . and NEXT . . .	D_2, D_3

The letters A, B, C, and D are used to name the sample programs to indicate levels of refinement, the D's being the most sophisticated or best programs.

Most computers have a stored program for finding square root. In BASIC this square root function, generally referred to as a subroutine, can be called in with the letters SQR followed by parentheses enclosing the number to be operated on. Thus we can eliminate all the previous programming and merely write the following program. The output is also given, for your information.

Program D_4

```
10 PRINT SQR(7)
20 END
RUN
```

Output

```
2.64575
```

Exercises

(Sample programs for all exercises in Sections I and II are given in the concluding section of this booklet.)

1. Write a program to extend C, D_1, or D_2 so that the computer will read any value for N from a data list and then find the square root by successive approximations.
2. Check your results in 1 above by using a READ . . . DATA . . . combination and the square root function.
3. Often we use an iterative process to find the square root of a number. One iterative process is to take your first approximation for the square root of N, call this X, and divide N by this approximation. Call the quotient Q. Then average Q and X, take their average as the second approximation, and repeat the process. Write a computer program to do this to find $\sqrt{7}$ to four decimal places.

II Sample Problems and Exercises

THE following section contains three sample problems to illustrate further the programming techniques presented in Section I. These problems illustrate three different mathematical settings that lend themselves to computer analysis. The first is from number theory and forces the student to recall precise definitions of certain key mathematical concepts in order to write the necessary computer program. The second is a problem from advanced algebra. The last is from the area of statistics and represents a useful program for students (or teachers) for analyzing data. The three problems are independent of one another and can be read in any order. At the end of each sample problem there are a few related exercises for you to try. Answers to all will be found in the back of this booklet.

Problem 1—Divisors of a Positive Integer

Now that we know something about the BASIC language, we will use it to help us solve the following problem:

Let M be a positive integer. Compute the set of positive integral *divisors* of M.

Let's look at a specific value of M—say 36. The divisors of 36 are 1, 2, 3, 4, 6, 9, 12, 18, and 36. We can obtain these divisors in a very short time by using paper and pencil. But how would we find the divisors of 3,942? One way is to divide 3,942 by 1, 2, 3, 4, 5, and so on, each time examining the remainder. If the remainder is zero, then we have found a divisor.

We will develop a BASIC program directing the computer to do most of the work for us. To do this, we must first select variables to represent the various results of our divisions: that is, we wish to represent the integral part of a quotient and also the remainder. For any integral trial divisor $N > 0$ let us consider Q to be the integral part of M/N and R to be the remainder, where

$$\frac{M}{N} = Q + \frac{R}{N}.$$

Therefore, for any positive integer M, consider N to be any consecutive integer from 1 to M—that is, 1, 2, 3, . . . , M. Since division can be checked by multiplication, the dividend can be expressed further in terms of N, Q, and R—specifically, $M = N \cdot Q + R$. Thus $R = M - N \cdot Q$. To clarify this notation further, consider the following example: If $M = 48$ and $N = 5$, then $Q =$ integral part of $48/5 = 9$; $R = 48 - 45 = 3$.

Does $M = N \cdot Q + R$?

Check: $48 = 5 \cdot 9 + 3$.

In this case $R \neq 0$; therefore 5 is not a divisor of 48. However, if $N = 6$, then $Q =$ integral part of $48/6 = 8$, and $R = 48 - 6 \cdot 8 = 0$. In this case $R = 0$; therefore 6 is a divisor of 48.

One approach to obtaining the set of divisors of M is to write a BASIC program directing the computer to divide an input value of M by N for $N = 1, 2, 3, \ldots, M$. After each division we will have the computer print the value of N and the value of the remainder R. Hence we can look at the output and determine which values of N are divisors of M.

In order to obtain the required integral part of the quotient on division of M by N, we will make use of a special function that is available to us. This is a stored program, as was the square root function, SQR( ), described in the previous section. This is the integer function, which is part of the BASIC language. This function can be elicited with the notation INT( ) where the expression within the parentheses is to be operated on. If e is any algebraic expression, then INT(e) = the greatest integer $\leq$ e. For example,

INT(5) = 5.	INT(5.8) = 5.	INT(3.6 + 2.7) = 6.
INT(−6) = −6.	INT(−6.3) = −7.	INT(48/5) = 9.
INT(0) = 0.	INT(57.392) = 57.	INT(48/6) = 8.

Hence INT(M/N) is the integral part of the quotient on dividing M by N.

Now, since we are able to represent Q and R, we are ready to write a BASIC program to find the division of a positive integer M. For our first program, let us consider M to be 10. The following program is one possible approach to this problem.

Program E_1

```
10 INPUT M
20 FOR N = 1 TO M
30 LET Q = INT(M/N)
40 LET R = M - N * Q
50 PRINT N, R
60 NEXT N
65 GO TO 10
99 END
RUN
```

We must first set our value for M. Then, using a FOR . . . NEXT . . . loop, we can generate our set of trial divisors and compute the integer quotient Q and the remainder R (Statements 30 and 40). The values of N and R are printed, and the computer is instructed to continue this process for all trial divisors.

Output

```
? 10
 1    0
 2    0
 3    1
 4    2
 5    0
 6    4
 7    3
 8    2
 9    1
10    0
? STOP
```

Looking at the output, can you see which are divisors of 10? The divisors of 10 are 1, 2, 5, and 10, since in each of these cases R = 0.

