RA

stichting mathematisch centrum

MC

REKENAFDELING RA CR 27/72 OCTOBER

JACK ALANEN

SCIENTIFIC PROGRAM ANALYSIS TECHNIQUES

Preliminary edition

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

Important

This report is a preliminary version and, as such, contains both errors and omissions. The author invites readers to submit corrections and criticisms of this preliminary version.

Acknowledgements

This report on scientific program analysis techniques was the basis for a syllabus for that part of the course Basiscursus Wetenschappelijk Projectprogrammeur given by the author at the Mathematical Center from January - June, 1972. J.W. de Bakker enlisted me to teach part of this course for scientific applications programmers and encouraged me to prepare the syllabus. In addition to him I am indebted to Andy Tanenbaum and Leo Geurts for their helpful comments on the syllabus contents.

Much of the material herein on algorithmic analysis is based upon Professor D.E. Knuth's important writings (see References).

My students made many corrections to the syllabus and I have included the computer solutions to several problems as programmed by A.M.B. de Jong and H.G. Mulder.

I acknowledge Astrid Schuyt-Fasen for her fine mathematical typing; also, thanks to D. Zwarst and J. Suiker for their expert printing.

Table of Contents

6. Measures of program performance. Reasons for and tradeoffs between performance measures. Absolute versus comparative measures. Disclaimers. Robustness. Portability. Ease of expression. Accuracy. Reliability. Adaptability. Closing remarks. Exercises with solutions.

References

1. Introduction

What is computer programming? Most programmers view their work as the construction of algorithms and their expression in a computer language. They usually first create or find (from textbooks, program libraries, etc.) an algorithm to solve a stated problem. Then they specify the algorithm in the rigorous and unambiguous form of a computer program. Finally, the program is debugged by running it with test inputs for which the output is known. A few programmers even document the program before moving to their next problem!

In outline form we have:

Steps in programming (traditional view)

1. Understand the problem. (What is the input and output? Are there computer time/memory restrictions? How often will the program be used? Etc.)
2. Find an algorithm. (See definition below.)
3. Construct a computer program. (Express the algorithm in a computer language.)
4. Debug and test the program. (Translate it to detect syntax errors; run it with representative inputs for which the output is known to detect semantic errors.)
5. Document the above. (Rarely done!)

Features of an algorithm

An algorithm is a finite set of rules for solving the problem. It has five important features:

1. Finiteness. It terminates after a finite and reasonable number (say $10! \approx 3\frac{1}{2}$ million) of steps.
2. Definiteness. Each step is precisely defined so that the actions to be carried out are rigorously and unambiguously specified for each case.
3. Input. The ≥ 0 data items.

4. Output. The ≥ 1 answers.

5. Effectiveness. The operations to be done can be performed by your computer exactly and in a reasonable length of time.

This traditional approach to computer programming has successfully produced much software; however, there is definitely need for more efficient production of more efficient software. The newly discussed area of "software engineering" (Naur and Randell 1969; Buxton and Randell 1970; Turski 1971; Bauer 1972) seeks to meet this need. Employing concepts such as "structured programs" (Dijkstra 1970), "stepwise refinement" (Wirth 1971), and "algorithmic analysis" (Knuth 1971), attempts are being made to turn the "art" of programming into the "science" of programming. Unfortunately, it is difficult to learn and adopt these new programming concepts and techniques when your education and experience are grounded in the traditional notions of programming.

For example, years ago Dijkstra (1968) rejected goto statements (branches, jumps) as being logically unnecessary, a frequent source of error, and demanding an unnatural mode of thought. After his short note "Go to considered harmful" appeared, Dijkstra was immediately rebutted by a colleague (Rice 1968) who was worried about effects the note would have on "young, novice programmers". It is currently a well-justified view that the use of goto statements in programming is neither desirable nor necessary; Wulf (1971) reports on favorable long-term experiences with a programming language (Bliss) which has absolutely no goto. Naturally, a language such as FORTRAN which has no compound statements forces the programmer to use goto statements. For these reasons ALGOL 60 without goto will be employed in this syllabus.

We will view the activity of programming as the construction and analysis of computer algorithms. By analysis of a computer algorithm is meant, roughly, an investigation to answer the two questions:

1. Does the algorithm work?
2. Is the algorithm any good?

Obviously, a programmer who constructs a well-structured algorithm will find its analysis facilitated. And, conversely, analysis of an algorithm will often lead to the construction of an improved algorithm.

Some theory and techniques currently known for answering the above two questions are the subject of sections 2-6. An example algorithm is analyzed in section 2. The first question is answered by a proof of correctness as described in section 3. A program correctness proof does not consist of testing the program with representative input data. As Dijkstra remarks, "Testing is a very inefficient way of convincing oneself of the correctness of a program". The second question may be answered by evaluating the performance of the algorithm, particularly with respect to running time (section 4) and storage requirements (section 5). To show that a particular algorithm is optimal in the sense that it involves the fewest number of computational steps in a precisely defined class or it uses the minimum memory possible or it maintains a desired accuracy is, in general, very difficult. But to demand, seek, and prefer computational efficiency in an algorithm can yield significant savings in both computer and programming time. Moreover, the solution of a problem may actually be impossible before development of an "efficient" algorithm. For example, to determine that a forty digit integer n is prime by successively dividing it by $2,3,4,\ldots,\sqrt{n}$ is impractical on a contemporary computer; yet algorithms for proving the primality of such an n in a few seconds of computer time exist (Knuth 1969). Much of the research in artificial intelligence today is vitally concerned with defining "good" algorithms for chess-playing, picture analysis, theorem-proving, and other problem-solving areas.

Section 6 covers some additional measures of computational performance besides running time and storage. For instance, aspects of a program such as its accuracy and portability are receiving deserved attention nowadays. By asking questions about portability, ease of expression, stability, accuracy and precision, reliability, adaptability, and so on, a programmer can approach problems of how to choose the programming language, select/build library procedures, adapt existing programs, etc.

This syllabus emphasizes the analysis rather than the construction of computer programs, because it seems easier to discuss whether a program works or is any good, than to describe how to construct good, working programs. As Polya (1945) said: "A person who behaves the right way does not care to express his behavior in clear words and, possibly, he cannot express it so".

Nevertheless, construction and analysis of programs are the two intimately related aspects of programming. We now construct two computer programs using stepwise refinement. The first example shows how transparent an algorithm can become by constructing it carefully. The second example shows how during the construction, an algorithm can be found deficient in efficiency.

A program can be constructed by the process of successive refinements (Wirth 1971). Each refinement is a more detailed specification of the previous step and refinement terminates when all instructions are expressed in terms of a computer or programming language. A simple example to exhibit program development by stepwise refinement follows:

```
1. begin Solve the Problem end

2. begin comment Problem statement;
                 Read input;
                 Compute solution;
                 Write output
   end

3. begin comment Tabulate $M(n) = \lfloor \log_2 n \rfloor + v(n) - 1$ versus n-1 for various
                 values of n, where v(n) = number of ones in the binary
                 representation of n (see exercise 2.1);
         until   No More Input repeat
                 begin Input;
                       for n:= nMIN step 1 until nMAX do
                       begin Compute M(n);
                             Output
                       end
                 end
   end
```

4. Refinement of "Input" in step 3:

```
Input: nMIN:= read; output("nMIN",nMIN);
       nMAX:= read; output("nMAX",nMAX);

where  procedure output (text, variable); string text;
       begin nlcr; printtext (text); printtext ("="); print (variable) end;
```

RULE: Whenever an input value is read in, *immediately* output it. Otherwise, how do you know what problem you are (or were) solving?

5. Refinement of "Compute M(n)" in step 3:

```
integer Mn, n;
Compute Mn: Mn:= log 2(n) + v(n) - 1;

where   real procedure log 2(x); value x; integer x;
        log 2:= ln(x)/ln(2.0);
```

because $\log_a x = \log_b x/\log_b a$.

6. Refinement of "Output" in step 3:

```
Output: nlcr; output("n",n); output("Mn",Mn); output("n-1",n-1);
```

7. Refinement of "v(n)" in step 5:

```
integer procedure v(n); value n; integer n;
begin comment v(n) = Number of 1's in the binary representation of n;
      integer Answer;
      Answer:= 0;
      for n:= n, n÷2 while n≠0 do
      if odd(n) then Answer:= Answer + 1;
      v:= Answer
end;
```

8. Our final refinement (an MC ALGOL 60 program) appears in Figure 2.4.

Note that only at refinement step 5 did we make a decision about data types. In the final program, variables nMIN and nMAX could have been type *real*. Other equally good solutions to our example problem could be developed by the method of stepwise program refinement; nevertheless, the above detailed elaborations of our relatively short program indicate that programming can be done by a careful, gradual development.

We next program a nontrivial problem using the method of stepwise refinement:

```
1. Comment For various values of y and n, compute $y^n$ by the S-and-X Binary
           Method;
   until No More Input repeat
                  begin Input;
                        yTOn:= Exponentiation (base, exponent);
                        Output
                  end;
```

```
2. Refinement of the procedure Exponentiation in step 1;

   integer base, exponent;
   if base = 0 ∨ exponent ≤ 0 then
        begin printtext ("Illegal argument to y↑n procedure");
              output ("y", base); output ("n", exponent)
        end
   else begin
        Initialize;
        for Bit:= Nextbit of exponent while There is a bit do
        begin Square Z;
              if Bit = 1 then Multiply Z by base
        end;
        Exponentiation:= Z
        end;
```

```
3. Refinement of statement after the else in step 2:

   Z:= base; i:= Number of bits(k) in exponent;
   for i:= i-1 while i≥0 do
   begin Bit:= The i-th bit ($d_i$) of exponent, which is
         $d_k d_{k-1} \ldots d_1 d_0$ in binary notation;
         Z:= Z×Z;
         if Bit = 1 then Z:= Z × base
   end;
```

4. Step 3 is difficult to refine in ALGOL 60 because

$$d_i = 1 \text{ \underline{if and only if} } \text{base} \div 2^i \text{ is odd}$$

cannot be determined efficiently. In machine language this quotient can be found by shifting. Thus we are lead to alter the S-and-X Binary Method so that it is based on a right-to-left scan of n; see Algorithm R in section 2 for details.

EXERCISES

1.1. Restate the following ALGOL 60 program without using goto statements:

```
comment B1 and B2 are Boolean procedures;
loop: A[i]:= e;
if B1 then begin e:= e+1; go to loop end;
if B2 then begin i:= i+1; go to loop end;
```

1.2. Restate the following ALGOL 60 program without using goto statements:

```
comment B1 and B2 are Boolean procedures;
loop: A[i]:= e;
if B1 then begin e:= e+1; goto loop end
else if B2 then begin i:= i+1; goto loop end;
```

1.3. To measure computer performance we might use the standard of measurement, n/t, where m = size of the memory and t = basic add instruction time. This ratio measures roughly the capacity both to hold and to process information. Perform an order of magnitude calculation to see how this criterion of computer performance has increased over the past twenty years.

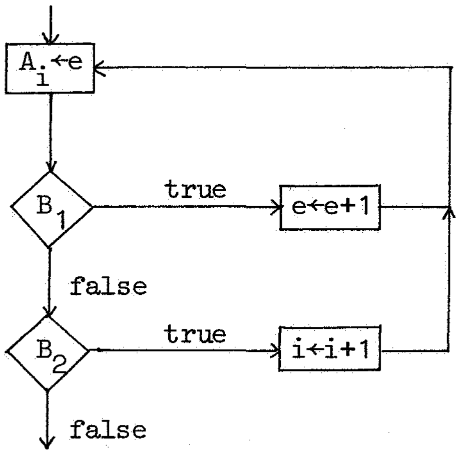
1.4. Suppose you wanted to do exhaustive testing of a procedure whose input is a binary matrix (integer array A[1:n,1:n] with every element either 0 or 1). How many cases must you test? When would n be too large for testing all these cases on your computer?

1.5. Write a general ALGOL 60 program which consists of r nested for statements, where r is a parameter. For example, when r = 3 the program should have the effect of:

```
for k[1]:= 0 step 1 until K[1] do
for k[2]:= 0 step 1 until K[2] do
for k[3]:= 0 step 1 until K[3] do
begin comment Compute using k[1], k[2], k[3]; end;
```

SOLUTIONS

1.1.



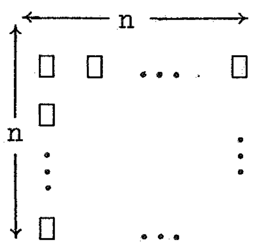
```
Boolean b1,b2;
loop: b1:= true;
while b1 do begin b2:= false; A[i]:= e;
                  if B1 then e:= e+1 else b2:= true;
                  while b2 do if B2 then begin i:= i+1;
                                               b2:= false
                                         end
                              else b1:= b2:= false
           end;
```

1.3.

Year	Computer	t(sec)	m(words)	comments
1951	Univac I	10^{-3}	1K	First commercially available computer.
1971	IBM 360	10^{-7}	128K	Faster and bigger machines exist.

Thus computer performance (m/t) has increased by a factor of ten (an order of magnitude) every three years. This is a conservative estimate!

1.4.



There is a 0 or 1 in each box and hence $(2^n)^n = 2^{n^2}$ cases to test. When n = 5, this number already exceeds 3 billion.

```
1.5. comment Assume K[1],...,K[r] ≥ 0 and r > 0;
     integer j;
     for j:= 1 step 1 until r do k[j]:= 0;
     compute: begin comment Use k_1,...,k_r; end;
              j:= r;
     loop: k[j]:= k[j] + 1;
           if k[j] ≤ K[j] then goto compute;
           k[j]:= 0; j:= j-1;
           if j > 0 then goto loop;
```

As a further exercise, the reader is asked to rewrite the above in a goto-less form.

2. An example: Evaluation of powers

In this section we shall program and analyze the "binary method" for computing y^n, given y and n, where n is a positive integer. Many ALGOL (y↑n) and FORTRAN (y**n) compilers obtain the answer with successive inline multiplications. For example, to compute y^{13} we could simply start with y and multiply by y twelve times. But is is possible to obtain the same answer with only five multiplications, if we again start with y and then "square, multiply by y, square, square, and multiply by y". In other words, we obtain

$$y^{13} = (((y^2)y)^2)^2y$$

by successively computing y^2, y^3, y^6, y^{12},y^{13}.

The same idea applies to any value of n in the following way: Given the exponent n in binary representation (e.g., $n = 13_{10} = 1101_2$), replace each "1" by "SX" and each "0" by "S" (e.g., 1101 → SXSXSSX), Cancel the "SX" at the left end and then interpret the resulting string as ordered instructions, where "S" = "Square" and "X" = "Multiply". This general algorithm for evaluation of powers is known as the "binary method".

Since we have specified the binary method in the English language, there is the possibility the reader might not understand exactly what the author intended. We must therefore be more "definite"; that is, each step of our algorithm must be precisely defined so that the actions to be carried out are rigorously and unambiguously specified for each case. Consider the following expression of the binary method:

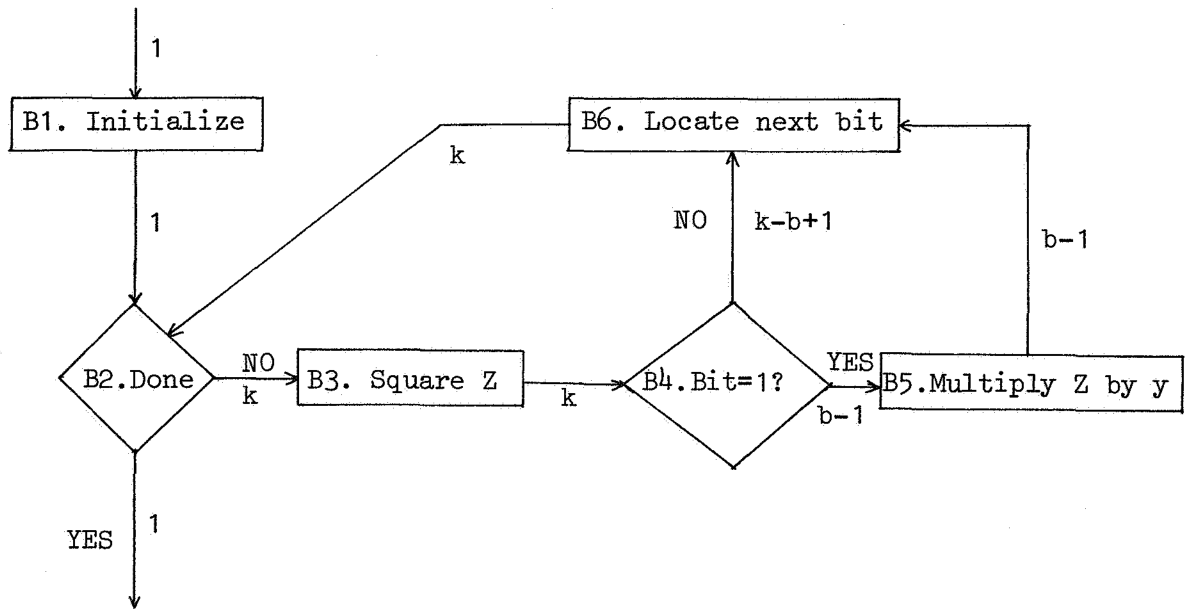
<u>Algorithm</u> B. (Binary method for exponentiation). This algorithm evaluates y^n, where n is a positive integer. Functions f and g are defined below.

```
B1. [Initialize.] Set $Z \leftarrow y$ and $i \leftarrow f(n) - 1$.

B2. [Done?] If $i < 0$, the algorithm terminates, with Z as the answer.

B3. [Square.] Set $Z \leftarrow Z$ times Z.

B4. [Bit = 1?] If $g(n,i) \neq 1$, skip to step B6.

B5. [Multiply.] Set $Z \leftarrow Z$ times y.
```

B6. [Next bit.] Decrease i by one and return to step B2.

Definition: Let $d_k d_{k-1} \ldots d_1 d_0$ be the binary notation for n.
Then $f(n) \equiv k$ and $g(n,i) \equiv d_i$ for $0 \leq i \leq k$.

<u>Figure</u> 2.1. Flowchart for Algorithm B, with the arrows between boxes labelled by the number of times that path will be followed during one run of the Algorithm.



Before we render Algorithm B in a programming language such as ALGOL 60, a "local" analysis of the amount of work it does will be given. "Work" is usually measured in terms of the number of times each step is performed, or how much memory the algorithm needs:

<u>Storage analysis</u>. The S-and-X binary method (Algorithm B) for obtaining y^n requires variable storage only for the inputs y and n, for the current partial result Z, and for the bit index i. This assumes that the functions f and g require no temporary storage; that is, f(n) and g(n,i) must be computable directly from their arguments, without using temporary storage. Hence Algorithm B needs a small, fixed amount of storage for variables.

Frequency analysis. The "profile" (collection of frequency counts) of Algorithm B is easily deduced from Figure 2.1, where each pathway in the algorithm was labelled with the frequency it is traversed.

Figure 2.2. Profile of Algorithm B.

Step	Number of times executed
B1	1
B2	k+1
B3	k
B4	k
B5	b-1
B6	k

This profile gives us information necessary to determine the running time of the algorithm on a particular computer. To complete the analysis we must interpret the quantities k and b. Clearly, k equals $\lfloor \log_2 n \rfloor$, one less than the total number of bits in the binary notation for n. ($\lfloor X \rfloor$ denotes the greatest integer $\leq$ X.) The quantity b equals v(n), the number of ones in the binary representation of n.