Some of you may have been surprised by the computer's ability to understand what we meant by the expression M − N * Q in Line 40 of the program. After all, it is possible to interpret it as either "multiply Q times the difference of M and N" or "subtract the product of N and Q from M." How can we be sure that the computer will correctly interpret our intentions? In cases such as this the computer has a set of rules, which it will always follow. When parentheses are not used to indicate otherwise (and this, also, is part of the set of rules), it will perform all exponentiations first; next it will perform all multiplications and divisions, and finally it will do all additions and subtractions. If the total number of multiplications and divisions in a given expression is larger than one, then the computer will perform the leftmost multiplication or division (regardless of which it is) first, skip over any additions or subtractions, and perform the next multiplication or division from the left. When all multiplications and divisions are complete, this process is repeated for additions and subtractions.

Hence in Line 40 the computer first looked for exponentiations. Finding none, it looked for a multiplication or division. It found N * Q and executed the multiplication. It found no further multiplications or divisions and hence looked for additions or subtractions. It found M − and the product it had already computed, and it performed that subtraction. Hence it found R to be the difference of M and the product of N times Q, as we desired.

This same program can be rerun for the case M = 3942. The only change necessary is in Statement 10, which will use 3942 for M. A partial output for this program is given below.

Output

```
? 3942
 1     0
 2     0
 3     0
 4     2
 5     2
 6     0
 7     1
 8     6
 9     0
10     2
11     4
12     6
13     3
14     8
15    12
16     6
17    15
18     0
19     9
20     2
```

Now we might wish to develop a more elegant procedure to generate the set of divisors of a positive integer. Our first procedure does not begin to tap the power of the computer! As in our square-root example, it would be nice to have the computer make some of our decisions. Since the quantity of output can become quite excessive, as in the case for M = 3942, it would be more efficient to include only the values of N that are divisors of M in the output. For each such value of N, it would also be convenient to include the quotient Q in the PRINT . . . statement. Can you write the refined program? One possible attempt is given in Program E_2 below (36 is a completely arbitrary choice for M).

Program E_2

```
10 INPUT M
20 FOR N = 1 TO M
30 LET Q = INT(M/N)
40 LET R = M - N * Q
50 IF R <> 0 THEN 70
60 PRINT N, Q
70 NEXT N
80 GO TO 10
99 END
RUN
```

If R $\neq$ 0, then the computer is instructed to go to Statement 70, and hence the PRINT . . . statement is skipped.

Output

```
? 36
 1    36
 2    18
 3    12
 4     9
 6     6
 9     4
12     3
18     2
36     1
? STOP
```

Something rather interesting occurred in the above output. Every divisor of 36 appeared twice! In fact, all the divisors were printed in the first six lines of output (1, 2, 3, 4, and 6 in the first column and 36, 18, 12, 9, and 6 in the second). Why did this happen? Can we take advantage of this to write a more efficient program? We can.

First, we note (it can be shown) that if we print the divisor N *and* the quotient Q for each trial value of N, then we need use only the following set of trial divisors:

$$\{1, 2, 3, \ldots, \text{INT}(\sqrt{M})\}.$$

That is, we can stop when we have reached the value of N equal to the integral part of the square root of M. (It is interesting to note that students in the classroom quickly come to this generalization or one very similar to it.) For large values of M, this reduction in output represents a tremendous savings in computer time (and cost!). We have incorporated this in the procedure below. (Note the use of the READ . . . DATA . . . statements rather than the INPUT . . . ; this allows us to consider more than one replacement for M. In this case we have M equal to 36, 42, and 851, respectively.)

***Program** E_3*

```
10 READ M
15 PRINT M
20 FOR N = 1 TO INT(SQR(M))
30 LET Q = INT(M/N)
35 LET R = M - N * Q
40 IF R <> 0 THEN 60
50 PRINT N, Q
60 NEXT N
70 GO TO 10
80 DATA 36, 42, 851
99 END
RUN
```

Output

```
? 36
 1   36
 2   18
 3   12
 4    9
 6    6
```

The divisors of 36 are 1, 2, 3, 4, 6, 9, 12, 18, and 36.

```
? 42
 1   42
 2   21
 3   14
 6    7
```

The divisors of 42 are 1, 2, 3, 6, 7, 14, 21, and 42.

```
? 851
 1   851
23    37
? STOP
```

The divisors of 851 are 1, 23, 37, and 851.

There are still many ways to make the procedure more efficient. You will discover some of these by working the exercises at the end of this section.

Let us now extend the problem we have worked on thus far to consider a concept frequently encountered in the upper elementary or junior high school grades, that of finding the greatest common divisor (G.C.D.) of two positive integers.

Let A and B be positive integers. How can we utilize the computer to obtain the G.C.D. of A and B? One way would be to print out the sets of divisors and look at them to determine the G.C.D. Before continuing let us establish our notation as follows:

Let D_A be the set of positive integral divisors of A and D_B be the set of positive integral divisors of B. Then let $C = D_A \cap D_B$. The greatest common divisor of A and B is the largest element of C. For example, let $A = 36$ and $B = 42$. Then

$$\begin{aligned} D_A &= \{1, 2, 3, 4, 6, 9, 12, 18, 36\}, \\ D_B &= \{1, 2, 3, 6, 7, 14, 21, 42\}, \\ C &= \{1, 2, 3, 6\}, \\ \text{and G.C.D.} &= 6. \end{aligned}$$

Now, since we already have a program to generate the set of divisors of a positive integer, let us use it to get the G.C.D. of any positive integer A and B. Here's how:

1. Use the program to generate the set of divisors of A (Set D_A).
2. Use the program to generate the set of divisors of B (Set D_B).
3. By inspection, write down the set $C = D_A \cap D_B$.
4. Select the largest element of C. This is the G.C.D. of A and B.

Below is the program for $A = 910$ and $B = 798$.

Program F_1

```
10 INPUT M
20 FOR N = 1 TO INT(SQR(M))
30 LET Q = INT(M/N)
35 LET R = M - N * Q
40 IF R <> 0 THEN 60
50 PRINT N, Q
60 NEXT N
70 GO TO 10
99 END
RUN
```

Output

```
? 910
 1    910
 2    455
 5    182
 7    130
10     91
13     70
14     65
26     35
```

The value of A was given to the computer, and the list of divisors of A was computed and printed.

```
? 798
 1    798
 2    399
 3    266
 6    133
 7    114
14     57
19     42
21     38
? STOP
```

The value of B was given to the computer, and the list of divisors was computed and printed.

After getting the above output, we need only select the common divisors from the sets. In our example the set of common divisors is

$$\{1, 2, 7, 14\},$$

and the G.C.D. is 14.

Again we might ask ourselves: Is there a better way to do this problem? Can we get the computer to search and make the decisions so only the results of concern are printed? This problem is given for you as an exercise.

Exercises

1. The program above causes INT(SQR(M)) to be evaluated for *each* time through the loop. A square root computation is relatively time-consuming compared to many other operations in BASIC. Rewrite the program so that INT(SQR(M)) is computed only *once* at the beginning and assigned to the variable, S. Then use S in the FOR . . . statement.
2. We have used these trial divisors:

   N = 1, 2, 3, . . . , INT(SQR(M)).

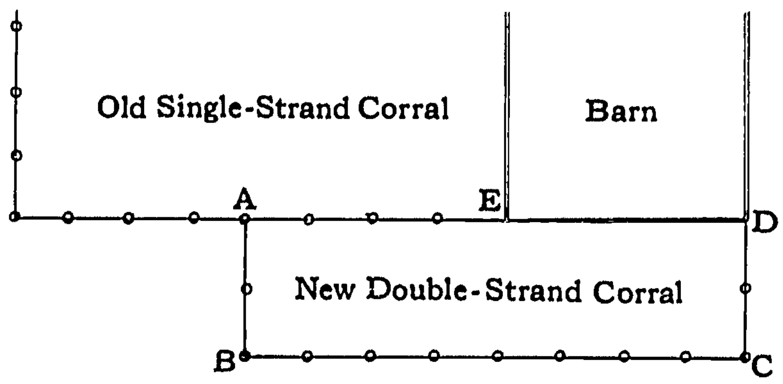
   That is, our program terminates when N > INT(SQR(M)). An equivalent condition is to terminate the procedure when N > Q. Write a program in which this fact is used to terminate the program.

3. Write a program to determine if an input value of M is a *prime number*. We assume that the input value for M is an integer $\geqq 2$. (A number, M, is a prime number if and only if it has *exactly* two *distinct* divisors, 1 and M. Hence 1 is *not* a prime number.) If M is prime, print 1 and M. If M is not prime, print 1, M, and N, where N is a divisor of M *other than* 1 or M.
4. Write a program for computing the G.C.D. of two positive integers, A and B, that prints only those divisors of A and B. (Hint: Use READ A, B and direct the computer to print N only if N is a divisor of both integers.)

Problem 2—Maximizing an Area

The following is an example of a well-known type of problem involving a maximum value of a simple function.

A farmer has a barn 37.0 feet long with a very long single-strand barbed wire corral on one end of the barn. He also has 525.1 feet of barbed wire to build a new two-strand rectangular corral along the side of the barn and extending out to the side of the old corral, using the barn and one strand of wire from the old corral in one side of the new corral as shown in the figure. It is then necessary to add one strand of wire from A to E and to put two strands along each of the line segments, AB, BC, and DC.



What should the dimensions of the new corral be in order for the corral to have the greatest possible area?

If AB is X feet long and BC is Y feet long, then AE is Y − 37.0 feet long; and, adding the length of one wire for AE and two wires for each other side, we get the equation for the length of barbed wire used in the new corral:

$$2x + 2y + 2x + y - 37.0 = 525.1,$$

or

$$y = (562.1 - 4x)/3.$$

The area of the new rectangular corral is $A = XY$, or, substituting for Y from above;

$$A = XY,$$
$$A = X \cdot \frac{562.1 - 4x}{3},$$
$$A = \frac{562.1x - 4x^2}{3}.$$

(This is a quadratic function whose graph is a parabola opening down.) The reader should study this carefully to be sure he understands the derivation of this function.

The questions now become:

For what value of x (that is, length of AB) is the area of the corral a maximum? From the graph, what is the x-coordinate of the vertex of the parabola?

Let us consider a very simple program to solve this problem. (You may wish to try a first attempt on your own.) A first approach might be to input a value for x and then have the computer print x and A. The following program illustrates this approach. Values of 50, 70, and 90 were arbitrarily selected for the sample output.

Program G_1

```
10 INPUT X                The student makes an initial guess for x.
20 PRINT X, (562.1 * X - 4 * X ↑ 2)/3
                          The computer prints x, the field length, and the
                          calculated area.
30 GO TO 10               The computer is instructed to return to Step 10
                          above for a new guess by the student.
50 END                    The end of the program. However, this program
                          never gets to the end, since Step 30 always
                          branches to Step 10.
```

Output

```
? 50
  50     6035
? 70
  70     6582.33
```

```
? 90
  90    6063
? STOP
```

This, of course, is a slow process, since there is a necessary wait while the student chooses and types in data. It may also, unless the student is a very astute guesser, take a long time to produce a nearly correct answer.

As in the earlier examples, the question is how we can improve this program. Let us look more closely at the problem. If $x = 0$, then the area is 0. If $x = 1$, then the area is $(562.1 \cdot 1 - 4 \cdot 1^2)/3$—that is, 558.1/3. It appears that as we choose larger values for x, the area gets larger. But what about x very large, say 1000? Then A is negative. Thus the area reaches a maximum value for some x; then becomes smaller until $x = 562.1/4$ and the area is again 0. To verify this pattern, the reader should refer to the output from Program G_1.