We were at the start of this section interested in computing y^n with fewer multiplications than the n-1 required by the serial method. Now we know that the number, M(n), of multiplications required by Algorithm B is precisely $M(n) = \lfloor \log_2 n \rfloor + v(n) - 1$. Thus the execution time in applications with large exponents n can be reduced from order n (serial method) to order log n (binary method). For small values of n, say $n \leq 10$, the bookkeeping time required to evaluate f and g values in Algorithm B exceeds the time saved by fewer multiplications, unless the time for a multiplication is comparatively large. Multiplication would require a significant amount of time if, for example, y was a matrix or polynomial or multiple-precision number, instead of a simple variable.

Several authors asserted that the binary method has "absolute" efficiency, i.e. gives the minimum possible number of multiplications in all cases. The smallest counterexample is n = 15, when M(15) = 6 and yet we can calculate

$$y^{15} = ((y^2y)^2)^2(y^2y)$$

with only five multiplications by using the intermediate result y^2y. This leads us to do a "global" analysis of the entire family of algorithms to evaluate y^n. In particular, we could investigate the "best possible" procedures in this class from the point of view of minimal multiplications required. This has been done by Knuth (1968, section 4.6.3). An optimal procedure for all n is not known. Nevertheless, Figure 2.3 taken from Knuth gives a systematic method to compute y^n in the minimum number of multiplications for every value of $n \leq 100$. Computer tests have shown that this "tree method" is indeed optimal for all $n \leq 100$. In summary, a global analysis of evaluation of powers algorithms shows that the binary method excels the serial method in minimizing multiplications, but the tree method (Figure 2.3) is optimal for most n which occur in practical applications.

To render Algorithm B in ALGOL causes problems because of the functions f and g. This S-and-X binary method requires that the binary representation of n be scanned from left-to-right, while it is more convenient in ALGOL to deduce the binary representation from right-to-left by successively dividing by 2 until zero is reached. That is, if n in binary notation equals $d_k d_{k-1} \ldots d_1 d_0$, then we use the fact that for $0 \leq i \leq k$:

$$g(n,i) = d_i = 1 \ \underline{\text{iff}}\ n \div 2^i \text{ is odd.}$$

Therefore the following Algorithm R, based on a right-to-left scan of n, can be easily translated into ALGOL:

Algorithm R. (Right-to-left binary method for exponentiation.) This algorithm evaluates y^n, where n is a nonnegative integer.

R1. [Initialize.] Set $N \leftarrow n$, $Y \leftarrow y$, and $Z \leftarrow 1$.

R2. [Done?] If $N = 0$, the algorithm terminates with Z as the answer.

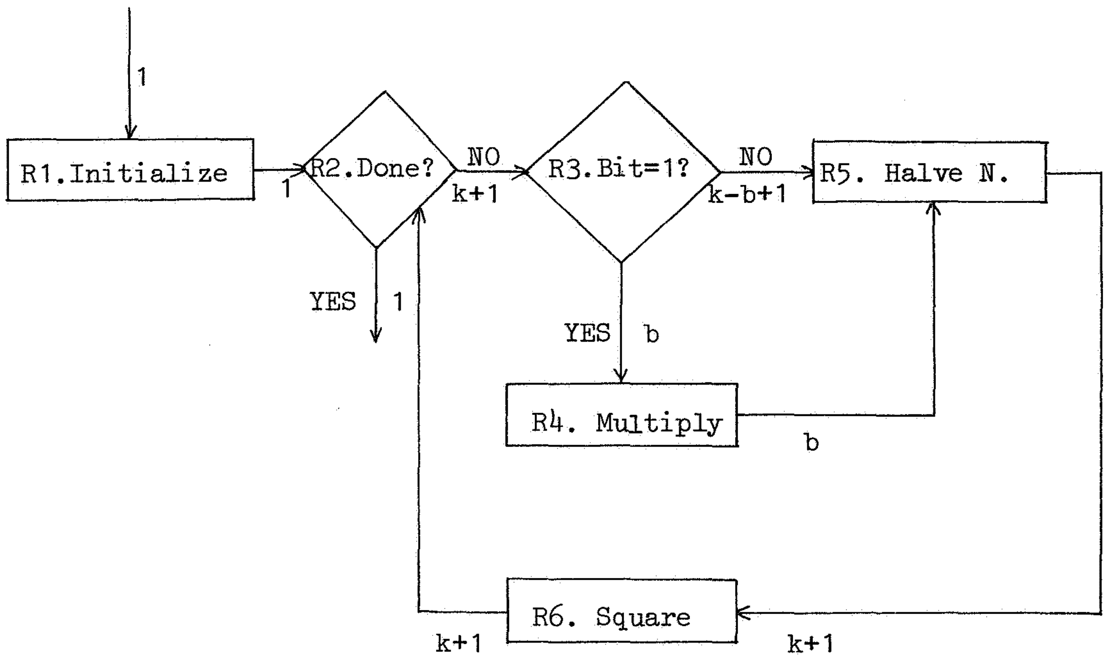
R3. [Bit = 1?] If N is even, skip to step R5.

R4. [Multiply.] Set $Z \leftarrow Z$ times Y.

R5. [Halve N.] Set N $\leftarrow \lfloor N/2 \rfloor$.

R6. [Square.] Set Y $\leftarrow$ Y times Y, and return to step R2.

<u>Figure</u> Flowchart for Algorithm R, with paths labeled by the frequency of their traversal.



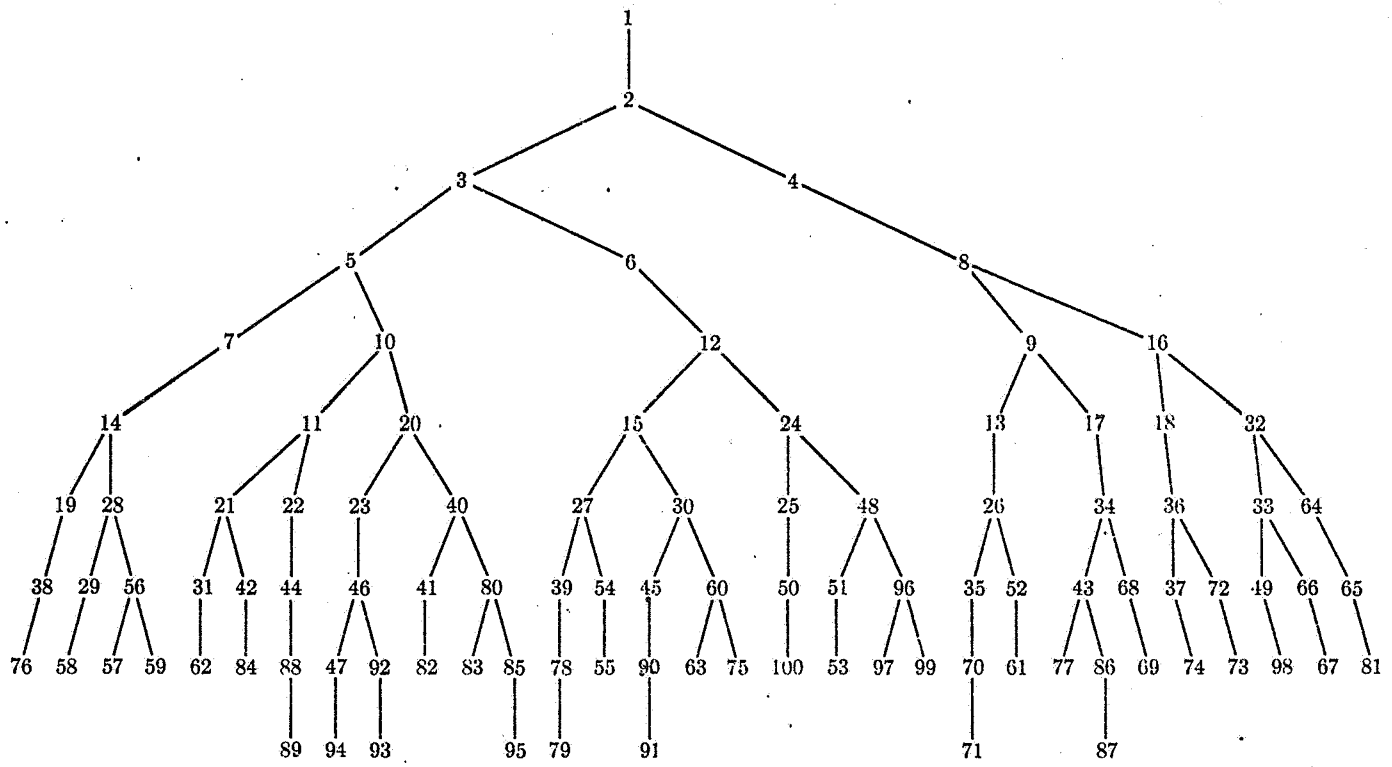


Figure 2.3. A tree which minimizes the number of multiplications, for $n \leq 100$. To calculate y^n, find n in this tree, and the path from the root to n indicates the sequence of exponents which occur in the optimal evaluation of y^n.

Figure 2.4. Program to tabulate $M(n) = \lfloor\log_2 n\rfloor + v(n) - 1$ versus n-1 for various values of n. See exercise 2.1.

```
begin integer Mn, n, nMIN, nMAX;
      procedure output(text,variable); string text;
      begin nlcr; printtext(text); printtext("="); print(variable) end;
      real procedure log 2(x); value x; integer x;
      log 2:= ln(x)/ln(2.0);
      integer procedure v(n); value n; integer n;
      begin comment v(n) = Number of ones in the binary
                    representation of n;
            integer Answer; Answer:= 0;
            for n:= n, n÷2 while n≠0 do
            if odd (n) then Answer:= Answer+1;
            v:= Answer
      end;
      Boolean procedure odd(n); value n; integer n;
      odd:= (n÷2)×2≠n;
      for nMIN:= read while true do
      begin output ("nMIN", nMIN);
            nMAX:= read; output ("nMAX", nMAX);
            for n:= nMIN step 1 until nMAX do
            begin Mn:= log 2(n) + v(n) - 1;
                  nlcr; output ("n", n); output ("Mn", Mn);
                  output ("n-1", N-1)
            end
      end
end
```

Figure 2.5. Profile of Algorithm R.

Step	Number of times executed
R1	1
R2	k+2
R3	k+1
R4	b
R5	k+1
R6	k+1

From the profile of Algorithm R (Figure 2.5), we find that it requires $k + b + 1 = \lfloor \log_2 n \rfloor + v(n) + 1$ multiplications. This is two more than Algorithm B, due to the multiplication by unity in the first execution of step R4 and to the redundant execution of step R6 when N = 0. We next give an ALGOL program for Algorithm R and prove its correctness.

Figure 2.6. Program in an ALGOL dialect for Algorithm R.

```
comment Evaluate y^n for integral n ≥ 0;
[1]  N:= n; Y:= y; Z:= 1;
[2]  while N ≠ 0 do
[3]            begin if odd(N) do Z:= Z * Y;
[4]                  N:= N ÷ 2;
[5]                  if N ≠ 0 do Y:= Y * Y
[6]            end;
```

As an example of this program, consider the steps in the evaluation of y^{23}:

	N	Y	Z
After line 1	23	y	1
After line 3	23	y	y
After line 5	11	y^2	y
After line 3	11	y^2	y^3
After line 5	5	y^4	y^3
After line 3	5	y^4	y^7

	N	Y	Z
After line 5	2	y^8	y^7
After line 3	2	y^8	y^7
After line 5	1	y^{16}	y^7
After line 3	1	y^{16}	y^{23}
After step 5	0	y^{16}	y^{23}

An informal but rigorous correctness proof for this program has to show that all variable keep integer values and that the "inductive assertion":

$$N \geq 0 \ \underline{\text{and}}\ y^n = Z * Y^N$$

always holds before and after execution of the _while_ clause (line 2). Our approach is to show that if this assertion is true before execution of lines 3-6, then it is also true after execution of these four lines.

Correctness proof. After line 1 is executed, the assertion is trivially true because $N = n \geq 0$ and $Z * Y^N = 1 * y^n = y^n$. Suppose next that it holds for fixed values of Z, Y, and N before execution of lines 3-6. If $N = 0$, then the test in the _while_ clause will successfully avoid execution of these three lines and will end the program with $y^n = Z * Y^N = Z * Y^0 = Z$. Otherwise, $N > 0$ and we have two cases depending upon the parity of N:

Case 1. N is odd. Then line 3 multiplies Z by Y, line 4 replaces N with $\lfloor N/2 \rfloor$, and (assuming $N \neq 1$ before line 4) line 5 squares Y so that the net effect is to assign the value

$$(Z*Y) * (Y^{\lfloor N/2 \rfloor})^2$$

to $Z * Y^N$. Had N equaled 1, the test in line 5 would have prevented the squaring of Y so that

$$Z * Y^N = Z * Y\text{, as required.}$$

Case 2. N is even. Then line 4 halves N and line 5 squares Y so that the net effect is to assign the value

$$Z * (Y^{N/2})^2$$

to $Z * Y^N$.

Thus, in both cases, $y^n = Z * Y^N$ remains invariant after execution of lines 3-6.

Clearly, the only change in value to N occurs in line 4 and always results in a new, nonnegative integral value since N is initialized in line 1 to $n \geq 0$. The reason we restricted all variables to integers in our proof was to avoid the complication of accuracy considerations.

Termination. This program terminates because the initial value $n \geq 0$ for N will be repeatedly halved in line 4 until N = 0 in a finite number of steps and then the while clause ends the execution.

EXERCISES

2.1. Tabulate and graph $M(n)$ versus $n-1$ for $n = 1(1)100$.

2.2. Construct and analyze an algorithm which computes y^n in a serial manner (multiplying repeatedly by y). Make comparisons with Algorithm B.

2.3. How is y^{975} calculated by the (i) binary method? (ii) method of Figure 2.3? Can you do it in fewer multiplications?

2.4. (Knuth 1969, 4.6.3.10) Figure 2.3. shows a tree that indicates one way to compute y^n with the fewest number of multiplications, for all $n \leq 100$. How can this tree be conveniently represented within a computer, in just 100 memory locations?

2.5. Let e be a fraction, $0 < e < 1$, expressed in the binary number system as

$$e = (.d_1 d_2 \ldots d_k)_2$$

Design and analyze an algorithm to compute y^e using the operations of multiplication and square-root extraction.

2.6. The "factor method" is a recursive procedure for evaluating y^n based on a factorization of n: If $n=1$, we have y^n trivially. If n is prime, we calculate y^{n-1} and multiply by y. If $n = pg$, where $g > 1$ and p is the smallest prime factor of n, we calculate y^n by first calculating y^p and then raising this quantity to the g-th power.
For example, to calculate y^{55} by the factor method, we first evaluate

$$z = y^5 = y^4 y = (y^2)^2 y,$$

and then form $z^{11} = z^{10} z = (z^2)^5 z$.

Prove that there are infinitely many values of n

a) for which the factor method is better than the binary method;

b) for which the binary method is better than the factor method;

c) for which some other method is better than both the binary and factor methods. (Here "better" means using fewer multiplications.)

2.7. Construct and analyze an MC-ALGOL procedure to find d(n) = the number of decimal digits in the integer $n \geq 0$.

2.8. How does the Binary Method compute y^{31}? Can you compute y^{31} with fewer arithmetic operations if division is allowed? In general, the serial method for evaluating y^n requires about n multiplications; what is the order of magnitude for the number of multiplications required by the Binary Method for a large exponent n? (For example, about how many multiplies when $n = 10^6$?) Prove that the Binary Method does <u>not</u> minimize the number of multiplications required in <u>all</u> cases.

2.9. Construct a recursive version of the binary method for computing y^n.

SOLUTIONS

2.1. See ALGOL program and plotter output on next pages.

2.7. Here are three different solutions (assume integer d, k, D;):

```
(i)    d:= 0; for D:= 0, D+1 while mod(n,10↑D) ≠ n do d:= d + 1;
(ii)   d:= ln(n)/ln(10.0) + 1;
(iii)  k:= 1; d:= 0; for D:= 0, D+1 while n ÷ k ≠ 0 do begin
       k:= 10×k; d:= d+1 end;
```

2.8. $y^{31} = (((y^2y)^2y)^2y)^2y$ (8 multiplications)

$= ((((y^2)^2)^2)^2)^2/y$ (6 operations)

$= (([y^2y]^2y)^2)^2[y^2y]$ (7 multiplications)

The smallest counter-example is:

$y^{15} = ((y^2y)^2y)^2y$ (binary method; 6 multiplies)

$= ([y^2y]^2)^2[y^2y]$ (5 multiplies)

$$\log_2 10^6 \approx \log_2(2^3)^6 = \log_2 2^{18} = 18,$$

or $\log_2 10^6 = 6 \log_2 10 \approx 6 * 3.3 \approx 20.$

Furthermore, $v(10^6) \leq \log_2 10^6$ so that

$M(10^6) = \lfloor \log_2 10^6 \rfloor + v(10^6) - 1 \approx 2 \log_2 10^6 \approx 40.$

2.9. In ALGOL 68 we have:

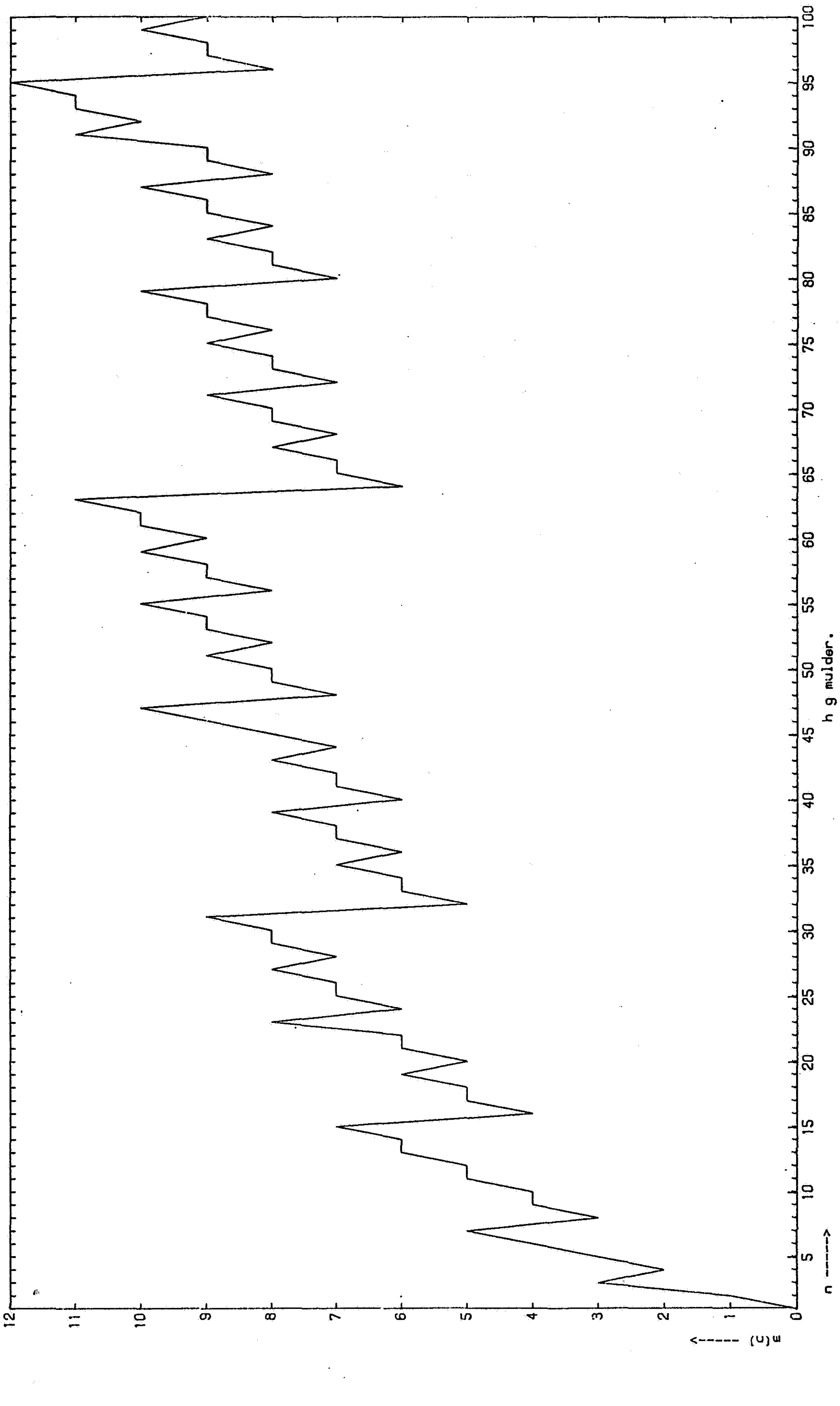
```
op ↑ = (int y, n) int:
if n = 0 then 1
else if n = 1 then y
     else (if odd(n) then y else 1 fi) * (y*y) ↑ (n÷2) fi fi;
```

D2296V.006,H G MULDER,T100,P1000

```
'BEGIN' 'INTEGER' MN,N,NMIN,NMAX,MAXMN;
  'BOOLEAN' 'PROCEDURE' ODD(N); 'VALUE' N; 'INTEGER' N;
  ODD:=(N÷2)*2 ≠ N;
  'PROCEDURE' OUTPUT(TEXT,VARIABLE); 'STRING' TEXT;
  'BEGIN' PRINTTEXT(TEXT); PRINTTEXT("="); PRINT(VARIABLE) 'END';
  'REAL' 'PROCEDURE' LOG2(X); 'VALUE' X; 'INTEGER' X;
  'BEGIN' 'REAL' W; W:=X; LOG2:=LN(W)/LN(2.0) 'END';
  'INTEGER' 'PROCEDURE' V(N);  'VALUE' N; 'INTEGER' N;
  'BEGIN' 'COMMENT' V(N) = NUMBER OF ONES IN THE BINARY REPRESENTATION OF N;
    'INTEGER' ANSWER; ANSWER:=0;
    'FOR' N:= N,N÷2 'WHILE' N≠0 'DO'
    'IF' ODD(N) 'THEN' ANSWER:=ANSWER+1;
    V:=ANSWER
  'END';
  OUTPUT("LN(2)",LN(2)); NLCR;
  'FOR' NMIN:=READ 'WHILE' 'TRUE' 'DO'
  'BEGIN' OUTPUT("NMIN",NMIN);
    NMAX:=READ; OUTPUT("NMAX",NMAX);
    NLCR; NLCR;
    'BEGIN' 'INTEGER' 'ARRAY' GRAPH[NMIN:NMAX];
      MAXMN:=0;
      'FOR' N:= NMIN 'STEP' 1 'UNTIL' NMAX 'DO'
      'BEGIN' MN:=LOG2(N)+V(N)-1;
        NLCR; OUTPUT("N",N); OUTPUT("M(N)",MN);
        OUTPUT("N-1",N-1);
        OUTPUT("LOG2(N)",LOG2(N)); OUTPUT("V(N)",V(N));
        'IF' MN>MAXMN 'THEN' MAXMN:=MN;
        GRAPH[N]:=MN;
      'END';
      PLOTPICTURE(N,GRAPH[N],N,NMAX,9,0,0066,
                  NMIN,NMAX,1. 3800,
"N ------>                                                  H G MULDER,",
                  0,MAXMN,1,2300,"M(N) ------>",PLOTLINE)
    'END'
  'END'
'END'
```

```
LN(2)=+.6931471805601⏨-  0
NMIN=          +1        NMAX=         +100

N=             +1        M(N)=            -0        N-1=            -0        LOG2(N)=                -0       V(N)=                 +1
N=             +2        M(N)=            +1        N-1=            +1        LOG2(N)=                +1       V(N)=                 +1
N=             +3        M(N)=            +3        N-1=            +2        LOG2(N)=+.1584962500721⏨+  1 V(N)=                 +2
N=             +4        M(N)=            +2        N-1=            +3        LOG2(N)=+.2000000000004⏨+  1 V(N)=                 +1
N=             +5        M(N)=            +3        N-1=            +4        LOG2(N)=+.2321928094887⏨+  1 V(N)=                 +2
N=             +6        M(N)=            +4        N-1=            +5        LOG2(N)=+.2584962500725⏨+  1 V(N)=                 +2
N=             +7        M(N)=            +5        N-1=            +6        LOG2(N)=+.2807354922061⏨+  1 V(N)=                 +3
N=             +8        M(N)=            +3        N-1=            +7        LOG2(N)=+.2999999999996⏨+  1 V(N)=                 +1
N=             +9        M(N)=            +4        N-1=            +8        LOG2(N)=+.3169925001443⏨+  1 V(N)=                 +2
N=            +10        M(N)=            +4        N-1=            +9        LOG2(N)=+.3321928094887⏨+  1 V(N)=                 +2
N=            +11        M(N)=            +5        N-1=           +10        LOG2(N)=+.3459431618638⏨+  1 V(N)=                 +3
N=            +12        M(N)=            +5        N-1=           +11        LOG2(N)=+.3584962500721⏨+  1 V(N)=                 +2
N=            +13        M(N)=            +6        N-1=           +12        LOG2(N)=+.3700439718137⏨+  1 V(N)=                 +3
N=            +14        M(N)=            +6        N-1=           +13        LOG2(N)=+.3807354922057⏨+  1 V(N)=                 +3
N=            +15        M(N)=            +7        N-1=           +14        LOG2(N)=+.3906890595605⏨+  1 V(N)=                 +4
N=            +16        M(N)=            +4        N-1=           +15        LOG2(N)=                +4       V(N)=                 +1
N=            +17        M(N)=            +5        N-1=           +16        LOG2(N)=+.4087462841257⏨+  1 V(N)=                 +2
N=            +18        M(N)=            +5        N-1=           +17        LOG2(N)=+.4169925001443⏨+  1 V(N)=                 +2
N=            +19        M(N)=            +6        N-1=           +18        LOG2(N)=+.4247927513446⏨+  1 V(N)=                 +3
N=            +20        M(N)=            +5        N-1=           +19        LOG2(N)=+.4321928094891⏨+  1 V(N)=                 +2
N=            +21        M(N)=            +6        N-1=           +20        LOG2(N)=+.4392317422782⏨+  1 V(N)=                 +3
N=            +22        M(N)=            +6        N-1=           +21        LOG2(N)=+.4459431618641⏨+  1 V(N)=                 +3
N=            +23        M(N)=            +8        N-1=           +22        LOG2(N)=+.4523561956055⏨+  1 V(N)=                 +4
N=            +24        M(N)=            +6        N-1=           +23        LOG2(N)=+.4584962500725⏨+  1 V(N)=                 +2
N=            +25        M(N)=            +7        N-1=           +24        LOG2(N)=+.4643856189774⏨+  1 V(N)=                 +3
N=            +26        M(N)=            +7        N-1=           +25        LOG2(N)=+.4700439718137⏨+  1 V(N)=                 +3
N=            +27        M(N)=            +8        N-1=           +26        LOG2(N)=+.4754887502168⏨+  1 V(N)=                 +4
N=            +28        M(N)=            +7        N-1=           +27        LOG2(N)=+.4807354922057⏨+  1 V(N)=                 +3
N=            +29        M(N)=            +8        N-1=           +28        LOG2(N)=+.4857980995133⏨+  1 V(N)=                 +4
N=            +30        M(N)=            +8        N-1=           +29        LOG2(N)=+.4906890595608⏨+  1 V(N)=                 +4
N=            +31        M(N)=            +9        N-1=           +30        LOG2(N)=+.4954196310391⏨+  1 V(N)=                 +5
N=            +32        M(N)=            +5        N-1=           +31        LOG2(N)=                +5       V(N)=                 +1
N=            +33        M(N)=            +6        N-1=           +32        LOG2(N)=+.5044394119366⏨+  1 V(N)=                 +2
N=            +34        M(N)=            +6        N-1=           +33        LOG2(N)=+.5087462841257⏨+  1 V(N)=                 +2
N=            +35        M(N)=            +7        N-1=           +34        LOG2(N)=+.5129283016948⏨+  1 V(N)=                 +3
N=            +36        M(N)=            +6        N-1=           +35        LOG2(N)=+.5169925001450⏨+  1 V(N)=                 +2
N=            +37        M(N)=            +7        N-1=           +36        LOG2(N)=+.5209453365635⏨+  1 V(N)=                 +3
N=            +38        M(N)=            +7        N-1=           +37        LOG2(N)=+.5247927513446⏨+  1 V(N)=                 +3
N=            +39        M(N)=            +8        N-1=           +38        LOG2(N)=+.5285402218869⏨+  1 V(N)=                 +4
N=            +40        M(N)=            +6        N-1=           +39        LOG2(N)=+.5321928094891⏨+  1 V(N)=                 +2
N=            +41        M(N)=            +7        N-1=           +40        LOG2(N)=+.5357552004629⏨+  1 V(N)=                 +3
N=            +42        M(N)=            +7        N-1=           +41        LOG2(N)=+.5392317422782⏨+  1 V(N)=                 +3
N=            +43        M(N)=            +8        N-1=           +42        LOG2(N)=+.5426264754708⏨+  1 V(N)=                 +4
N=            +44        M(N)=            +7        N-1=           +43        LOG2(N)=+.5459431618641⏨+  1 V(N)=                 +3
N=            +45        M(N)=            +8        N-1=           +44        LOG2(N)=+.5491853096333⏨+  1 V(N)=                 +4
N=            +46        M(N)=            +9        N-1=           +45        LOG2(N)=+.5523561956063⏨+  1 V(N)=                 +4
N=            +47        M(N)=           +10        N-1=           +46        LOG2(N)=+.5554588851686⏨+  1 V(N)=                 +5
N=            +48        M(N)=            +7        N-1=           +47        LOG2(N)=+.5584962500725⏨+  1 V(N)=                 +2
N=            +49        M(N)=            +8        N-1=           +48        LOG2(N)=+.5614709844122⏨+  1 V(N)=                 +3
N=            +50        M(N)=            +8        N-1=           +49        LOG2(N)=+.5643856189781⏨+  1 V(N)=                 +3
N=            +51        M(N)=            +9        N-1=           +50        LOG2(N)=+.5672425341974⏨+  1 V(N)=                 +4
N=            +52        M(N)=            +8        N-1=           +51        LOG2(N)=+.5700439718144⏨+  1 V(N)=                 +3
N=            +53        M(N)=            +9        N-1=           +52        LOG2(N)=+.5727920454570⏨+  1 V(N)=                 +4
N=            +54        M(N)=            +9        N-1=           +53        LOG2(N)=+.5754887502168⏨+  1 V(N)=                 +4
N=            +55        M(N)=           +10        N-1=           +54        LOG2(N)=+.5781359713532⏨+  1 V(N)=                 +5
N=            +56        M(N)=            +8        N-1=           +55        LOG2(N)=+.5807354922064⏨+  1 V(N)=                 +3
```

```
N=        +57      M(N)=        +9     N-1=      +56     LOG2(N)=+.5832890014164⏨+  1 V(N)=        +4
N=        +58      M(N)=        +9     N-1=      +57     LOG2(N)=+.5857980995133⏨+  1 V(N)=        +4
N=        +59      M(N)=       +10     N-1=      +58     LOG2(N)=+.5882643049365⏨+  1 V(N)=        +5
N=        +60      M(N)=        +9     N-1=      +59     LOG2(N)=+.5906890595608⏨+  1 V(N)=        +4
N=        +61      M(N)=       +10     N-1=      +60     LOG2(N)=+.5930737337570⏨+  1 V(N)=        +5
N=        +62      M(N)=       +10     N-1=      +61     LOG2(N)=+.5954196310398⏨+  1 V(N)=        +5
N=        +63      M(N)=       +11     N-1=      +62     LOG2(N)=+.5977279923500⏨+  1 V(N)=        +6
N=        +64      M(N)=        +6     N-1=      +63     LOG2(N)=+.6000000000007⏨+  1 V(N)=        +1
N=        +65      M(N)=        +7     N-1=      +64     LOG2(N)=+.6022367813035⏨+  1 V(N)=        +2
N=        +66      M(N)=        +7     N-1=      +65     LOG2(N)=+.6044394119359⏨+  1 V(N)=        +2
N=        +67      M(N)=        +8     N-1=      +66     LOG2(N)=+.6066089190463⏨+  1 V(N)=        +3
N=        +68      M(N)=        +7     N-1=      +67     LOG2(N)=+.6087462841257⏨+  1 V(N)=        +2
N=        +69      M(N)=        +8     N-1=      +68     LOG2(N)=+.6108524456780⏨+  1 V(N)=        +3
N=        +70      M(N)=        +8     N-1=      +69     LOG2(N)=+.6129283016940⏨+  1 V(N)=        +3
N=        +71      M(N)=        +9     N-1=      +70     LOG2(N)=+.6149747119503⏨+  1 V(N)=        +4
N=        +72      M(N)=        +7     N-1=      +71     LOG2(N)=+.6169925001443⏨+  1 V(N)=        +2
N=        +73      M(N)=        +8     N-1=      +72     LOG2(N)=+.6189824558882⏨+  1 V(N)=        +3
N=        +74      M(N)=        +8     N-1=      +73     LOG2(N)=+.6209453365635⏨+  1 V(N)=        +3
N=        +75      M(N)=        +9     N-1=      +74     LOG2(N)=+.6228818690499⏨+  1 V(N)=        +4
N=        +76      M(N)=        +8     N-1=      +75     LOG2(N)=+.6247927513446⏨+  1 V(N)=        +3
N=        +77      M(N)=        +9     N-1=      +76     LOG2(N)=+.6266786540698⏨+  1 V(N)=        +4
N=        +78      M(N)=        +9     N-1=      +77     LOG2(N)=+.6285402218869⏨+  1 V(N)=        +4
N=        +79      M(N)=       +10     N-1=      +78     LOG2(N)=+.6303780748181⏨+  1 V(N)=        +5
N=        +80      M(N)=        +7     N-1=      +79     LOG2(N)=+.6321928094883⏨+  1 V(N)=        +2
N=        +81      M(N)=        +8     N-1=      +80     LOG2(N)=+.6339850002885⏨+  1 V(N)=        +3
N=        +82      M(N)=        +8     N-1=      +81     LOG2(N)=+.6357552004614⏨+  1 V(N)=        +3
N=        +83      M(N)=        +9     N-1=      +82     LOG2(N)=+.6375039431347⏨+  1 V(N)=        +4
N=        +84      M(N)=        +8     N-1=      +83     LOG2(N)=+.6392317422789⏨+  1 V(N)=        +3
N=        +85      M(N)=        +9     N-1=      +84     LOG2(N)=+.6409390936133⏨+  1 V(N)=        +4
N=        +86      M(N)=        +9     N-1=      +85     LOG2(N)=+.6426264754700⏨+  1 V(N)=        +4
N=        +87      M(N)=       +10     N-1=      +86     LOG2(N)=+.6442943495844⏨+  1 V(N)=        +5
N=        +88      M(N)=        +8     N-1=      +87     LOG2(N)=+.6459431618641⏨+  1 V(N)=        +3
N=        +89      M(N)=        +9     N-1=      +88     LOG2(N)=+.6475733430969⏨+  1 V(N)=        +4
N=        +90      M(N)=        +9     N-1=      +89     LOG2(N)=+.6491853096333⏨+  1 V(N)=        +4
N=        +91      M(N)=       +11     N-1=      +90     LOG2(N)=+.6507794640202⏨+  1 V(N)=        +5
N=        +92      M(N)=       +10     N-1=      +91     LOG2(N)=+.6523561956063⏨+  1 V(N)=        +4
N=        +93      M(N)=       +11     N-1=      +92     LOG2(N)=+.6539158811109⏨+  1 V(N)=        +5
N=        +94      M(N)=       +11     N-1=      +93     LOG2(N)=+.6554588851679⏨+  1 V(N)=        +5
N=        +95      M(N)=       +12     N-1=      +94     LOG2(N)=+.6569855608330⏨+  1 V(N)=        +6
N=        +96      M(N)=        +8     N-1=      +95     LOG2(N)=+.6584962500725⏨+  1 V(N)=        +2
N=        +97      M(N)=        +9     N-1=      +96     LOG2(N)=+.6599912842190⏨+  1 V(N)=        +3
N=        +98      M(N)=        +9     N-1=      +97     LOG2(N)=+.6614709844114⏨+  1 V(N)=        +3
N=        +99      M(N)=       +10     N-1=      +98     LOG2(N)=+.6629356620077⏨+  1 V(N)=        +4
N=       +100      M(N)=        +9     N-1=      +99     LOG2(N)=+.6643856189774⏨+  1 V(N)=        +3

ER  998     16        +100
```



n ----->
h g mulder.
ε(n) ----->
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
0
1
2
3
4
5
6
7
8
9
10
11
12

3. Correctness proofs

A program (or algorithm) ought to be accompanied by a proof of correctness.

Advantages. The discipline of proof has the advantages:

1. Provides a systematic search for errors.
2. Gives sufficient reasons why the program must be correct.
3. May lead to ways by which the program can be improved.
4. Makes explicit the assumptions on which correctness rests.

Hence an attempt to satisfy yourself as to the correctness of a program is the first and most basic part of the analysis of any computer algorithm.

Feasibility. What is a "correctness proof?" It is not reading a program closely and then announcing that it works. Nor is it using the standard debugging technique of testing "representative" input and checking the resultant output. As Dijkstra says (in Burton and Randell 1970), "Testing shows the presence, not the absence of bugs". By correctness proof we will mean a rigorous mathematical proof which verifies that a program which appears to be intuitively adequate is in fact correct. It is hardly easier to prove the correctness of programs than to establish proofs of theorems.

However, a correctness proof expressed completely formally in, say, predicate calculus notation will not be our goal. Indeed, we will emphasize informal, but rigorous, demonstrations given as standard mathematical arguments in prose form. The detail and precision used will depend in part on the particular program to be proved, on the programming language used, and on the audience to whom the proof is directed. Preliminary work has been done on automated verification of correct programs (let the computer do it!), but current techniques fall short of producing such proofs automatically in all (or even most) cases.