For our second approach to the problem, we might have the computer write out x and the area for every integral value of x in the set

$$\{x: 1 \leq x \leq 141\},$$

since $141 > 562.1/4$.

Program G_2

```
10 FOR X = 1 TO 141   This statement produces a loop that will be re-
                      peated 141 times, each time increasing x by 1.
                      Thus x takes on the values 1, 2, 3, . . . , 141.
20 PRINT X, (562.1 * X - 4 * X ↑ 2)/3
                      The computer calculates and prints x and the
                      area when the length is x, for each x from x = 1
                      to x = 141. Each FOR . . . must have a NEXT. . . .
30 NEXT X             Each time this step is encountered the computer
                      returns to Step 10, where x is increased by 1,
                      until the process has been completed with
                      x = 141. On the 141st pass the program goes on
                      to the statement following the NEXT . . . com-
                      mand: in this case, Statement 40.
40 END
```

Part of the output

```
 1    186.033
 2    369.4
 3    550.1
 4    728.133
```

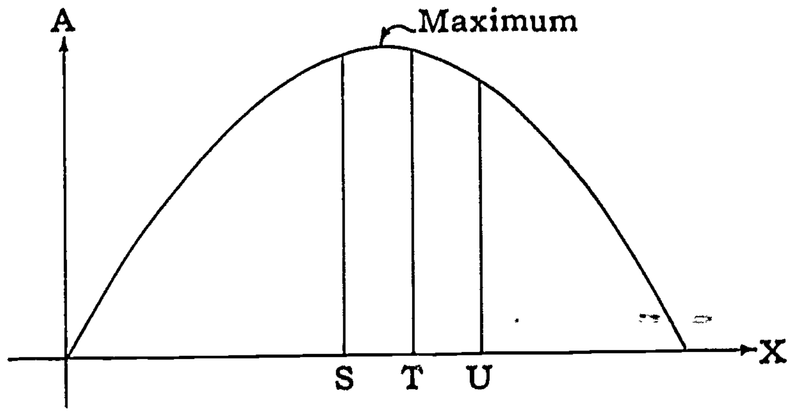
```
5     903.5
6    1076.2
7    1246.23
8    1413.6
9    1578.3
10   1740.33
11   1899.7
12   2056.4
13   2210.43
14   2361.8
15   2510.5
 .     .
 .     .
 .     .
65   6545.5
66   6558.2
67   6568.23
68   6575.6
69   6580.3
70   6582.33
71   6581.7
72   6578.4
 .     .
 .     .
 .     .
STOP
```

It is still not clear what value of x provides a maximum area, but the solution must be between 69 and 71, since in this interval the value for the area reaches a value greater than that for either 69 or 71 and begins a continuing decrease. We can now write a program that would search between 69 and 71, with smaller intervals, for a better approximation of the correct x value. This would entail only a change in the FOR . . . statement. The object of this example is to design a better program to search for x.

In the next paragraphs in this section we will set up a program that will determine for itself the interval for the finer search and then search it. Notice that this is quite similar to our task of narrowing down our estimate for $\sqrt{7}$. We can make this a continuing process so that each search of an interval yields a new, smaller, interval for a new search with even smaller subintervals.

Before considering our program, a word of caution about the problem is in order. It might have been assumed in the preceding discussion that, because when x = 71 the area is less than the area when x = 70, the maximum area would be found somewhere in the interval $70 \leq x \leq 71$; this is not necessarily true, although it may be.

We can assume that the area function looks something like the figure below, with S, T, and U consecutive integers. In this example, even though the maximum area is reached between the integers S and T, the first noted decrease is between the integers T and U. In order to be certain that the chosen subinterval will include the maximum value for the area, it will be necessary that the subinterval be $S \leq x \leq U$. That is, the subinterval must be two units long, these units being the last interval where the area increases and the first interval where the area decreases.



With this information in mind, our new program should do the following:

1. Search from x = 1 toward x = 141 looking for the first decrease in area.

2. Type out the end coordinates (x, area) of the interval on the curve that includes the maximum area, with the x interval two spaces long.

3. With units one tenth as long as the units of the previous search, the program will search the new interval (end points given above) for a decrease in area and again type out coordinates.

4. Return to Step 3 three times, each time finding and typing the end points of shorter and shorter subintervals containing the maximum value of the function.

5. Stop.

Referring to the graph given earlier, then:

1. The *interval of search* will always be $S \leq x \leq U$. The first value

of S will be 1 and of U, 141; but these will change as new subintervals are chosen.

2. R will always be the *length of the subinterval* into which we are breaking SU for the search. On the first search R = 1; for the second, R = .1; on the third, .01, etc.

The final program for this problem is given below, followed by a detailed discussion of each of the statements.

Program G₃

```
10 LET S =1
20 FOR I = 1 TO 3
30 LET R = 10/10 ↑ I
40 LET A = (562.1 * S − 4 * S ↑ 2)/3
50 LET B = (562.1 * (S + R) − 4 * (S + R) ↑ 2)/3
60 IF B > A THEN 75
70 GO TO 110
75 LET C = A
80 LET A = B
90 LET S = S + R
100 GO TO 50
110 PRINT S − R, C, S + R, B
120 LET S = S − R
130 NEXT I
140 END
```

Discussion of Program G₃

10 LET S = 1	We will set S = 1, since for the first search this interval SU is 1 to 141.
20 FOR I = 1 TO 3	This statement forms a loop that ends at Step 130. This loop will be traversed three times, the first time through I = 1, the second through I = 2, the third through I = 3. (This marks the beginning of the search of an interval.)
30 LET R = 10/10 ↑ I	When I = 1 (first time through the loop) this is R = $10/10^1$ or 1; when I = 2, R = $10/10^2$ = .1; when I = 3, R = $10/10^3$ = .01.
40 LET A = (562.1 * S − 4 * S ↑ 2)/3	The computer calculates the value for A at the left end of the first subinterval. Each time a new subinterval is defined for a search, the program

	will return to this step, where the first (right-end) value of the area will be computed. At all other times when a left-end value for the area is needed the old right-end value will be used and only a new right-end value computed, in order to save computer time.
50 LET B = (562.1 * (S + R) − 4 * (S + R) ↑ 2)/3	The computer calculates the area at the right end of the first interval, and the program returns here (every time the area does not decrease) to compute a new right end.
60 IF B > A THEN 75	B is compared to A, and the computer then executes Step 75 if B > A; otherwise Step 70 is executed. See Section I for other possibilities for the IF . . . THEN . . . statements using >, =, etc.
70 GO TO 110	When B < A we have discovered a subinterval two units long containing the maximum value, and the computer is instructed to go to the PRINT . . . statement.
75 LET C = A	This is done to save A as C for printing in case it is needed as the left end of a two-unit interval, should the next interval show a decreased area.
80 LET A = B 90 LET S = S + R	Statement 80 moves the interval under consideration over one unit to the right by moving the value at the right end to the left end, and S is increased for use in the calculation of a new right end in Step 50.
100 GO TO 50	The computer now returns to calculate a new right end. Notice that this does not affect the loop established in Step 70, since the NEXT . . . statement is not encountered.
110 PRINT S − R, C, S + R, B	The computer reaches the PRINT . . . when B < A. The PRINT . . . includes the end coordinates of an interval two units long containing the maximum area. Since the value of S is now the value of X at the left end of the one-unit-long subinterval where the first decrease was noted, the value of X at the left end of the two-unit-long interval will be one subinterval back, at S − R.

120 LET S = S − R	This statement resets the starting point of the new, shorter, interval for the next search.
130 NEXT I	The computer returns to the beginning of the interval search. Since this is the end of a FOR . . . loop, the value of I is incremented on the return to Step 20, and therefore in Step 30 the length of the subinterval is divided by 10 for the next search. After the third pass the I is exhausted; the computer will not return to Step 20 but will go on to Statement 140, the statement following the NEXT . . . statement.
140 END	The end of the program.

Output

```
69       6580.3
70.2     6582.42
70.25    6582.43
71       6581.7
70.4     6582.4
70.27    6582.43
```

Exercises

1. Rewrite the above program to find the maximum value of the function

$$f(x) = -4x^4 - 3x^3 + 2x - 1.$$

2. Using a FOR . . . loop, write a program to evaluate the polynomial at the following values for $x : x = 1, 2, 4, 10, 30$.

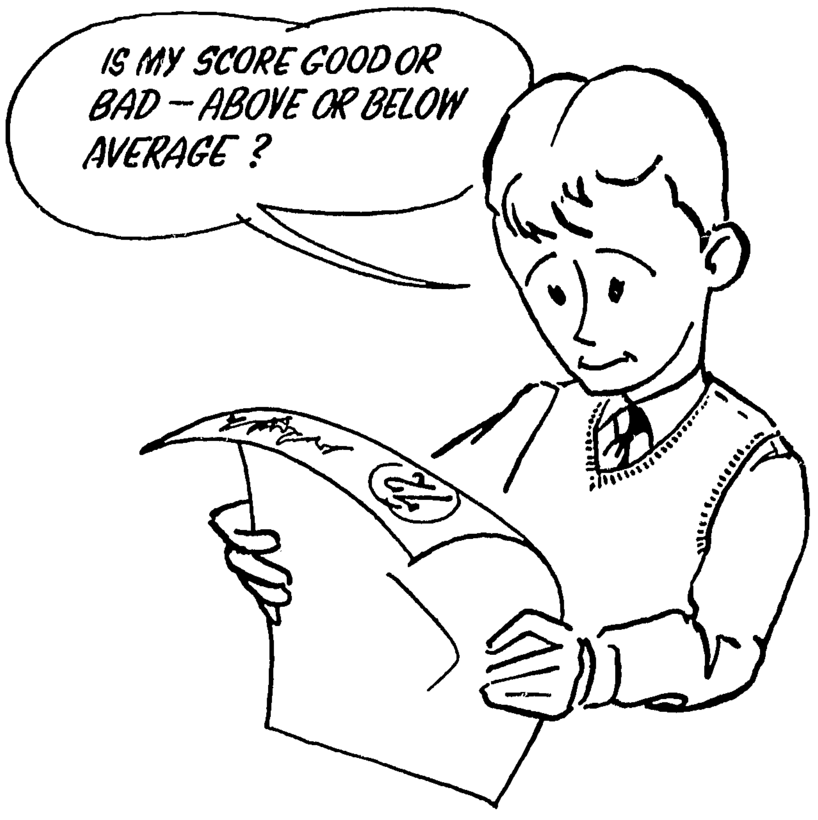
$$f(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots + \frac{x^{20}}{20!}.$$

Problem 3—Mean and Standard Deviation

Given a set of test scores, write a program to compute the mean and standard deviation of the data.

Review: The defining formulas for the means, designated as x, and the standard deviation, s, are given below:

$$\text{(i)} \qquad \overline{x} = \frac{\sum_{i=1}^{N} x_i}{N} = \frac{x_1 + x_2 + x_3 + \dots + x_N}{N}.$$



(ii) $$s^2 = \frac{\sum_{i=1}^{N} (x_i - \bar{x})^2}{N} \rightarrow s = \sqrt{\frac{\sum_{i=1}^{N} (x_i - \bar{x})^2}{N}}.$$

(Note: When sample data are used to estimate s^2 for a larger population, the denominator used is $N - 1$ rather than N.)

Generally the following computational formula is used for s^2, rather than the definitional form given in (ii) above. This computational formula can be readily derived from (ii) by expanding and simplifying.