Saddle point program proof. To provide an example correctness proof, we consider the problem of finding a saddle point of an $m \times n$ matrix A. Element A[i,j] is a saddle point if

(3.1) $$A[i,j] = \min_{1 \leq k \leq n} A[i,k] = \max_{1 \leq k \leq m} A[k,j].$$

Figure 3.1. specifies an ALGOL program to output a saddle point of A if there is at least one, or a zero if there is no saddle point. We now show that this program works properly for all matrices A. Our method is to label each key step of the program with an assertion about the current state of affairs at the time the computation reaches that step. These "key inductive assertions" are given as comments in the program text.

Assertion A1 always hold by virtue of the initialization of the row index i to zero and the <u>while</u> clause controlling the <u>for</u> statement on i.

Similarly, assertion A2 is always true when control reaches that point.

Assertion A3 follows from the fact that the <u>for</u> statement preceding it sets SP to <u>false</u> iff

(3.2) $$A[i,k] < A[i,j] \text{ for some } 1 \leq k \leq n.$$

Clearly, equation (3.2) holds iff

$$A[i,j] \neq \min_{1 \leq k \leq n} A[i,k],$$

that is, element A[i,j] is not a saddle point of A because there is a smaller element in its row.

Assertion A4 has two parts which follow from the two cases:

(1) SP is <u>true</u> after assertion A3.

Then the <u>for</u> statement preceding assertion A4 sets SP to <u>false</u> iff

(3.3) $$A[k,j] > A[i,j] \text{ for some } 1 \leq k \leq m.$$

Clearly, equation (3.3) holds iff

$$A[i,j] \neq \max_{1 \leq k \leq m} A[k,j],$$

that is, element A[i,j] is not a saddle point of A because there is a larger element in its column.

(2) SP is _false_ after assertion A3.
Then A[i,j] is not a saddle point of A and the statements between assertions A3 and A4 have no effect on SP.

Assertion A5 is obvious from the block structure of the program.

Finally, assertion A6 holds because both _for_ statements on i and j terminate only when either: (1) SP = _true_, and hence A[I,J] is a saddle point by virtue of assertions A3 and A4 along with definition (3.1); or (2) SP = _false_, with I = m and J = n, so that all possible values of I and J have been tried without finding a saddle point.

The author readily confesses that constructing the above proof demonstrated to him the need for the variables I and J; in other words, my first _incorrect_ version of the program printed out A[i,j] as a saddle point. Note it is also not possible to print A[i-1,j-1] if SP is _true_, because ALGOL 60 leaves the controlled variables i and j "undefined" after exit from the _for_ clauses.

Termination. There remains one important point: We never showed that the program terminates! Indeed, proofs of termination are usually handled separately from the verification of correct results. Termination of the saddle point program is easily established because the program does not use transfers of control by means of _goto_ statements. We need only remark that each of the four _for_ statements starts with controlled variable equal to 0, incrementing it by 1 for a _finite_ number of times since it cannot exceed max{m,n}, by virtue of the _while_ clauses. Thus control always reaches the print procedure call after executing each line of code at most

$$(m+1)(n+1)(\max\{m,n\}+1)$$

times.

Key inductive assertions. This method of algorithm-proving in terms of key inductive assertions is essentially due to Floyd (1967) and Naur (1966), who called them "general snapshots". In general, the method places assertions concerning the progress of the computation between lines of code. Next, it is demonstrated that each assertion is true every time control reaches that assertion, under the assumption that the previously encountered assertions hold. Using induction on the number of lines of code, it follows (Knuth 1968) that this yields a valid proof. Termination of the program is then shown separately.

Square root example. That a program terminates can be more difficult to show than that the correct result is achieved. This is the case for program ROOT in Figure 3.2 which computes the square root of a real positive argument x, by the Newton-Raphson method. In the Newton-Raphson method (see Hamming 1971, section 2.8: "Newton's method (another method to avoid)"), an initial guess y_0 for $\sqrt{x}$ is iteratively improved by

(3.4) $$y_{i+1} = (y_i+x/y_i)/2 \qquad \text{for } i = 0,1,2,\ldots .$$

We take $y_0 = 1$ and stop the guessing when

(3.5) $$y_i < [(1+\varepsilon)/(1-\varepsilon)]^{1/2}\sqrt{x} \qquad \text{for } \varepsilon = 10^{-6}.$$

From equations (3.4) and (3.5) it is easy to see that we stop guessing when

(3.6) $$\frac{|y_{i+1}-y_i|}{y_{i+1}} = \frac{y_i^2 - x}{y_i^2 + x} < \varepsilon = 10^{-6}.$$

In our program (Figure 3.2), z and y correspond to y_{i+1} and y_i, respectively. Thus the while clause correctly stops the guessing when equation (3.6) is satisfied.

But is (3.6) ever satisfied? To show that this program does indeed terminate, we note that (except for y_0 and assuming exact arithmetic) each guess y_{i+1} is $\geq \sqrt{x}$ and is $< y_i$. (x=1 is a trivial special case.) Therefore, we have a bounded monotone sequence of guesses which, of course, must eventually produce successive guesses which differ in relative error by less then any $\varepsilon > 0$.

A mathematician would now be satisfied that program ROOT terminates.

A numerical analyst or programmer should continue to worry about the precise number of iterations needed. We could set an upper limit to the number of iterations, so that ROOT stops if it does not converge rapidly enough. In fact, it is good programming practise to set an upper limit to the number of iterations used in any iterative algorithm. But we can do better; for assuming exact arithmetic, the maximum number of times that procedure ROOT will execute the statement "y:= z;" is

$$5 + |\log_2 \sqrt{x}| \qquad (3.7)$$

Formula (3.7) is based upon two facts: (i) The first guesses y_i are approximately equal to $x/2^i$; (ii) When ROOT gets close to $\sqrt{x}$, it tends on each step almost to double the number of decimal places that are accurate.

Before the author supplied the above correctness proof for procedure ROOT, he felt that he understood the Newton-Raphson method. But after struggling to prove correctness, especially termination, he had a greatly increased understanding of programming the Newton-Raphson method for square roots. We often fail to realize how little we know about an algorithm until we attempt to prove it works!

Because some programmers believe correctness proofs to be impossible, too difficult, trivial, and/or not worth the effort, let us seek to clarify what is meant by "increased understanding" and at the same time discuss complaints made against proving correctness. (See Smith 1972.)

Levels of understanding. E. de Bono (1971) distinguishes between five levels of understanding in practical thinking:

1. Simple description. (Just describe what ROOT seems to do.)
   "It computes $\sqrt{x}$ for positive x".

2. Porridge words. (Use vague words like approximation, iteratively, accuracy.)
   "It approximates $\sqrt{x}$, for positive x, by iteratively improving guesses until the desired accuracy is attained".

3. Give it a name. (Identify and name the process.)
"It approximates $\sqrt{x}$, for $x > 0$, by the Newton-Raphson method."

4. The way it works. (Describe process in broad terms.)
"It approximates $\sqrt{x}$, for $x > 0$, by y in the following steps: Guess a value of 1 for y; find the average of y and the quotient x/y; this is a better guess; repeat until the relative error between successive guesses is less than 10^{-6}."

5. Full details. (Full details of what is happening.)
"Refer to the ALGOL 60 program text (Figure 3.2) and to the correctness proof for it given above."

It is not uncommon for programmers to use level 1 (Simple description) understanding to provide the basis for action and for decision in their computer work. For example, if they need a logarithm program, then a simple description such as

"Library procedure ln(x) computes the natural logarithm of real positive x."

can satisfy them.

Level 2 (Porridge words) understanding is more specific than level 1 because it is based upon useable explanations instead of just a simple description. For example, although

"Library procedure CURVEFIT outputs the parameter values of that particular curve of best fit at the input points."

is a vague explanation of CURVEFIT ("curve" and "best fit" are clearly Porridge words), it offers a useable explanation. The author knows computer users who often used programs which

"perform a significance test for independence in a two-way contigency table",

without worrying about the Porridge word "significance test", which includes the cases: Chi-square (χ^2), χ^2 with Yate's correction for continuity, and an exact test (such as Fisher's for 2 × 2 tables) based on a multinomial distribution. Differences in significance can arise depending upon which test is used.

Level 3 (Give it a name) understanding is a very big step forward from levels 1 and 2, because as soon as an algorithm is named (Newton-Raphson method, Least squares straight line, Binary sort), you can look it up or otherwise identify it to your satisfaction. Even a cautions numerical analyst will frequently settle for level 3 understanding since a name like "Fast Fourier transform algorithm" can convey so much information to him.

Level 4 (The way it works) understanding is based upon a general description of the way the program works. Consider the following level 4 description of a binary search in a sorted list:

> "Comparison is made with the element at the center of the list: whichever way the comparison goes, the item being searched for is now known to lie in some list which is one half as long as the original list. Comparison is now made with the element at the center of this list, and the process continues. At every stage, it is possible to identify a list half as long as one previously identified, as the one containing the item. Hence at most n + 1 tests are necessary to find the item if there are 2^n elements in the list."

Textbook authors will specify computer algorithms in this way to avoid painful details and yet explain the way it works. They strive to stop at that fullness of detail which makes it unnecessary for anyone to ask why or how.

Level 5 (Full details) understanding is the most detailed although it is obviously impossible to give complete details in any absolute sense. For example, "complete details" for an ALGOL 60 program would have to include the correctness of its compiler and the hardware of the computer used! But as de Bono says, "if you go beyond the practical detail to further detail the situation may become unfamiliar again".

In this syllabus full details of an algorithm should always include an annotated program text accompanied by an correctness proof.

Arguments against correctness proofs. Viewed from the above five levels of understanding, whether or not you require program correctness to be shown depends on what level of detail you desire. As an illustration of this, there was a recent essay contest (McCracken 1971) on the topic "Would you trust the lives of your children to a highly complicated computer system that cannot be checked out?" The computer system is the Safeguard Anti Ballistic Missile system (ABM) in America. Essays in this contest argue at all levels from 1 to 5. A level 1 point is that the mere description of an ABM system should prevent its ever being used. (Like the "Doomsday machine" in Dr. Strangelove.) A level 5 point is that the ABM cannot be "fully" checked out without testing under actual operating conditions. One programmer (Glass 1971) with 15 years experience in the aerospace industry claimed in his essay, "And I have never yet written a checked out program". Arguments against level 5 understanding are based upon expediency; that is, "Practical man has to be right as soon as possible because he has things to do (de Bono 1971)". Yet the practical explanation of a program which is more useful under certain circumstances is not necessarily better than a deeper explanation. Surely one has increased confidence whenever a program is accompanied by a correctness proof, even though proofs of correctness share problems with more usual mathematical proofs (e.g., communication of the proof to the reader, level of detail, finding the proof).

Proof techniques. That concludes our definitions and justifications for program correctness proofs. We next examine some useful techniques, however imprecisely defined, for constructing convincing program proofs:

Variable change table. A cross reference table of the identifiers declared, changed, and/or used in an ALGOL 60 block is useful in:

(i) showing variables are unchanged between two points.
(ii) detecting undeclared or multiply declared identifiers.
(iii) finding variables used before they are changed.
(iv) locating control statements, variable usages, etc.

To illustrate these uses for a variable change table, we consider the meaningless "nonsense program" in Figure 3.5. Corresponding to the one procedure (SORT) and the outer block (exclusive of SORT), there are two variables change tables as given in Figures 3.3 and 3.4. These tables are clear from inspection of the program and could be produced almost entirely automatically.

Information provided by the table in Figure 3.3 follows. Firstly, we see that the label "loop" is defined at line number 20 and there exists two jumps to loop, one at line 13 and one at line 24. Whenever goto's are used, the labels should be included in a variable change table. If there is no line number in the "declared" column, then obviously the program is transferring control to a nonexistent label. Similarly, if there are no line numbers in the "used" column, then the program contains a redundant label.

There are precisely two procedure statements and they both invoke procedure SORT whose declaration appears in lines 4 to 8 inclusive. From the procedure names which have no line numbers in the "declared" column, you can determine which library procedures the program requires.

Array D is referenced in lines 20 and 21 but was never declared. Of course, ALGOL demands that all variables be declared so that the compiler should catch this error. Languages such as FORTRAN and PL/1, however, permit declaration by default and then knowledge of a missing explicit declaration may be useful. Multiple declarations are easily detected by looking for two or more line numbers in the "declared" column.

The simple variable alpha, although properly delcared and assigned a value, is never used in an arithmetic expression. On the other hand, the array element B[3] is undefined when it is used at line 15. In general, when a variable is used at line number U, changed at line number C, and U < C, then beware for an undefined (not undeclared) variable.

Note that array A is both used and changed in line 14 where it appears as a parameter to procedure SORT. You must trace procedure parameters to check if their values are changed by the activated procedure. Naturally, when a formal parameter of the procedure is a value parameter, then the corresponding parameter in the procedure call can be used but not changed. A call by name parameter must ultimately appear on the left hand side of the assignment operator (:=) in order to be changed.

Besides locating errors a variables change table permits you to make assertions like:

"Except for Y and D[ ], nothing else is changed in lines 21 through 25."

"Formal parameter n in procedure SORT is used only in an arithmetic expression after an <u>until</u> (line 6)."

Such assertions that a variable value is changed only in a specified way, are useful in program correctness proofs. A variable change table is a precise verification of such assertions.

<u>Debugging syntax errors</u>. Errors in grammar should be uncovered before a correctness proof is attempted. Much confusion and frustration is caused by spelling mistakes in mathematics texts and, technically, any formula is just not correct when it contains a grammatical error. Of course, ALGOL 60 compilers often will not translate a program until its syntax agrees with the Revised Report. For example, the code

```
y:= 6. × w;
```

is illegal because a zero is missing after the decimal point.

Because FORTRAN and PL/1 have default options and their syntax was ill-defined until recently, debugging syntax errors in these languages is more difficult. It even happens that a legal FORTRAN statement such as

```
FORMAT(6H)=(A+B)
```

often stops compilers. Explicit and mandatory declarations in ALGOL avoid most spelling errors in identifiers from going unnoticed.

Certainly the clerical exercise of detecting syntax errors is "work unfit for a Christian" and should be automated. Both compile- and runtime diagnostics are desirable. In the program

```
begin array A[1:5]; print (A[read]) end
```

only a run-time check will detect an illegal subscript. Warning messages are also desirable. It can be helpful to be warned that an identifier was declared but never used in an ALGOL program.

A common error in ALGOL programming is to omit a semicolon after an end which thereby turns the text following the end into a comment. A warning can be provided when this comment contains a delimiter (:=, +, go to, etc.). Automated error detection can even be coupled with automatic correction schemes to convert the illegal ALGOL text into a legal program.

Testing for semantical and logical errors. It is not reasonable to "prove" correctness details before we have good reasons to believe that the program works. The classical testing techniques are not only easier to carry out, they also provide timing data end extensive empirical evidence for algorithmic analyses. Consideration of the cost for certification of a program often leads to the experimental (testing) rather than the analytic (correctness proof) approach. However, no matter how many experiments are conducted, a program can never be shown to be correct by testing alone. A tested program may be considered "empirically O.K." and testing can establish milestones for the measurement of programming progress, but only a correctness proof can precisely and sufficiently demonstrate that the program achieves the desired results.

We want to distinguish between testing for semantic errors and for logic errors. An example of a semantics error in ALGOL is

```
begin integer a,b;
      procedure one(x); two(x)
      procedure two(x); x:= 4;
      a:= b:= 1; one (a+b)
end
```

where the meaning of "a+b:= 4" is undefined. Note that the above program is syntactically valid ALGOL; it is semantically invalid. Explanations of the semantics of ALGOL 60 are considered too well-known to be stated explicitly in a correctness proof. Nevertheless, details of rarely used (e.g., Jensen's device) ALGOL features should be spelled out when they are incorporated within a program. It is surprising that many ALGOL programmers fail to take into account that the controlled variable in a for loop is undefined upon exit from the loop. They rely upon their particular hardware representation for an exit value, rather than adhere to the semantics in the reference language.

Logical errors are mistakes in program construction where the text has both valid syntax and semantics, but it does not have the intended effect. When "i:= i+1" is written erroneously for "i:= i-1", when an array is sorted erroneously in ascending rather than descending order, when the relational "$\geq$" is used erroneously instead of "<" --- these are logical errors in the program.

Testing for semantical and logical errors involves the classical techniques of dumps, snapshots, traces, etc. The idea is to run the program with a test case for input and to verify the output or else trace the route to failure. The testing is best done in an incremental fashion so that only one procedure or block is being tested at one time. A good strategy is to replace untested program sections with simplified working sections, and then substitute the tested sections for their simplified versions in steps.

Test input may consist of actual data or constructed data. The data may be stereotyped or a statistically generated random sample. A program to generate the test data may help. For example, to test a matrix inversion procedure the ill-conditioned Hilbert matrix

$$A[i,j] = 1/(i+j-1) \qquad \text{for } 1 \leq i, j \leq n$$

may be input. These matrices probably represent an extreme case relative to what the inversion procedure will usually handle. But extreme and exceptional conditions should be tested when possible. Often library procedures are undocumented with respect to division by zero, square root of negative argument, subscript out of range, overflow, and so forth. You may have to test these situations to learn what to expect.

Solutions for test data may be obtained from: (1) books, journals, etc.; (2) hand calculation; (3) a program which has been proved correct and which solves the problem in another way. It pays to always perform an order of magnitude check on final results. When the sample variance of a set of observations seems large, for example, a mispunched data value should be suspect.

Redundant output. Every program ought to produce lists of intermediate results for examination. This output will allow you to measure program progress should your time limit be exceeded or a machine failure occur or a programming error prevent successful completion. Such output should also be planned to provide interesting statistics of long successful runs. For example, when the program is sorting a large file, there are numerous questions which might be answered upon termination: How unsorted was the original file? (Perhaps measured by the total number of interchanges required.) Which step in the sort algorithm consumed the most time? (An empirical profile - see section 4 - would answer this.) Was the sort input/output bound?

When the output of a program is basically negative in value, there are still possibilities for useful statistics and/or further work. For example, if a number theory calculation fails to show that $n = 2^p-1$ is a Mersenne prime for some large prime p, then the factors of n should be output both for human verification and for possible interesting properties.

The need for well-labeled output is obvious, for when you cannot identify an answer you must examine perhaps the entire program to learn its meaning. Output should include every input value that was read in; otherwise, you may not know which problem was solved. When the program has options (like more than one sorting or matrix inversion procedure), the alternative actually used should be identified.

Indeed, an alternate algorithm to double-check results may be advantageous. In any case, you should compare the output with a commonsense estimate of the answer. Check the output. Look at it. Be shocked by unlikely results. Does the sum of some computed probabilities equal one? Does the product of A and A^{-1} (as computed) equal I? If not, how much do they differ? Perform a difference table analysis to detect errors in a table of computed values. Although a final printout or postmortem dump of global variables may be redundant output, many bugs and other useful information have been gleaned from such output by observant programmers.

Enumeration proof. Why doesn't exhaustive testing prove correctness? After all, if your n! procedure accepts only the ten values $1 \leq n \leq 10$, then by successively calling this procedure with each of these ten values and then checking the answers, have you not proved correctness of the procedure by enumeration? No, because there are programs which work on "day 1" but fail on "day 2". Figure 3.6 displays three versions of an n factorial procedure, each one of which could work properly for the first ten calls (day 1) and then fail miserably (day 2). Version 1 depends on a loop counter, version 2 tests a random number, and version 3 relies on special initialization. Whenever a new version of the library (compiler, sqrt procedure, overflow handler, etc.) is installed, some users usually complain because their correct program no longer works. There is an authentic case where physicists had been "successfully" running their lengthy, complicated ALGOL programs for a year. Then one day run-time subscript checking was introduced into the system. Their programs immediately terminated with "subscript out of bounds" error messages. They demanded that subscript checking be made optional and then chose to supress it!