(iii) $$s^2 = \frac{N \sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i \right)^2}{N^2},$$

and s is the square root of this quantity.

With formulas (i) and (iii) one can compute the mean and standard deviation by summing the values of x_i and x_i^2 and then substituting these summations in the appropriate formulas.

To approach the problem stated initially, we now need to write a program to take any set of input data and perform the indicated calculations. You may wish to make an attempt at solving this problem before reading the following discussion. Consider the scores for this problem to be the following: 71, 65, 70, 85, 90, 95, 61, 73, 81, 88, 75, 75, 91, 66, 69. (Your own program will probably deviate slightly from the sample that will follow.)

To input the data given in our problem, an appropriate technique is to use the READ . . . and DATA . . . statements. Now you must decide what you wish the computer to do with each data value. In this problem it seems our first step is to sum the x's and the x^2's; we shall call these sums M and S, respectively. To have the computer do the summing, it is necessary to start with the values of M and S each equal to zero and then increase them by the appropriate x_i and s_i^2 values. At this time it is also appropriate to input N; this can be done with a READ . . . , an INPUT . . . , or a LET . . . statement. (Can you write the program now?)

Your program should look something like the following one.

Program H

10 LET N = 0 — A counter N is established to enable the computer to keep track of the number of scores in the DATA statement.

20 LET M = 0
30 LET S = 0 — Statements 20 and 30 set the variables M and S, arbitrary choices for variables, equal to zero.

40 READ X — Now we wish to take each piece of data (in the DATA . . . statement, 50) and do the calculations given in Statements 70 and 80.

50 DATA 71, 65, 70, 85, 90, 95, 61, 73, 81, 88, 75, 75, 91, 66, 69, 9999

(Note that 9999 is not one of our original test scores.—The use of this value or one like it, which is often referred to as a "flag," will be pointed out in the discussion of the next statement.)

60 IF X > = 9999 THEN 100

Statement 60 asks if the value of X has reached the large "extra" value in the DATA . . . statement. In the event $x \leqq 9999$ then the computer will continue with the next statement in the program. If, however, $x \geqq 9999$ then the computer will jump to Statement 100, which is the first of our two PRINT . . . statements. The technique of including the 9999 or some other "flag" is essential to the program. If such a technique is not used, the computer will automatically go to END when out of data in 60; that is, when asked to READ X and there is no value left for X, the computer will go to END.

`65 LET N = N + 1` This increments the counter N by 1.

`70 LET M = X + M` This statement increases M by X for each successive value of X; that is, the final M will be the sum of the X's.

`80 LET S = X ↑ 2 + S` Do you see what happens to S? It increases by X^2 for each successive value of X. Thus the final S will be equal to the sum of the X's.

`90 GO TO 40` Statement 90 sends us back to 40, where the next value of X is read, and the interim statements are executed again until X = 9999 is encountered.

`100 PRINT M, N, "MEAN = ," M/N`

Statement 100 tells the computer to PRINT M, N, the phrase "MEAN = ," and M/N. (This is the application of formula (i).) Note that we can include letters or words in our output merely by enclosing them in quotation marks in a PRINT . . . statement. You may not wish to print out all this information; this is up to the discretion of the programmer.

`110 PRINT S, N, "S.D. = ," SQR((N * S − M ↑ 2)/(N ↑ 2))`

Statement 110 instructs the computer to print the values of S and N, "S.D. = ," and the square root of the number within the parentheses. The computations come from formula (iii). (Note that this last calculation could be simplified somewhat by taking just the square root of the numerator and dividing this by N.) The BASIC subroutine for square root was introduced in Section I of this booklet. Recall that to use this function it is only necessary to use the letters SQR and insert the quantity to be operated upon within parentheses.

The program is repeated below, and the output is given.

Program H

```
10 LET N = 0
20 LET M = 0
30 LET S = 0
40 READ X
50 DATA 71, 65, 70, 85, 90, 95, 61, 73, 81, 88, 75, 75, 91, 66, 69, 9999
60 IF X > = 9999 THEN 100
65 LET N = N + 1
70 LET M = X + M
80 LET S = X ↑ 2 + S
90 GO TO 40
100 PRINT M, N, "MEAN =," M/N
110 PRINT S, N, "S.D. =," SQR((N * S − M ↑ 2)/(N ↑ 2))
120 END
```

Output

```
 1155    15    MEAN =     77
90519    15    S.D.   =   10.2762
```

The above program can be easily altered for a different set of data by rewriting Statement 60.

Exercises

1. Write a program to be used with the program already written that will translate each raw score x_i into an appropriate z-score. The z-score translation changes the given distribution into a new distribution with mean 50 and standard deviation 10. The translational formula is

$$z_i = 10\left(\frac{x_i - \overline{x}}{\text{S.D.}}\right) + 50.$$

   Use S.D. = 10.3. (Hint: In this problem you will need to PRINT . . . after each translation.)
2. Write a program to PRINT the sum of sets of N consecutive odd integers—that is, 1, $1 + 3 = 4$, $1 + 3 + 5 = 9$, etc.—for N = 1 to 15.

ogue

THIS booklet is an introduction to the use of a problem-oriented computer language in the teaching of mathematics. For the sake of simplicity, no attempt has been made to utilize all the available features of the BASIC language. These features are available to the reader who wishes to refer to the manuals listed in the bibliography.

Besides BASIC there are currently in existence several other problem-oriented computer languages. Examples of these are JOSS, TELCOMP, ITTRAN, CUPL, ESI, CAL, INSTRUCTRAN, FORTRAN, and ALGOL. PL1 is a new one—available to a limited extent but still under development. The COBOL language, which is in common use, is primarily for the business type of problems.

There are also many special program languages. LISP is especially designed for handling lists of elements, and it is one of many such. COGO and STRESS are languages that have been especially designed for civil engineering problems. It is highly probable that we will see in the future many more especially designed computer languages to solve particular classes of problems.

"Flow charting" can also be called a language. This language is, in essence, a systematic and reasonably complete method of outlining the method and steps in solving a problem. It employs a sequence of boxes, circles, and other such closed figures connected and interconnected in such a way that the problem solution is laid out by their use. This is accomplished by having each box contain a verbal or mathematical expression of one or more steps of the problem. This scheme has the advantage that it does not in any way depend on the particular machine to be used or on any particular machine language.

Many people think of flow charting as a necessary prerequisite to computer programming. It is certainly an aid in programming complex mathematical problems. The degree of complexity of the problem and the skill of the programmer help determine the extent to which flow charts are used in problem solution.

These languages are at the very heart of what is now called computer science, which is crystallizing out as a separate discipline and being included in the college curriculum. Thus teachers now have an opportunity to receive more assistance from colleges where this discipline is being incorporated. It is expected that these curricula will soon include explicit courses for teachers.

Bibliography

THE following bibliography is not intended to be complete. Rather, it is given to suggest sources of information relevant to curriculum utilization of computer systems.

ALBRECHT, ROBERT L., and MARA, WALTER. *Computer Math for High School.* Reading, Mass.: Addison-Wesley Publishing Co., scheduled for publication August 1968.

CONTROL DATA CORPORATION. *The Teacher-Student Approach to Computer Programming Concepts,* Vols. I and II. The Corporation, 1963.

W. H. FREEMAN AND COMPANY. *Information* (a "Scientific American" book). San Francisco: The Company, 1966.

GENERAL ELECTRIC COMPANY. BASIC *Language, Reference Manual.* The Company, 1967.

———. *Introduction to Programming in* BASIC: *An Elementary Instruction Guide.* The Company, 1967.

HOFFMAN, WALTER, *et al.* "Computers for School Mathematics," *The Mathematics Teacher,* LVIII (May 1965), 393–401.

JOHNSON, DONOVAN A., and RISING, GERALD R. *Guidelines to Teaching Mathematics.* Belmont, Calif.: Wadsworth Publishing Co., 1967.

NATIONAL COUNCIL OF TEACHERS OF MATHEMATICS. *Computer Facilities for Mathematics Instruction.* Washington, D.C.: The Council, 1967.

———. *Computer Oriented Mathematics: An Introduction for Teachers.* Washington, D.C.: The Council, 1963.

SCHOOL MATHEMATICS STUDY GROUP. *Algorithms, Computation and Mathematics.* Stanford, Calif.: Stanford University, 1965.

[illegible]swers to Exercises

Section I

Introductory Problem

Exercise 1

To extend C, the program should look like the following. Extensions for D_1 and D_2 make use of similar changes.

Program

```
 5 READ N
10 LET X = 2.0
20 IF X ↑ 2 > N THEN 50
30 LET X = X + .1
40 GO TO 20
50 PRINT X − .1, (X − .1) ↑ 2,
                        X, X ↑ 2
60 GO TO 5
70 DATA 7, 14, 21
80 END
50 PRINT X − .1, (X − .1) ↑ 2, X, X ↑ 2
RUN
```

Note that this program fails for N < 4. You may wish to change the LET . . . statement to overcome this. Also, the increment of .1 is quite small, if the first N is something like 9980. You may wish to improve your program to handle this.

Output

```
2.6    6.76    2.7     7.29
3.7   13.69    3.8    14.44
4.5   20.25    4.6    21.16
OUT OF DATA IN 5
```

Exercise 2

Program

```
10 READ X
20 PRINT SQR(X)
30 GO TO 10
40 DATA 7, 14, 21
50 END
RUN
```

Output

```
2.64575
3.74166
4.58258
OUT OF DATA IN 10
```

Exercise 3

Program

```
10 READ N
20 LET X = 2.0
30 LET Q = N/X
40 IF X > Q THEN 70
50 IF Q − X < .0001 THEN 100
60 GO TO 80
70 IF X − Q < .0001 THEN 100
80 LET X = (Q + X)/2
90 GO TO 30
100 PRINT X, Q, X * Q − N
105 GO TO 10
110 DATA 7, 14, 21
120 END
RUN
```

Note the need for both Statement 50 and Statement 70. The difference between Q and X may be great but negative, and the approximation would be such that this would be the printed result even though incorrect.

Output

```
2.64575     2.64575     −7.45058 E−9
3.74166     3.74166     −1.49012 E−8
4.58258     4.58257     −2.98023 E−8
```

Section II

Problem 1—Divisors of a Positive Integer

Exercise 1

Program

```
10 READ M
15 PRINT M
18 LET S = INT(SQR(M))
20 FOR N = 1 TO S
30 LET Q = INT(N/N)
35 LET R = M - N * Q
40 IF R <> 0 THEN 60
50 PRINT N, Q
60 NEXT N
70 GO TO 10
80 DATA 36, 42, 851
99 END
RUN
```

Output

```
 36
  1      36
  2      18
  3      12
  4       9
  6       6
 42
  1      42
  2      21
  3      14
  6       7
851
  1     851
 23      37
OUT OF DATA IN 10
```

Exercise 2

Program

```
10 READ M
15 PRINT M
```

```
20 LET N = 1
25 LET Q = INT(M/N)
30 IF N > Q THEN 10
35 LET R = M - N * Q
40 IF R <> 0 THEN 50
45 PRINT N, Q
50 LET N = N + 1
55 GO TO 25
80 DATA 36, 42, 851
99 END
RUN
```

Output

```
 36
  1      36
  2      18
  3      12
  4       9
  6       6
 42
  1      42
  2      21
  3      14
  6       7
851
  1     851
 23      37
OUT OF DATA IN 10
```

Exercise 3

Program

```
10 READ M
15 IF M <> INT(M) THEN 10
20 IF M < 2 THEN 10
25 LET S = INT(SQR(M))
30 FOR N = 2 TO S
35 LET Q = INT(M/N)
40 LET R = M - N * Q
45 IF R = 0 THEN 65
50 NEXT N
55 PRINT 1, M
60 GO TO 10
65 PRINT 1, M, N
70 GO TO 10
80 DATA 3.7, 0, 1, 2, 3, 4, 15, 17
99 END
RUN
```

Output

```
1    2
1    3
1    4   2
1   15   3
1   17
OUT OF DATA IN 10
```

Note that no output occurred for M = 3.7, 0, or 1, since these values violated the conditions for input.