The "It-works-if-a-test-case-does" school of programming also ignores compiler and language specifications. To determine the exit value of a controlled variable in ALGOL, they run a test case instead of reading the Revised Report. Furthermore, the "Change-it-until-it-works" school of programming ignores compiler errors, documentation ambiguities, and so on. When their program doesn't work, they make changes until "it does".

There exists one situation where testing can indeed prove correctness. That is the case where the program output can be checked by hand or by another correct program. If the output yields the correct answer, then it can be asserted that the program worked for that specific case. For example, a matrix inversion procedure which computes B, the supposed inverse to A, can be verified correct for a specific input A by testing that AB = I. But if it is run again, you must test the output each time.

Using the problem domain. When developing a program correctness proof, it should be expected that relations or properties from the problem domain will be used. For example, in the exponentiation programs of Section 2 we needed the law of exponents

$$y^a \times y^b = y^{a+b},$$

which was assumed as obvious. A relation such as

$$(x+y) \bmod n = x \bmod n + y \bmod n,$$

may have to be proven explicitly for some readers while others will accept it as an obvious fact from modular arithmetic.

Besides relations peculiar to the problem domain, the proof may follow and depend on general principles unrelated to the problem domain or to programming. Much of the discrete mathematics found in (Knuth 1968) can be applied in correctness proofs as well as in frequency and storage analyses. Recall the correctness proof for square root by Newton-Raphson iteration was based upon a theorem in calculus on bounded monotone series. Your ability to exploit mathematics (combinatorial, statistics, probability, logic, etc.) will determine the ease with which a correctness proof is established. As usual, the more math you know, the better off you usually are.

Well-defined generalized input. There is a computing cliché: "Garbage in, garbage out". And it is true that numerical mistakes can arise from bad input. Only by printing out the input values read in will you be sure to eliminate keypunching and format errors. In addition, the program should check that certain conditions are satisfied. For example, while reading in a probability distribution p_i $(i=1,2,\ldots,n)$ the program should check $n > 0$ and each $0 \leq p_i \leq 1$, while printing the values input. The sum $\sum p_i$ should also be verified equal to 1.

To avoid clerical errors, free format helps. Instead of demanding that n be punched right-justified in columns 6 to 10, it is helpful to allow n to appear anywhere (including after blank cards) as the first data item. Even better would be to use unordered input:

```
n = 3;  p[3]:= 0.5;
p[1]:= 0.25;
p[2]:= 0.25;
```

The advantages are that a shuffled data deck is not an error and that a specific input value such as p[3] can be changed by simply adding its new value to the end of the input deck. Also, such labeled input is self-documenting. Additional comments can be placed between, say, "¢" signs:

```
n = 3 ¢Number of probabilities¢;  p[3]:= 0.5;
```

Backus-Naur notation can be used to define what input syntax is legal in order to avoid misunderstandings. Questions like "Are leading zeroes permitted (0.61)?", "Do plus signs have to be punched?", and "Is there a power-of-ten notation?", can be easily answered by referring to the well-defined input specifications.

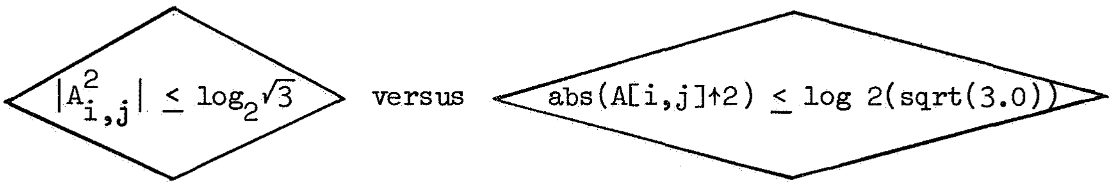
A library program with sizable input should be designed around an "input language". For instance, the self-explanatory input to a linear programming procedure might look like:

```
begin comment Problem 1 input;
      number of equations = 4;
      number of variables = 3;
      initial guess x = (1,1,0);
      no objective function;
      3x[1] + 4x[2] -  x[3] ≤ 1;
              5x[2] - 7x[3] = 3;
       x[1] +  x[2]         ≥ 0;
                     x[3] ≥ 0;
end
```

The difficulty with input languages is the task of programming compiler-like procedures to translate them. Nevertheless, some of the most popular library procedures are and should be based upon a well-defined, generalized input language for the user.

Flowcharts. Understanding both a program and its correctness proof can be facilitated by introducing suitable notation and drawing a figure (flowchart). Flowcharts are easy to produce, easy to recognize, and easy to remember. They are a pregnant notation for expressing more at a glance than the program text. Do not give too much detail in your flowcharts; otherwise, you might as well simply read the program text directly. See the sample flowchart in Figure 3.7.

Flowchart notation for for loops should consists of a single box. Figure 3.8 gives possible notations for the three kinds of for statements. Flowchart standards (diamond-shaped box for decisions, square box for computations, etc.) exist and should be adhered to if you seek a wider audience for your flowcharts. The notation used within the flowchart boxes should be two-dimensional (like the ALGOL publication language) rather than linear (like the ALGOL reference language). For example,


$|A^2_{i,j}| \le \log_2\sqrt{3}$
versus
abs(A[i,j]↑2) ≤ log 2(sqrt(3.0))

Type analysis. In science there is a simple but powerful technique named "dimensional analysis" which consists of substituting the units of each variable into a formula and then canceling to check for consistency of the units. In ALGOL there are chances to perform a somewhat similar "type analysis". For example, when calling the procedure sin(x), you should always check that x has the right type (real or integer) and the right units (radians or degrees). Some compilers do not check type compatibilities, so it will be the programmer's responsibility to call ln(x) with a real or integer or either, depending upon local library conventions. Type compatibility of actual and formal parameters of a procedure should be made anyways to avoid an embarassing error message after the program is supposedly correct.

Mixed modes are useful but must be checked for the desired effect. Most language manuals provide a table of mode possibilities for the left- and right-hand variables in substitution statements. Thus, whether

```
real y; Boolean b;
y:= b;
```

is allowed and what it means is specified. Substituting a real into an integer variable usually has the result of applying the greatest integer function to the real. Allowable input data types should also be carefully defined and checked.

By using or assuming all numeric variables to be of type integer, the correctness proof need not consider round-off error.

Lastly, a word about array references being within bounds. Sometimes checking of subscripts is an optional feature at run-time. This is analogous to a man who always keeps a fire-extinguisher in his car whenever it is not being used, and takes the fire-extinguisher out whenever he goes on a trip. That is, a program in production needs subscript error checking because that is when money or life depends on it. It is not disastrous, on the other hand, when a subscript error occurs while testing your program.

Documentation. Communicating the correctness proof is an art because it requires ingenuity and creativity. Reading a convincing proof should be easier than reading the program. The need for documentation of the correctness proof is clear, for "If it isn't written down, it doesn't exist". Consequently, the user will not know the program is well-tested if it isn't well documented.

In the documentation there should be an informal statement of the problem in a natural language followed by explicit identification of all assumptions including accuracy, round-off error, input ranges, overflow, and so forth. Crossreferencing in both directions should exist between documentation and program text. Hence the program must be highly readable with meaningful line identification numbers and indentations (don't be afraid to insert blank line between, for example, procedures and blocks). Furthermore, the key inductive assertions of the correctness proof should appear in the program as comments. Where the documentation is lacking or incomplete with respect to these points, the program itself must be considered to be defective.

A documented correctness proof may be checked for error-freeness by the various program users, and it serves to set forth the best method for program certification known.

Program equivalence. One approach to showing program Q is correct, is to show that Q is equivalent to a correct program P. Thus you reduce the problem to one previously solved.

For example, the two traversal programs in Figures 5.5 and 5.6 both visit a tree in postorder. The recursive version (Figure 5.5) is easy to write, elegant, and trivial to prove. The iterative version is portable to languages which have no recursion. To prove the iterative version (Q) one can simply show it is equivalent to the correct recursive version (P). Equivalence is shown by proving that any tree input to Q produces the same output as P does, and vice-versa. Output here means the same sequence of calls to procedure VISIT;

Induction. A general method applicable to proving the validity of any algorithm uses mathematical induction. After labeling each of the arrows in the flowchart of the algorithm with an assertion about the current state of affairs at the time the computation reaches that arrow, induction is used to show that all these assertions are true during any execution of the algorithm. Consider the example of computing the sum:

$$\text{SUM} = \sum_{i=1}^{I} x_i . \tag{3.8}$$

Figure 3.9 gives the flowchart along with the "key inductive assertions". It is easy to prove that each key inductive assertion leading into a box implies each assertion leading out, for this particular example. By induction if follows that (3.8) holds upon exit from this algorithm.

Thus this proof technique consists mostly of inventing the key assertions to put in the flowchart. In loops we require an assertion describing the processing accomplished by the i-th execution of the loop. In recursive calls of a procedure we require an assertion describing the result of invoking the procedure for the i-th time. The well-known change problem (how many ways can you change one guilder?) can be stated as a recursive algorithm and then proved correct by recursive induction (see Polya 1957).

Case analysis. The old strategy "Divide and conquer" can be employed in correctness proofs. We have already mentioned that the checkout of assemblages of program components is best accomplished in an incremental fashion. Further, input possibilities can be sectioned so that, for example, the cases $n < 0$, $n = 0$, and $n > 0$ are treated separately.

When labeling the flowchart with inductive assertions, the key steps can be chosen to be procedures, blocks, and starting or ending points of loops. This breaks the program up into managable pieces. Complex decision choices (if-then-else structures) can also be broken up into cases. The conditions with their resulting actions to be taken should be displayed in a table such as:

Income-tax calculation

Condition	Action
line 10 < line 11	Pay refund
line 10 = line 11	Close account
line 10 > line 11	Bill taxpayer

This table is used as an intermediate representation or notation. Hence, first you need to show that this table follows from the code and, secondly, you show that the table is implemented properly. One complicated step is thereby replaced by two simpler steps. This is merely the technique mathematicians use when they develop the proof of a main theorem through a series of lemmas.

Nevertheless, before decomposing the problem into cases and working at details, you should understand the program as a whole so that you don't lose yourself in details. A common fault of programmers is that they rush into constructing their program before they have thoroughly understood the problem and have devised a general plan for its solution.

It is obvious that a program which uses library procedure or well-known algorithms is partly proved correct since these parts have already been certified or otherwise shown correct.

Figure 3.1. Program to print a saddle point (i.e. an element which is the smallest value in its row and the largest value in its column) of matrix A if one exists; otherwise, it prints zero. Key steps are labeled with assertions which prove the validity of the program.

```
Boolean SP; integer i, j, k, I, J; array A[1:m,1:n];
SP:= false; i:= 0;
for i:= i+1 while ¬SP ∧ i ≤ m do
begin comment A1: 1 ≤ i ≤ m and SP = false;
      I:= i; j:= 0;
      for j:= j+1 while ¬SP ∧ j ≤ n do
      begin comment A2: 1 ≤ i ≤ m and 1 ≤ j ≤ n and SP = false;
            SP:= true; k:= 0; J:= j;
            for k:= k+1 while SP ∧ k ≤ n do
            if A[i,k] < A[i,j] then SP:= false;
            comment A3: SP = true iff $A[I,J] = \min_{1 \leq k \leq n} A[I,k]$;
            k:= 0;
            for k:= k+1 while SP ∧ k ≤ m do
            if A[k,j] > A[i,j] then SP:= false;
            comment A4: SP = true implies $A[I,J] = \max_{1 \leq k \leq m} A[k,J]$;

                        SP = false implies A[I,J] ≠ saddle point of A;
      end;
comment A5: Same as A4;
end;
comment A6: SP = true implies $A[I,J] = \min_{1 \leq k \leq n} A[I,k] = \max_{1 \leq k \leq m} A[k,J]$;

            SP = false implies A has no saddle point;
print (if SP then A[I,J] else 0);
```

Figure 3.2. Procedure to find an approximate square root of a positive real argument x by Newton's method.

```
real procedure ROOT(x); value x; real x;
begin real y, z, error;
      if x > 0 then begin
                    error:= 0.000001;
                    for z:= 1, (z+x/z)/2 while abs(y-z)/z ≥ error do y:= z;
                    ROOT:= (y+x/y)/2
                    end
      else if x = 0 then ROOT:= 0
           else begin
                printtext ("Procedure ROOT entered with negative argument
                           equal to");
           print(x); ROOT:= 0
           end
end
```

comment Test of ROOT with x = 10^{10} on the X-8 computer. The successive variable values are:

```
          z       x=10000000000.0     error=0.000001           y

          1                                                    ?
 5000000000.5                                                  1
 2500000001.3                                        5000000000.5
 1250000002.6                                                ETC.
  625000005.3
  312500010.7
  156250021.3
   78125042.7
   39062585.3
   19531420.7
    9765966.3
    4883495.1
```

```
          z        x=10000000000.0      error=0.000001           y

      2442771.4
      1223432.6
       615803.1
       316021.1
       173832.3
       101062.62
       100005.586
       100000.0001560
       100000.0000000
```

Note: $5 + \mathrm{abs}(\log_2(10^5)) = 21.61$;

Figure 3.3. Variables change table for outer block in the program of Figure 3.5.

Identifier	Declared at line	Changed at line	Used at line
i	2	12,20	15,16,17,20
alpha	10	12	--
Y	3	12,23	13,22
B[3] (array)	9	18	15
A[ ] (array)	9	14,17	14
loop (label)	20	--	13,24
D[ ] (array)	--	20,21	21
SORT (proc.)	4-8	--	14,21

Figure 3.4. Variables change table for procedure SORT in the program of Figure 3.5.

Identifier	Declared at line	Changed at line	Used at line
SORT (proc.)	4	--	--
C[ ] (array)	4	4,7	7
n	4	4	6
i	5	6	7
sqrt (proc.)	--	--	7

Figure 3.5. Nonsense program to illustrate possibilities in a variables change table. See Figure 3.3 and 3.4.

```
[ 1] begin comment Nonsense program;
[ 2]    integer i;
[ 3]    real Y;
[ 4]    procedure SORT (C,n); integer array C[1:n]; integer n;
[ 5]    begin integer i;
[ 6]          for i:= 1 step 1 until n-1 do
[ 7]          if C[i] < C[i+1] then C[i]:= sqrt (C[i+1])
[ 8]    end;
[ 9]    integer array B[1:10], A[1:50];
[10]    Boolean     alpha;
[11]
[12]    Y:= read; alpha:= true; i:= 0;
[13]    if Y ≤ 0 then go to loop;
[14]    SORT (A,25);
[15]    if B[3] = i then begin
[16]                     for i:= 1 step 1 until 10 do
[17]                     A[i]:= i;
[18]                     B[3]:= 0
[19]                     end;
[20]    loop: for i:= 1 step 1 until 10 do D[i]:= 0;
[21]    SORT (D,10);
[22]    if Y = 6.3 then begin
[23]                    Y:= 0;
[24]                    go to loop
[25]                    end;
[26] end
```

Figure 3.6. Three versions of an n factorial procedure (nfac) which work correctly only sometimes.

```
comment Version 1 works (assume i is initially zero) for the first ten
        calls only;
i:= i+1;
nfac:= if i ≤ 10 then n! else -1;

comment Version 2 works if and only if a random number is in the interval
        [0,½];
nfac:= if random ≤ 0.5 then n! else -1;

comment Version 3 works if and only if i is initially zero or one while
        n > 0, or else i is initially ≥ 0 and n=0;
answer:= 1;
for i:= i+1 while i ≤ n do answer:= answer × i;
nfac:= answer;
```

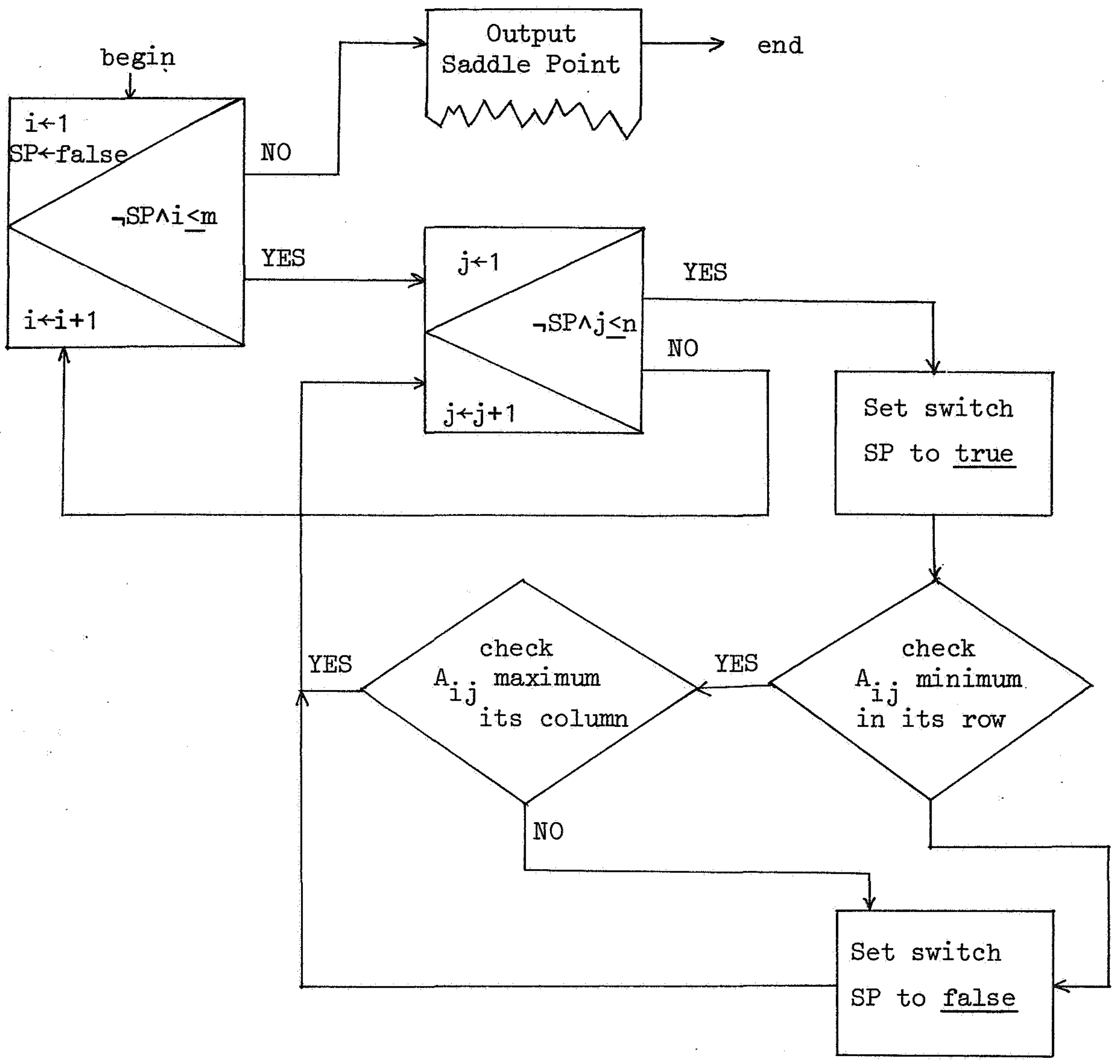
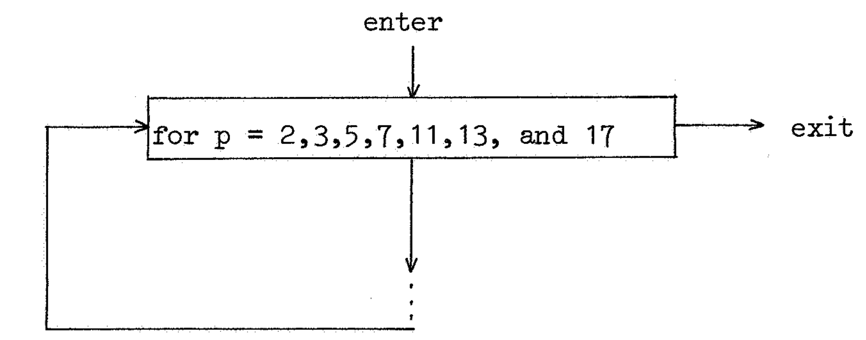
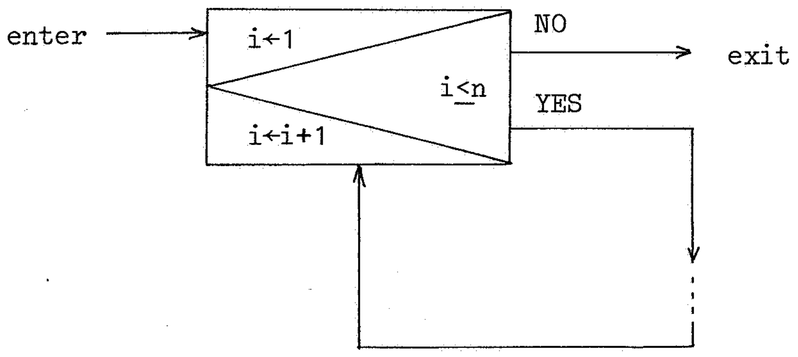


Figure 3.7. Possible flowchart for the saddle point program of Figure 3.1.

Figure 3.8. Possible flowchart notations for the list, increment (step - until), and while types of for statements.





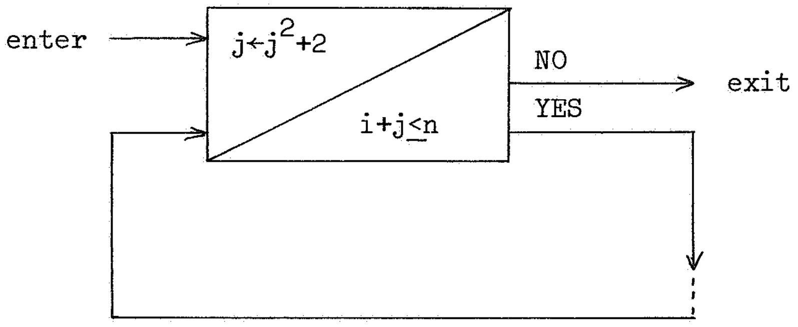
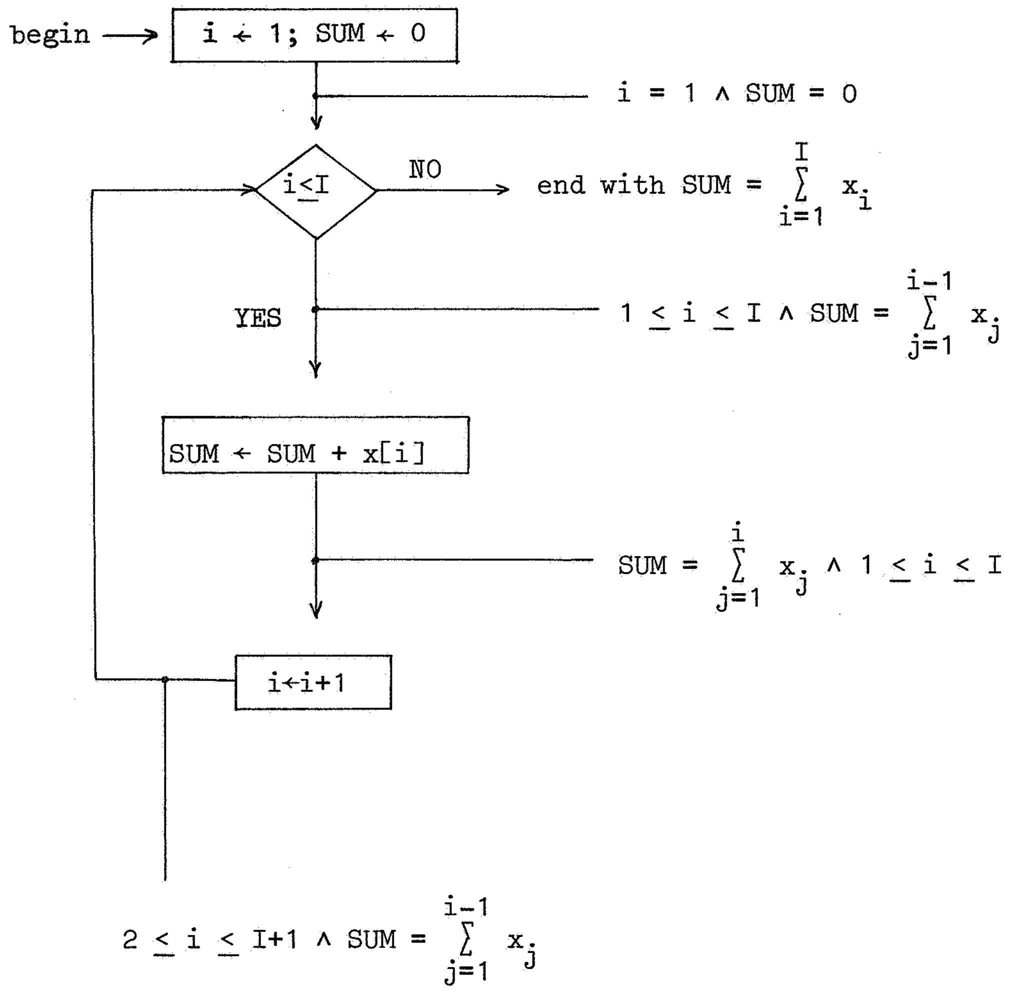


Figure 3.9. Flowchart labeled with key inductive assertions for computing the sum:

$$\sum_{i=1}^{I} x_i$$


begin
$i \leftarrow 1$; $SUM \leftarrow 0$
$i = 1 \wedge SUM = 0$
$i \leq I$
NO
end with $SUM = \sum_{i=1}^{I} x_i$
YES
$1 \leq i \leq I \wedge SUM = \sum_{j=1}^{i-1} x_j$
$SUM \leftarrow SUM + x[i]$
$SUM = \sum_{j=1}^{i} x_j \wedge 1 \leq i \leq I$
$i \leftarrow i+1$
$2 \leq i \leq I+1 \wedge SUM = \sum_{j=1}^{i-1} x_j$

EXERCISES

3.1. The saddle point program in Figure 3.1 is clearly inefficient with respect to comparisons made (see exercise 4.1). Write and prove correct a more efficient version.

3.2. It is desired to construct and prove correctness of an algorithm which will help prevent accidents between railroad trains. Suppose

n	= Number of trains on a particular track.
m_i	= Identification number of train i ($1 \leq i \leq n$).
P_i	= Position of front of engine of train i, measured from end of the track.
L_i	= Length of train i.
D	= Minimum allowable free distance between trains.

Construct and verify an algorithm which checks the spacing of the trains and output a message if two trains are too close to each other.

3.3. In the following ALGOL:

```
real x;
for x:= 0.3, x+0.3 while x ≠ 1.8 do begin
                                      :
                                    end
```

use of the rational ≠ is bad programming practice for at least two reasons. Explain.

3.4. (Newton's method for square root) Prove that successive guesses, equation (3.4), for $\sqrt{x}$ satisfy:

$$\sqrt{x} \leq y_{i+1} \leq y_i \qquad \text{for every } i > 0.$$

Derive formula (3.7) for the maximum number of steps before procedure ROOT (Figure 3.2) terminates. Prove that each application of Newton's rule squares the relative error under appropriate conditions.

3.5. Construct and prove correctness of a program for a binary search. Assume the element y being searched for is in the vector A[1:n] which is stored in ascending order.

3.6. The approximate cube root y of a number x > 0 may be calculated by Newton's method as:

$$y_{i+1} = \frac{1}{3}(2y_i + x/y_i^2) \qquad i = 0,1,2,\ldots$$

$$y_0 = 1$$

Construct and verify a program for this calculation.

3.7. Once an algorithm has been programmed, the question arises as to whether or not this program terminates for given input. For example, consider the "simple" program:

```
integer procedure f(n); integer n;
f:= if n = 1 then 0
    else if EVEN(n) then f(n÷2)
         else f(3*n+1);
```

Find the largest N such that the above procedure terminates for all n = 1,2,3,...,N. You are allowed only one minute of computer time to find this N!

3.8. There is a theoretical result on the subject of testing termination of a program which denies the existence of a general Boolean procedure T whose input is any program R such that T operates on R to yield:

$$T(R) = \begin{cases} \underline{\text{true}}, \text{ if R terminates when run} \\ \underline{\text{false}}, \text{ if R does not terminate.} \end{cases}$$

Here is an <u>informal</u> proof that no such procedure T can be programmed:

Assume, contrariwise, that T does exist and terminates for every input program R. Then consider program P defined by

```
program P.
loop: if T(P) then go to loop else stop.
```

and which uses procedure T as a subroutine. Hence if T(P) = true, then program P will loop forever. If T(P) = false, then P terminates. In each case T has exactly the wrong value.
This contradiction shows that T cannot exist.

This result means, roughly, that nobody can write a general program which would successfully check everyone elses program for termination. However, special programs can be written which verify termination sometimes. It is a termination checking program which works every time that is impossible.

Determine whether or not the function f defined below terminates for all positive input (n>0).

```
integer procedure f(n); integer n;
f:= if n > 100 then n-10 else f(f(n+11));
comment For example, the computation sequence of f(99) is:
f(99) → f(f(110)) → f(100) → f(f(111)) → f(101) → 91.
```

In this case (n=99) we say that f(99) is defined and f(99) = 91. When the computation sequence is infinite, we say that f(n) is undefined for that value of n;

3.9. Find an argument n for which the following recursive function g does not terminate:

```
integer procedure g(n); integer n;
g:= if n > 100 then n-11 else g(g(n+11));
```

3.10. What is your personal opinion about program correctness proofs?

3.11. Describe errors in the ALGOL program whose variables change table is given by:

Identifier	Declared at line	Changed at line	Used at line
i	2	7,19	9,35
alpha	5	40	35
Y	5	15	--
start (label)	2	--	6,25
stop (label)	100	--	--
A[ ] (array)	4	--	45,55
B[ ] (array)	5	16	18,23
C[ ] (array)	--	25	30,31
D[3] (array)	2,5	7	30
sort (procedure)	8-13	--	50,55
log 2 (procedure)	--	--	60

SOLUTIONS

3.1. Beware of equal elements in a given row or column.

3.2. Did you assume trains are arranged in the order i = 1,2,...,n along the length of the track?

3.3. (i) Because of precision problems, x may never equal 1.8 exactly. On many machines, the real integer 3 is represented internally as $0.3*10^1$; therefore the following code is just as bad:

```
real x, y;
for y:= 3, y+3 while y ≠ 18 do
                 begin x:= y/10;
                   ⋮
                 end;
```

(ii) If x accidently "skips" the value 1.8 (say, by a machine error or by an improper assignment between begin - end or by a jump into the for loop), then the for statement again may loop forever.

(iii) The ≠ relational means that the correctness proof must consider $x > 1.8$. Also, understanding the program becomes more difficult: Will x ever exceed 1.8?

3.4. $$y_{i+1} - \sqrt{x} = \frac{1}{2}(y_i - 2\sqrt{x} + x/y_i)$$
$$= \frac{1}{2y_i}(y_i^2 - 2\sqrt{x}y_i + x)$$
$$= \frac{1}{2y_i}(y_i - \sqrt{x})^2 \geq 0, \qquad \text{since } y_0 = 1 > 0.$$

$\therefore y_{i+1} \geq \sqrt{x}$ for all $i \geq 0$.

$i > 0$ implies:

$$\frac{y_{i+1} - \sqrt{x}}{y_i - \sqrt{x}} = \frac{1}{2y_i}\frac{(y_i - \sqrt{x})^2}{(y_i - \sqrt{x})} = \frac{y_i - \sqrt{x}}{2y_i} < 1, \qquad \text{since } y_i \geq \sqrt{x} \text{ if } i > 0.$$

$\therefore y_{i+1} < y_i$.

3.5.
```
array A[1:n]; integer i,j,k,n;
i:= 1; k:= n;
for j:= (i+k)÷2 while y ≠ A[j] do
if y < A[j] then k:= j-1 else i:= j+1;
comment j is the index of the element selected for comparison.
        i and k are the indices of the terminal elements of the
        remaining subset. If n = 2^m, then for a uniform distribution
        of arguments, the expected number of comparisons required in
        a binary search is
```

$$((m-1)2^m+m+2)/2^m,$$

which is approximately $(m-1)$. For general n, the number is approximately $\lceil \log_2 n \rceil - 1$, which differs only slightly from the worst case of $\lceil \log_2(n+1) \rceil$;

3.8. There is a known, but difficult, termination proof for this "famous 91-function". Its output value is always 91.

3.9. The computation sequence of g(99) is:

$$g(99) \rightarrow g(g(110)) \rightarrow g(99) \rightarrow g(g(110)) \rightarrow g(99) \rightarrow \dots$$

i.e., the computation sequence is infinite and hence g(99) is undefined (g(99) does not terminate).

3.10. Remarks on correctness proofs:

I suppose a "good" test case is one which uncovers at least one error. Therefore, if your program is correct, there are no "good" test cases, by this definition! Test cases may be "sufficient" for some programmers, but they are never "rigorous" for proving correctness.

Program correctness proofs are difficult. Mathematicians and machines may be better at proving correctness, but then they are also better programmers!

A correctness proof for a program which calculates the mean is not so trivial when you worry about accuracy.

You can learn about and concentrate on a program while discovering the inductive assertions for a correctness proof.
It is impossible to test for _all_ cases ...

3.11. The identifiers sort, B, and i seem O.K. Label "start" is declared before the declaration for procedure "sort"; hence a _possible_ error. Variable "Y" is never used, but _perhaps_ it is a counter like in:

```
for Y:= 1 step 1 until n do w:= w+u;
```

Variable "alpha" is _probably_ undefined because it is used in a line above where it is changed.
Array "D" is _probably_ multiply declared, although the two declarations could be in different blocks. Label "stop" appears to be redundant (not necessary), but _maybe_ it is being used as a comment.
Procedure "log2" must be in the library; otherwise it is undeclared.
There are only _two certain errors_: Array "A" is undefined (its elements are never given values) and array "C" is undeclared.

4. Frequency analysis

A common and obvious measure of performance for a computer program is running time. We can consider the execution time for a program in terms of:

(i) the worst case (maximum time used under the last favorable choice of inputs).

(ii) the best case (minimum time used under the most favorable choice of inputs).

(iii) an average case (expected time under a given input distribution).

(iv) the exact amount (the analytic formula for running time as a function of arbitrary input).

(v) an empirical estimate (an empirical formula fitted to certain input parameters).

However, rather than give the running time in seconds for a particular computer, we will count the number of times each step is executed. Clearly, the time required to perform an algorithm can always be determined when you know the number of times each step is executed. These counts, then, give us an essentially machine-independent method for the determination of running time.

Example. We illustrate the five above possibilities by a simple example. See Figures 4.1 and 4.2. The profile (collection of frequency counts) for program M is:

Step number	Number of times
M1	1
M2	n
M3	n-1
M4	A

The value of n is given, but we do not know the quantity A, which is the number of times we must change the value of the current maximum m. We thus study the possibilities for A:

(i) the worst case (for pessimistic people) is A = n-1 when y[1] > y[2] > ... > y[n].

(ii) the best case (for optimistic people) is A = 0 when y[n] is a maximum element of array y.

(iii) an average value (for probabilistic people) for A lies between 0 and n-1. In particular, based on the assumptions that the n input values y[1],y[2],...,y[n] are distinct and that each of the n! permutations of these values are equally likely, the average value of A is approximately ln(n) when n is large (Knuth, section 1.2.10, 1968).

(iv) the exact value of A does not depend on what the precise values of the y[k] are; only the relative order is involved. However, no simple formula for the exact value of A as a function of n and the relative order of the y values is available. Particular cases, such as n = 3 and y[3] > y[2] > y[1] for which A = 2, must be treated separately.

(v) an empirical estimate, n/3, for the value of A could be based upon, say, five samples

Sample	n	A
1	5	2
2	10	3
3	50	16
4	100	30
5	500	170

by fitting a straight line using the least squares criteria.

General principles. The following are general principles of attack for performing a frequency analysis:

1. Label a flowchart of the algorithm and apply "Kirchhoff's" conservation law for flowcharts (the amount of flow into each node must equal the amount of flow going out).

2. Reduce and identify the flowchart variables by using important characteristics of the problem.

3. Explore the behavior of the final profile parameters (worst case, average case, asymptotic, etc.).

Example. Let us now do a frequency analysis to illustrate these principles. Figure 4.3 is the flowchart of an algorithm, called T, found in (Alanen 1972). The first step is to label this flowchart; we have labeled the arrows with $a_1, a_2, \ldots, a_{11}$ and use the notation that step Ti is executed x_i times. Kirchhoff's law is

"sum of a's into box Ti = x_i = sum of a's leaving box".

It yields the equations:

$$
\begin{aligned}
x_1 &= 1 \\
x_2 &= 1 + a_9 + a_{12} = a_1 \\
x_3 &= a_1 + a_4 = a_2 + a_3 \\
x_4 &= a_2 = a_{10} + a_{11} \\
x_5 &= a_{11} = a_{12} \\
x_6 &= a_7 + a_{10} = a_6 \\
x_7 &= a_8 = a_9 \\
x_8 &= a_5 = a_4 \\
a_6 &= a_5 + a_8 \\
a_3 &= a_7 + 1
\end{aligned}
$$

We next reduce the number of unknowns by elimination of x_3, x_5, and x_6 using the above equations:

$$
\begin{aligned}
x_1 &= 1 \\
x_2 &= a_1 \\
x_3 &= x_2 + x_8 \\
x_4 &= a_2
\end{aligned}
$$

$x_5 = x_2 - x_7 - 1$

$x_6 = x_7 + x_8$

$x_7 = a_8$

$x_8 = a_5$

Kirchhoff's law does not completely determine the number of times each step is executed. More exactly, if there are n boxes and m arrows, then Kirchhoff's law allows us to eliminate n-1 unknowns among the arrows (<u>not</u> the boxes). In our example, $n = 10$ boxes and $m = 15$ arrows; we eliminated $n-1 = 9$ arrows (namely, $a_3,a_4,a_6,a_7,a_8,a_9,a_{10},a_{11},a_{12}$) which left us with the six unknowns $a_1,a_2,a_5,a_8,1$ (into T1), and 1 (the "done" exit). Related to the boxes we are left with five unknowns (x_1,x_2,x_4,x_7, and x_8). Clearly $x_1 = 1$ and we can further eliminate x_8 because k is initialized to zero (at step T1) and then the algorithm terminates only when $k = 0$. Thus for every time k is increased by one in step T5, k must be decreased by one in step T8; that is, $x_8 = x_5 = x_2 - x_7 - 1$.