Exercise 4

(3 alternatives)

Exercise 4A

Program

```
10 READ A, B
15 FOR N = 1 TO A
20 LET Q1 = INT(A/N)
25 LET R1 = A - N * Q1
30 LET Q2 = INT(B/N)
35 LET R2 = B - N * Q2
40 IF R1 <> 0 THEN 55
45 IF R2 <> 0 THEN 55
50 LET G = N
55 NEXT N
60 PRINT A, B, G
65 GO TO 10
70 DATA 45, 36, 34, 26, 50, 25, 97, 6, 25, 50
99 END
RUN
```

If both remainders are 0, then N is *a* divisor of both A and B. Since N increases by 1 each time, G will be the largest such divisor at the termination of the FOR . . . loop.

Output

```
45    36     9
34    26     2
50    25    25
97     6     1
25    50    25
OUT OF DATA IN 10
```

Exercise 4B

Program

```
10 READ A, B
15 FOR N = A TO 1 STEP - 1
20 LET Q1 = INT(A/N)
25 LET R1 = A - N * Q1
30 LET Q2 = INT(B/N)
35 LET R2 = B - N * Q2
40 IF R1 <> 0 THEN 50
45 IF R2 = 0 THEN 60
```

Since N starts at A and then decreases by 1 each time through the loop, the first time that *both* R1 and R2 are 0 on division by N the result will be the desired G.C.D.

```
50 NEXT N
60 PRINT A, B, N
70 GO TO 10
80 DATA 45, 36, 34, 26, 50, 25, 97, 6, 25, 50
99 END
RUN
```

Output

```
45      36       9
34      26       2
50      25      25
97       6       1
25      50      25
OUT OF DATA IN 10
```

Exercise 4C

Program

```
10 READ A, B
15 LET M = A
20 LET N = B
25 LET Q = INT(M/N)
30 LET R = M - N * Q
35 IF R = 0 THEN 55
40 LET M = N
45 LET N = R
50 GO TO 25
55 PRINT A, B, N
60 GO TO 10
70 DATA 45, 36, 34, 26, 50, 25, 97, 6, 25, 50
99 END
RUN
```

Output

```
45      36       9
34      26       2
50      25      25
97       6       1
25      50      25
OUT OF DATA IN 10
```

Problem 2—Maximizing an Area

Exercise 1

Program

```
10 LET S = 0
20 FOR I = 1 TO 3
30 LET R = 10/10 ↑ I
```

```
40 LET A = ((-4 * S - 3) * S * S + 2) * S - 1
```

The polynomial can be written in this factored synthetic substitution form (see Statement 40), and this reduces the total number of operations to be performed; an additional factor to consider is that the computer is relatively slow in working exponentials.

```
45 LET P = S + R
```

We use Statement 45 to save calculating time and computing time. We calculate P once and then use P instead of S + R in Statement 50.

```
50 LET B = ((-4 * P - 3) * P * P + 2) * P - 1
60 IF B > A THEN 75
70 GO TO 100
75 LET C = A
80 LET A = B
90 LET S = S + R
95 GO TO 45
100 PRINT S - R, C, S + R, B
120 LET S = S - R
130 NEXT I
140 END
RUN
```

Output

```
-1          0           1        -6
  .3      - .5134       .5      - .625
  .36     - .487153     .38     - .488021
```

Exercise 2

Program

```
10 READ X
20 LET S = X + 1
30 LET F = 1
40 FOR Z = 2 TO 20
50 LET F = F * Z
60 LET S = S + X ↑ Z/F
70 NEXT Z
80 PRINT X, S
90 GO TO 10
100 DATA 1, 2, 4, 10, 30
110 END
RUN
```

Output

```
 1     2.71828
 2      7.38906
 4      54.5981
10      21991.5
30      3.77068 E 11
OUT OF DATA IN 10
```

Problem 3—Mean and Standard Deviation

Exercise 1

Program

```
10 READ X
20 PRINT 10 * (X - 77)/10.3 + 50
30 GO TO 10
40 DATA 71, 65, 70, 85, 90, 95, 61, 73, 81, 88, 75, 75, 91, 66, 69
50 END
RUN
```

Output

```
44.1748
38.3495
43.2039
57.767
62.6214
67.4757
34.466
46.1165
53.8835
60.6796
48.0583
48.0583
63.5922
39.3204
41.2621
OUT OF DATA IN 10
```

Exercise 2

(2 alternatives)

Exercise 2A

Program

```
10 LET S = 0
20 READ X
30 DATA 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29
40 LET S = S + X
50 PRINT S
60 GO TO 20
```

```
70 END
RUN
```

Output

```
  1
  4
  9
 16
 25
 36
 49
 64
 81
100
121
144
169
196
225
OUT OF DATA IN 20
```

Exercise 2B

Program

```
10 LET S = 0
20 FOR X = 1 TO 29 STEP 2
30 LET S = S + X
40 PRINT S
50 NEXT X
60 END
RUN
```

Output

```
  1
  4
  9
 16
 25
 36
 49
 64
 81
100
121
144
169
196
225
```