There remain three unknowns (x_2,x_4,x_7) and to interpret them by relating them to pertinent characteristics of the data requires knowledge of what Algorithm T does. Since Algorithm T is rather complicated, we simply state the final answer in Figure 4.4. It turns out that the profile for Algorithm T (Figure 4.4) depends on three unknowns (α, β, and γ) which can be related to the input n.

Lastly, we remark on the behavior of the quantities α, β, and γ as n increases. The quantity γ can easily (if you understand Algorithm T) be shown to be small; indeed, when n-1 is prime, $\gamma = 2$. The quantity $\alpha-\beta$ seems to grow as $n^{0.815}$, which predicts observed values within relative error 3%. This is strictly an empirical result arrived at by fitting the curve $\alpha-\beta = n^a$ (a unknown) to the profile date in Figure 4.4. Similarly, β increases a little faster than $0.2\ n^{1.6}$. These empirical estimates for $\alpha-\beta$ and β were derived because it was not possible to deduce an exact (or even worst or average case) formula for their behavior in terms of n. Using these empirical estimates, Algorithm T would perform about 10^7 steps to handle the case $n = 50000$.

In summary, to derive the profile in Figure 4.4 we labeled the arrows and boxes in the flowchart for Algorithm T. Then we applied Kirchhoff's law to relate and eliminate these unknowns. Next we eliminated or identified the remaining unknowns by applying our knowledge of this particular algorithm. We were left with three parameters (α,β,γ) which depend upon the input n in complicated ways. Finally, we did an empirical study of the behavior of these final profile parameters. Our conclusion was that Algorithm T has a total running time proportional to n^2.

Local and global analyses. There are generally two kinds of frequency analyses, "local" and "global". A local frequency analysis investigates the running time requirements of some particular algorithm; a global frequency analysis, on the other hand, considers an entire family of algorithms and attempts to identify one that is "optimal", in the sense of using the least computer time.

Recall Algorithm M (Figure 4.1 and 4.2) which is a straightforward procedure for finding the maximum (or minimum if you change the inequality to $<$ in step M3) of a list of n numbers.

In terms of comparisons among the elements of array x, our local frequency analysis showed that algorithm M always requires n-1 comparisons. If you had some special knowledge of the x-values, you might be able to avoid some of these n-1 comparisons. Clearly, you could write an (inefficient) program which took more than n-1 comparisons.

A global analysis will show that the worst case optimal algorithm with respect to comparisons is also Algorithm M; that is, in its worst case (which is every case) it requires n-1 comparisons. No other algorithm requires fewer than n-1 comparisons in its worst case.

The proof that Algorithm M is "the best" algorithm for computing a maximum (minimum) is difficult. See exercise 4.3 for a related problem.

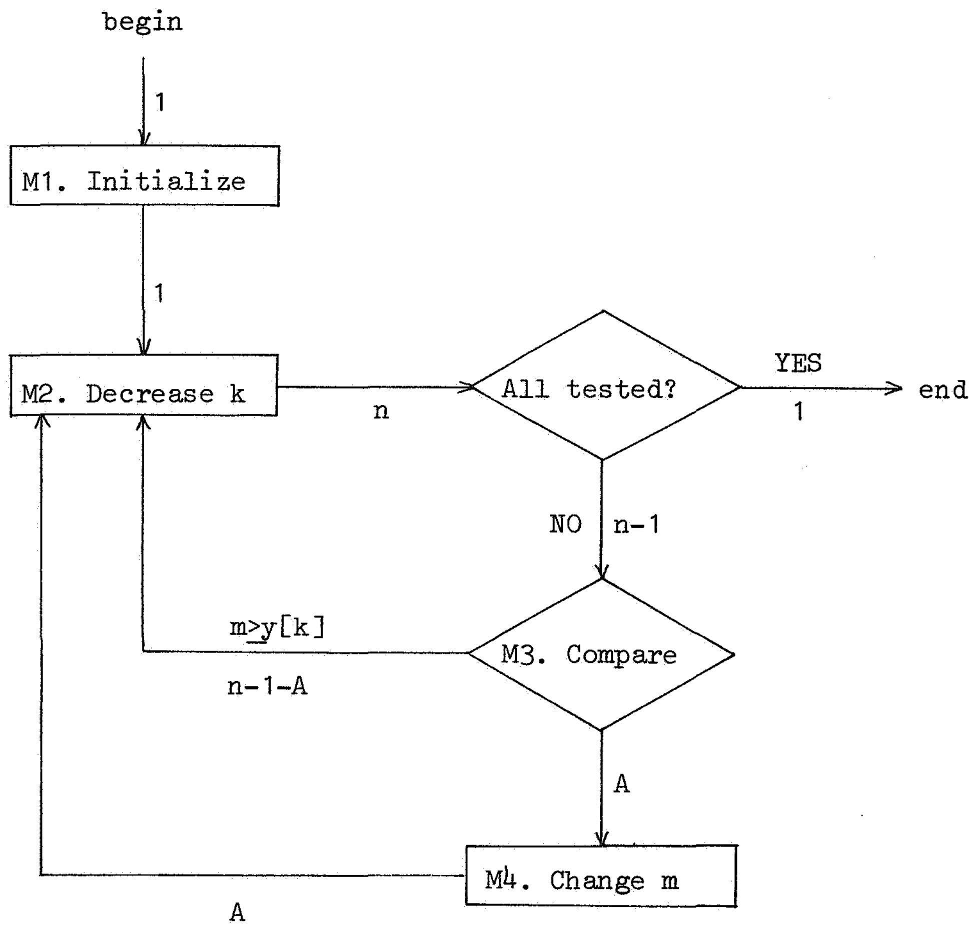
Figure 4.1. Program M to find the maximum

$$m = \max_{1 \le k \le n} y[k] = y[j]$$

such that j is as large as possible.

```
integer k, j, n; array y[1:n];
M1: j:= k:= n; m:= y[n];
M2: for k:= k-1 while k > 0 do
    M3: if y[k] > m then M4: begin j:= k;
                                   m:= y[k]
                             end;
```

Figure 4.2. Flowchart for program M in Figure 4.1. Labels on the arrows indicate the number of times each path is taken.



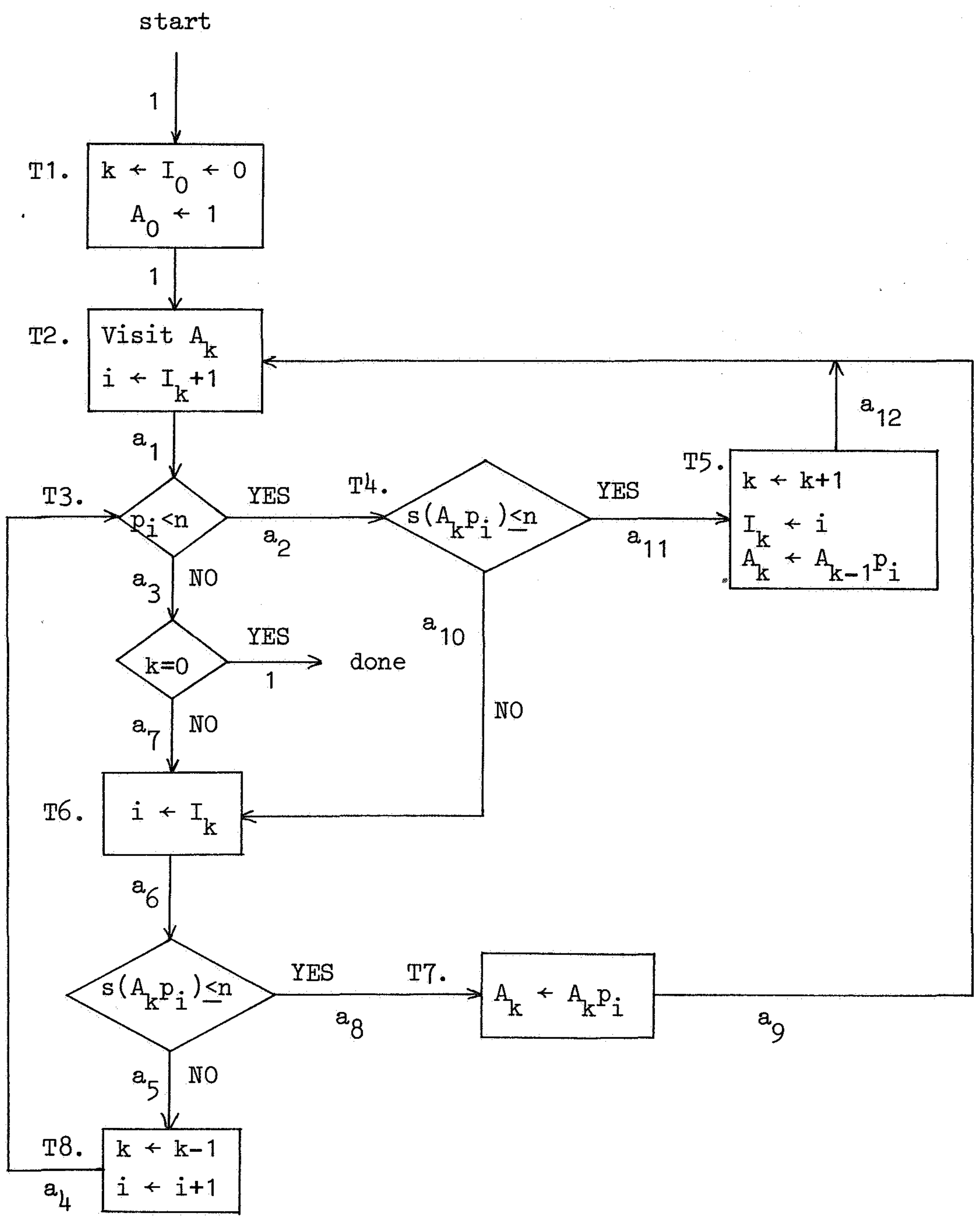


Figure 4.3. Flowchart of Algorithm T.

Figure 4.4. Profile of Algorithm T.

Step	Times each step is executed for given n				
	n=13	50	500	5000	general
T1	1	1	1	1	1
T2	19	114	3157	134550	α
T3	30	203	6160	268077	$\alpha+\beta$
T4	27	198	6157	268074	$\alpha+\beta-\gamma-1$
T5	11	89	3003	133527	β
T6	18	113	3156	134549	$\alpha-1$
T7	7	24	153	1022	$\alpha-\beta-1$
T8	11	89	3003	133527	β

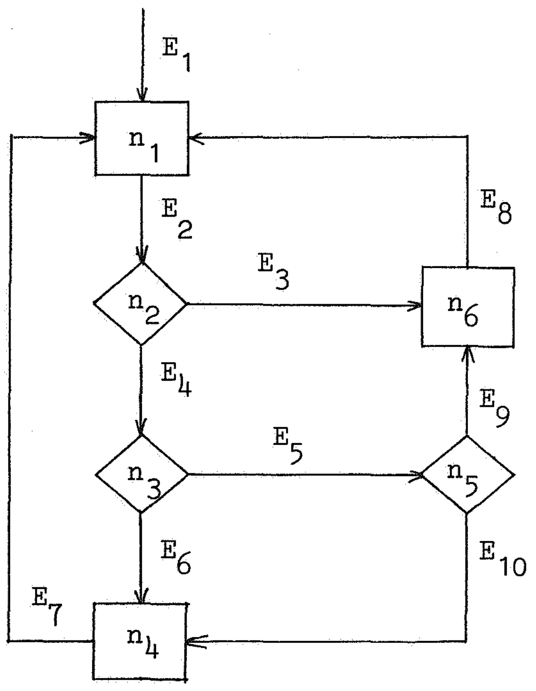
EXERCISES

4.1. Perform best and worst case analyses for the number of comparisons required by the saddle point program in Figure 3.1.

4.2. An improved I/O subroutine was written in one man-year (ƒ20000) using three hours of X8 time (ƒ1000/hour). It is 10% faster than the old version, which was used twice each day for 36 seconds per run. How long before the improved version "pays its way"?

4.3. Find the worst case optimal algorithm, with respect to comparisons, which computes both the maximum and the minimum of a list of n numbers $X_1, X_2, \ldots, X_n$. Note that this "best" algorithm executes $\lceil \frac{3}{2} n \rceil - 2$ comparisons and is somewhat wasteful of storage.

4.4. Use Kirchhoff's law to analyze the flowchart below so that boxes $n_1, \ldots, n_6$ are expressed completely in terms of the five arrow unknowns $E_1, E_3, E_6, E_9, E_{10}$.

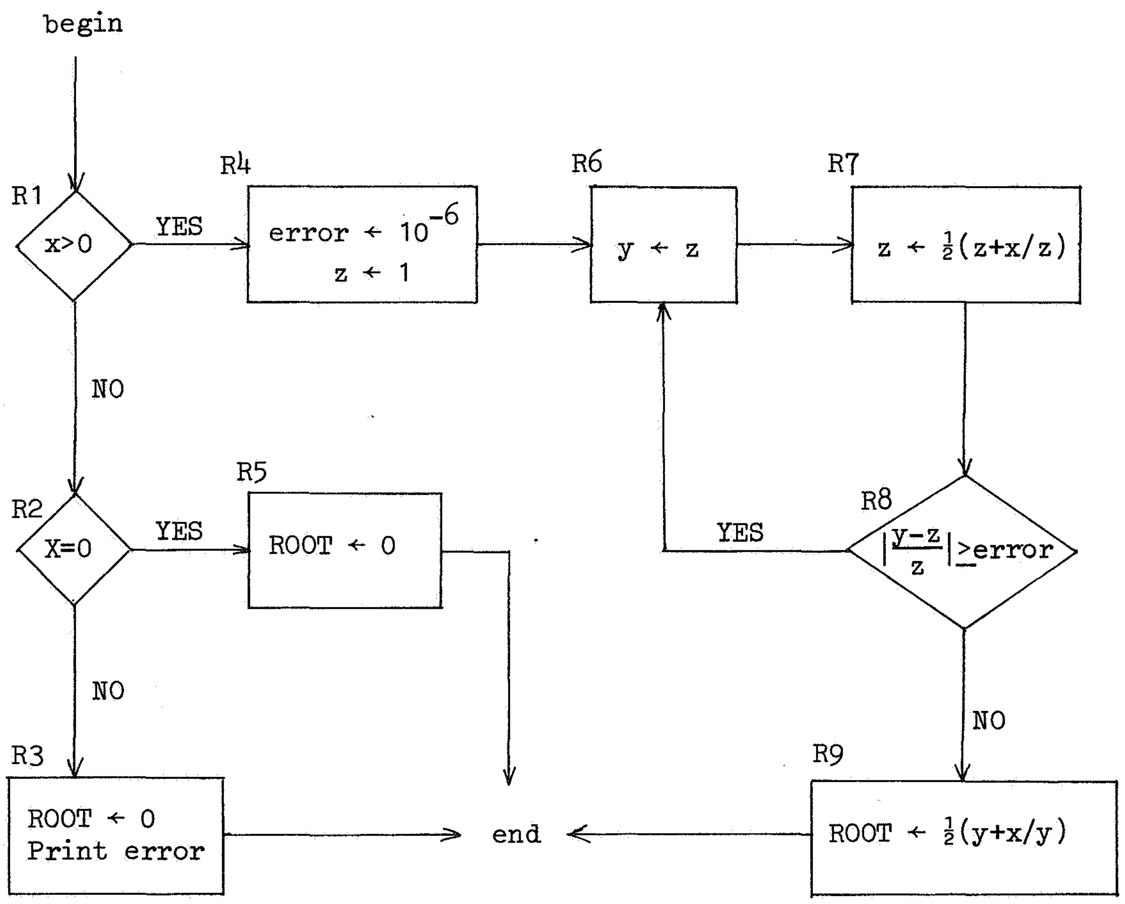


4.5. Find the worst case optimal algorithm with respect to comparisons for computing the median.

4.6. Time and memory are often used to measure the performance of a program. Describe <u>three</u> more things to look at in order to see if a program is "good".

4.7. The following flowchart is for procedure ROOT (Algorithm R) in Section 3. Fill in the profile below for this algorithm, assuming exact arithmetic. For the average case assume the three input possibilities ($x<0$, $x=0$ and $x>0$) are equally likely.

<u>Flowchart for Algorithm R</u>



Profile for Algorithm R

	Time executed			
Step	Minimum	Maximum	Average	Exact
R1	1	1	1	1
R2				
R3				
R4				
R5				
R6				
R7				
R8				
R9	0	1	1/3	c

where $c \equiv \frac{1}{2}[\mathrm{sign}(x) + \mathrm{sign}(|x|)]$ and $\mathrm{sign}(x) = 1,0,-1$ if $x > 0, = 0, < 0$.

SOLUTIONS

4.1. Worst case

mn(m+n) comparisons

$$A = \begin{pmatrix} 1 & 1 & \cdots & 1 & 0 \\ 2 & 2 & \cdots & 2 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ m & m & \cdots & m & m-1 \end{pmatrix}$$

saddle point = A[m,n] = m-1

Best case

m+n comparisons

$$A = \begin{pmatrix} 1 & 2 & \cdots & 2 \\ 0 & & & \\ \vdots & & * & \\ 0 & & & \end{pmatrix}$$

saddle point = A[1,1] = 1

4.2. ƒ20000 + ƒ3000 = ƒ 23000 (Cost)

2×360×.01×.1×1000 = ƒ 720/year (Saved)

32 years later!

4.3.
```
comment Minimum and maximum calculation using $\lceil \frac{3}{2} n \rceil - 2$ comparisons
        (worst case optimal);
integer i,k,n; array X[1:n],A,B[1:$\lceil \frac{n}{2} \rceil$];
k:= 1;
for i:= 1 step 2 until n-1 do
if X[i] < X[i+1] then begin A[k]:= X[i];
                            B[k]:= X[i+1];
                            k:= k+1
                      end
else begin A[k]:= X[i+1];
           B[k]:= X[i];
           k:= k+1
     end;
if odd (n) then begin if A[1] < X[n] then B[k]:= X[n]
                      else A[1]:= X[n];
                end;
comment Now find the minimum element in array A and the maximum element
        in array B using the usual optimal algorithms (Algorithm M in
        Figure 4.1);
```

4.4. From Kirchhoff's law we have the equations:

$$n_1 = E_1 + E_8 = E_2$$
$$n_2 = E_2 + E_7 = E_3 + E_4$$
$$n_3 = E_4 = E_5 + E_6$$
$$n_4 = E_6 + E_{10} = E_7$$
$$n_5 = E_5 = E_9 + E_{10}$$
$$n_6 = E_3 + E_9 = E_8$$

We can easily eliminate E_2, E_4, E_5, E_7, and E_8 as follows:

$$E_2 = E_1 + E_8 = E_1 + E_3 + E_9$$
$$E_5 = E_9 + E_{10}$$
$$E_4 = E_5 + E_6 = E_9 + E_{10} + E_6$$
$$E_7 = E_6 + E_{10}$$
$$E_8 = E_3 + E_9$$

The final equations for the boxes are:

$$n_1 = E_1 + E_3 + E_9 = E_3 + E_9$$
$$n_2 = E_1 + E_3 + E_6 + E_9 + E_{10} = E_3 + E_6 + E_9 + E_{10}$$
$$n_3 = E_6 + E_9 + E_{10}$$
$$n_4 = E_6 + E_{10}$$
$$n_5 = E_9 + E_{10}$$
$$n_6 = E_3 + E_9$$

But this implies that $E_1 = 0$ or $E_j = \infty$ for some j, because there is no exit arrow in the flowchart (i.e., an infinite loop exists iff you enter at E_1)!

4.5. No algorithm for computing medians is known which takes less than n log(n) comparisons in the worst case. And no proof that n log(n) comparisons are necessary has been found.

4.6. Ease of expression. (For example, a recursive program is often easier to read than the iterative version.)

Accuracy. (What is the precision of the output?)

Adaptability. (Can you change the program easily? For examples, are constants parameterized and how can you vary the precision?)

Reliability. (Are your data structures protected from illegal use? Is there subscript checking?)

Economy of representation. (For example, a transfer of control statement is ";goto label;" (ALGOL 60) or ";label;" (ALGOL 68) or "→ label" (APL\360)).

Portability (Can you easily run your program on another computer or even on the same computer somewhere else?)

Robustness. (Does the program give good results even when the input is approximate?)

4.7. Profile for Algorithm R

		Times executed		
Step	Minimum	Maximum	Average	Exact
R1	1	1	1	1
R2	0	1	2/3	1-c
R3	0	1	1/3	c-sign(x)
R4	0	1	1/3	c
R5	0	1	1/3	$1-\lvert\text{sign}(x)\rvert$
R6	0	$^*5+\lvert\log_2\sqrt{x}\rvert$	$\frac{1}{3}(5+\lvert\log_2\sqrt{x}\rvert)$	$c(5+\lvert\log_2\sqrt{x}\rvert)$
R7	0	$5+\lvert\log_2\sqrt{x}\rvert$	$\frac{1}{3}(5+\lvert\log_2\sqrt{x}\rvert)$	" "
R8	0	$5+\lvert\log_2\sqrt{x}\rvert$	$\frac{1}{3}(5+\lvert\log_2\sqrt{x}\rvert)$	" "
R9	0	1	1/3	c

where $c \equiv \frac{1}{2}[\text{sign}(x) + \text{sign}(|x|)]$ and sign(x) = 1,0,-1 if $x > 0$, $= 0$, < 0.

* $\approx$ 1000 when $x = 10^{628}$.

5. Storage analysis

In this section we consider storage analyses of computer algorithms, i.e. how much memory they are likely to need. It is important to acknowledge that "storage analysis" in this section will refer solely to the analysis of memory requirements for data structures; we will not examine the amount of memory needed for the program itself, that is, for the instructions stored in the computer memory. Furthermore, the data structures of an algorithm may best be chosen with consideration of the class of operations to be done on the data. If you repeatedly insert into the middle of a list, for instance, then a linked list data structure is preferrable to a sequential list, in order to minimize the steps executed in doing the many insertion operations. But linked lists usually require more memory than sequential lists, so there are tradeoffs here between storage and running time. Because we want to focus on data storage used, in this section we will not worry about running time. We thus postpone to later the important interrelations between a frequency analysis and a storage analysis.

Computers have both internal (usually core storage) and external (tapes, disk, drums) memory for storing data structures. Internal memory tends to have rapid (random) access time but is limited in size (64K for the X8). External memory has orders of magnitude greater capacity but a slower access time (usually due to its serial nature). To simplify the storage analyses in this section, we will ignore external memory and hence attempt to fit our data structures into internal memory alone. Analysis of algorithms which segment the data between internal and external memory will not be discussed.

Just as in the case of a frequency analysis (section 4), there are generally two kinds of storage analysis, "local" and "global". A local storage analysis investigates the memory requirements of some particular algorithm; a global storage analysis, on the other hand, considers an entire family of algorithms and attempts to identify one that is "optimal", in the sense of using least memory. In both kinds of storage analysis we can consider the amount of memory needed in terms of:

(i) the worst case (maximum storage used under the least favorable choice of inputs).

(ii) the best case (minimum storage used under the most favorable choice of inputs).

(iii) an average case (expected storage used under a given input distribution).

(iv) the exact amount (the analytic formula for storage used as a function of arbitrary input).

(v) an empirical estimate (an empirical formula fitted to certain input parameters).

Example. We illustrate these five possibilities by a simple example. Suppose we have n+1 inputs:

n,X[1],X[2],...,X[n]

and we wish to save only those values of X[i] > 0 ($1 \le i \le n$) in an array T. Figure 5.1. specifies a program to accomplish this. For a local storage analysis of this program, we investigate how large the variable k gets:

(i) the worst case is clearly when X[i] > 0 for all $1 \le i \le n$. Then the program saves n elements in T.

(ii) the best case corresponds to n = 0 or to X[i] $\le$ 0 for all $1 \le i \le n$. Then no locations are used in array T.

(iii) An average value for the upper subscript bound of array T is n/2, based on the assumption that positive and nonpositive inputs X[i] are equally likely (Prob{X[i]>0} = $\frac{1}{2}$ = Prob{X[i]$\le$0} for $1 \le i \le n$).

(iv) the exact value of k after execution of the program is always

$$\frac{1}{2} \sum_{i=1}^{n} (\text{sign}(X[i]) + \text{sign}(\text{abs}(X[i]))),$$

where

$$\mathrm{sign}(E) = \begin{cases} 1, & \text{if } E > 0. \\ 0, & \text{if } E = 0. \\ -1, & \text{if } E < 0. \end{cases}$$

(v) An empirical estimate, n/3, of the final value of k could be based upon, say, five samples

Sample	n	Final k
1	5	2
2	10	3
3	50	16
4	100	30
5	500	170

by fitting a straight line using the least square criteria.

Which of these five estimates for the final value of k is the most "meaningful"? Only the exact expression given in (iv) holds for every input combination, yet it may not be possible to evaluate it easily without an extra pass over the input values. The zero estimate of (ii) is clearly for optimists and not very useful in this particular problem. The pessimistic estimate (i), however, is the one most frequently used because programmers usually want their program to work for all inputs and thus use the maximum possible value of n for the upper subscript bound on array T. The average value (iii) is a statistical estimate and further straightforward analysis could be done to determine, for example, the value N such that:

$$\mathrm{Prob}\{k \leq N\} = 0.90 \text{ for fixed } n,$$

so that given n, an upper subscript bound of N for array T would imply 90% certainty of enough space being available. These statistical estimates are based on a specified input distribution whose appropriateness may be questionable. Lastly, the empirical estimate (v) provides a rough guess based upon some (hopefully) representative input samples.

Thus our local storage analysis of the program in Figure 5.1 reveals that anywhere from 0 to n locations are needed for storage into the T array, depending upon whether you prefer a best case, average case, exact, empirical, or worst case estimate of the final value of k.

Preliminary approximations. Will the input data and temporary results fit into memory? For a first approximate answer to this question, one should attempt an order of magnitude estimate of storage requirements. For example, 40000 locations of memory may handle the 200 by 200 matrix A, but where will you then put its computed inverse A^{-1}? One solution is to design a matrix inversion algorithm which stores A^{-1} on top of A. Another example: Suppose you need all permutations of seven numbers in an algorithm. This means storing 7! = 5040 seven-tuples, or 7 * 5040 = 35280 numbers if the permutations are generated "all at once". A solution to minimize memory would use a permutation algorithm which systematically generates every permutation given only its latest result. Lastly, if an algorithm requires all the prime numbers below X, then an asymptotic estimate for the number of such primes is well-known to be X/ln X. Hence $X = 10^6$ means storing approximately

$$10^6/\ln 10^6 \approx 10^6/(6*2.3) \approx 72000$$

primes.

Too much input. Next, we analyse several statistical problems with too much input data for memory. Such large volumes of input data are often produced by, for example, physicists in their experiments or companies in their management files. The need to "reduce" these data to summary statistics (means, medians, variances, correlations, etc.) is widespread.

Sample variance. Suppose there are N data values X[1],X[2],...,X[N] and it is required to compute their sample variance:

$$(5.1) \qquad s_X^2 = \frac{1}{N}\sum_{i=1}^{N} (X[i]-\bar{X})^2, \quad \text{where } \bar{X} = \frac{1}{N}\sum_{i=1}^{N} X[i].$$

When N is large, memory capacity may be exceeded by inputing and saving the N values X[1],...,X[N]. To compute s_X^2 according to the above formula, we first need the value of $\bar{X}$. But $\bar{X}$ is a function of all the N input values, so that the terms $X[i] - \bar{X}$ in the sum are not computable until after every value $X[i]$ has been read in and $\bar{X}$ computed. Then it is too late, for we cannot save the N values read in. One solution is to make a second pass over the input. Another solution is to employ the identity:

$$(5.2) \qquad \sum_{i=1}^{N} (X[i]-\bar{X})^2 = \sum_{i=1}^{N} X[i]^2 - N\bar{X}^2,$$

which clearly allows us to evaluate s_X^2 in one pass over the input (See Figure 5.2). The technique of partitioning sums of squares (such as in equation (5.2)) is used frequently in statistical calculations.

Sample median. A simple algorithm to find the sample median $X_{0.5}$ of the distinct numbers $X[1],X[2],\ldots,X[2n+1]$ is to sort these $2n+1$ numbers so that $X[n+1]$ becomes the median (half of the numbers are less than $X_{0.5}$ and half exceed $X_{0.5}$). When $2n+1$ is large or when our goal is to compute $X_{0.5}$ using minimal storage, the algorithm of Figure 5.3 is recommended; it inputs the numbers $X[1],X[2],\ldots$ one at a time and saves as few as possible on its way to finding the median. This algorithm can be shown (see exercise 5.3) to be worst case optimal with respect to memory used, for computing the median of $2n+1$ distinct numbers.

Fitting straight lines. The well-known least squares estimates

$$a_1 = \frac{\Sigma(X_i-\bar{X})(Y_i-\bar{Y})}{\Sigma(X_i-\bar{X})^2} \quad \text{and} \quad a_0 = \bar{Y} - a_1\bar{X},$$

for the slope and intercept of a straight line $Y = a_0 + a_1X$ can be rearranged to

$$a_1 = \frac{n\Sigma X_iY_i - \Sigma X_i\Sigma Y_i}{n\Sigma X_i^2 - (\Sigma X_i)^2} \quad \text{and} \quad a_0 = \frac{\Sigma Y_i - a_1\Sigma X_i}{n}.$$

This allows the computations to be performed with only one reading of the input data

$$n,X_1,X_2,\ldots,X_n,Y_1,Y_2,\ldots,Y_n$$

instead of the two passes ordinarily required when n is too large to save the input in storage. However, the input values must clearly be arranged in ordered pairs:

$$\begin{array}{l} n \\ X_1, Y_1 \\ X_2, Y_2 \\ \vdots \quad \vdots \\ X_n, Y_n \end{array}$$

to minimize storage. Ordering of input values in statistical programming to minimize storage requirements is a common and important technique.

We now turn our attention to the optimal allocation of storage for arrays, orthogonal lists, and tables.

Sparse matrices A commonly occurring matrix form is one in which many of the elements are zero. Such a matrix is called sparse. One scheme for storing the sparse matrix A[1:N,1:N] uses three vectors I,J,S[1:K] so that every nonzero element A[i,j] corresponds uniquely to some k ($1 \leq k \leq K$) for which

$$I[k] = i \wedge J[k] = j \wedge S[k] = A[i,j].$$

For example,

$$A = \begin{pmatrix} 0 & 1 & 0 & -3 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{pmatrix}$$

could correspond to

k	I[k]	J[k]	S[k]
1	1	2	1
2	1	4	-3
3	4	3	4
4	3	1	2

thereby saving $N^2 - 3K = 4^2 - 3 \times 4 = 4$ locations.

If the size N of the sparse matrix A is not large, then the storage saved by the above scheme is not a compelling reason to treat A differently from full matrices. But for large N and $\underline{O}(N)$ nonzero entries (typically, say 2 to 10 nonzero entries per row), the usual storage requirement of N^2 can be reduced by a factor of N in many instances. Such a savings may dictate whether or not some problems can be attempted.

Pointer indexing. An array declaration "array J[1:N];" in ALGOL causes storage to be reserved for N elements J[1],J[2],...,J[N] of vector J. If only selected elements

$$J[i_1],J[i_2],\ldots,J[i_n]$$

are stored, then it may save memory to use (index, element) pairs:

array i,Y[1:n];

j	i[j]	Y[j] = J[i[j]]
1	3	J[3]
2	47	J[47]
3	51	J[51]
⋮	⋮	⋮
n	300	J[300]

More explicitly, when 2n < N the above "pointer indexing" scheme conserves memory. We have already discussed pointer indexing as applied to sparse matrices. Contingency tables with impossible or empty entries can also be stored efficiently using pointer indexing. For example, a contingency table for the three variables

AGE	=	1,2,...,100 years
SEX	=	0(male), 1(female), 2(unknown)
INCOME	=	0,1,2,...,1000000 dollars per year

based upon a population of 10000 persons is best stored in the form of three sequential vectors

i	AGE[i]	SEX[i]	INCOME[i]
1	30	1	7500
2	10	0	150
3	40	0	15000
⋮	⋮	⋮	⋮
10000	16	0	6000

because many of the $3 \times 100 \times 1000001 = 300000003$ combinations of sex, age, and income are either impossible (a two-year old earning one million dollars) or unlikely (women with annual incomes above \$ 10000).

Tables sharing memory. Two tables A[1],A[2],...,A[m] and B[1],B[2],...,B[n] can be arranged to coexist in memory by growing toward each other

A[1]	A[2]	...	A[m]	→ ←	B[n]	...	B[2]	B[1]

rather than having them kept in separate independently bounded areas. This means replacing the ALGOL declarations and references

array A[1:m],B[1:n];

A[i] $1 \leq i \leq m$

B[j] $1 \leq j \leq n$

with the following

array C[1:N];

A[i] = C[i]

B[j] = C[N-j+1]

C "overflows" when $m+n > N$.

Such table sharing has great storage advantages when the subscript bounds m and n fluctuate, but their sum m+n never exceeds N.

Symmetric matrices of order n × n such as

$$A = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 4 \\ 3 & 4 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 2 & 5 & 0 \\ 5 & 7 & 6 \\ 0 & 6 & 2 \end{pmatrix}, \quad n = 3$$

can be made to share memory

$$C = \begin{pmatrix} 1 & 2 & 5 & 0 \\ 0 & 1 & 7 & 6 \\ 3 & 4 & 1 & 2 \end{pmatrix} = \begin{pmatrix} A[1,1] & B[1,1] & B[1,2] & B[1,3] \\ A[2,1] & A[2,2] & B[2,2] & B[2,3] \\ A[3,1] & A[3,2] & A[3,3] & B[3,3] \end{pmatrix}$$

and thereby reduce storage requirements by

$$2n^2 - n(n+1) = n(n-1)$$

elements. In general, we replace

```
array  A,B[1:n,1:n];
```

with

```
array  C[1:n,1:n+1];
```

and use the definitions

$$A[i,j] = \begin{cases} C[i,j] & \text{if } i \geq j \\ A[j,i] & \text{if } i < j \end{cases}$$

$$B[i,j] = \begin{cases} C[i,j+1] & \text{if } i \leq j \\ B[j,i] & \text{if } i > j \end{cases}$$

Recursion depth. A "recursive solution" to a problem can offer clarity and conciseness over the corresponding iterative solution. For example, compare the recursive (Figure 5.5) versus iterative (Figure 5.6) solutions for traversing a binary tree in postorder. With respect to a storage analysis, however, the recursive solution may be more difficult to analyze, because the "depth of recursion" must be determined. In the iterative program (Figure 5.6), stack A saves a maximum of n elements, whereas the recursive program (Figure 5.5) has a maximum recursion depth of n, the maximum level of the tree. Thus the iterative solution makes you explicitly save values in a stack, while the recursive solution stacks automatically through recursive procedure calls.

Packing. The packing of data into computer words is a standard machine/assembly language technique. Because bit and/or byte manipulation is not machine independent, ALGOL 60 is not an ideal language for expressing these packing operations. However, it is possible to pack in ALGOL 60 by multiplying and dividing by appropriate powers of 2 or 10, thus saving storage.

It should be realized that X-8 ALGOL automatically packs Boolean arrays so that 27 elements occupy one X-8 word (each truth value requires one bit).

Figure 5.1. Program to save those input values (among n values read in) which are positive in an array T.

```
real array T[1:?];
integer i,k,n; real X;
i:= k:= 0; n:= read; printtext ("n="); print (n);
for i:= i+1 while i ≤ n do
        begin X:= read;
                  if X > 0 then begin k:= k+1;
                                      T[k]:= X
                                end
        end;
```

Figure 5.2. Program to compute the sample variance

$$\frac{1}{N}\left(\sum_{i=1}^{N} X[i]^2 - \frac{1}{N}\left(\sum_{i=1}^{N} X[i]\right)^2\right)$$

from the N+1 input values

$$N, X[1], X[2], \ldots, X[N]$$

using minimal storage.

```
integer i,N;
N:= read; printtext ("N="); print (N);
Squares:= Sum:= 0; i:= 0;
for i:= i+1 while i ≤ N do
        begin X:= read;
              Sum:= Sum + X;
              Squares:= Squares + X × X
        end;
Variance: = if N ≤ 0 then 0 else (Squares-(Sum×Sum)/N)/N;
```

Figure 5.3. Program to compute the sample median of 2n+1 distinct input values, using minimal storage in a worst case analysis.

```
comment procedure "sort" arranges the N elements of array X in increasing
        order: X[1] < X[2] < ... < X[N];

integer n,N,i; real array X[1:?]; real J;
n:= read; printtext ("n="); print (n);
if n ≥ 0 then begin
              i:= N:= 1; X[1]:= read;
              for i:= i+1 while i ≤ 2 × n + 1 do
                  begin J:= read;
                        if J < X[N] ∨ N ≤ n then
                        begin if N ≤ n then N:= N+1;
                              X[N]:= J;
                              sort
                        end
                  end;
              Median:= X[N]
              end;
```

Figure 5.5. Recursive procedure for traversing a tree in postorder.

```
procedure TRAVERSE(P); value P; integer P;
if P ≠ 0 then begin TRAVERSE(LLINK(P));
                    VISIT(P);
                    TRAVERSE(RLINK(P))
              end;
```

Figure 5.6. Iterative procedure for traversing a binary tree in postorder, making use of an auxiliary stack A.

```
procedure TBTREE(T); value T; integer T;
begin integer P,i; integer array A[1:n]; Boolean B;
      P:= T; i:= 0; B:= true;
      while B do if P ≠ 0 then begin i:= i+1; A[i]:= P;
                                     P:= LLINK(P)
                               end
                 else if i ≠ 0 then begin P:= A[i];
                                          i:= i-1; VISIT(P);
                                          P:= RLINK(P)
                                    end
                      else B:= false;
end;
```

EXERCISES

5.1. Discuss input possibilities for sparse matrices and how to handle them in MC-ALGOL 60.

5.2. Find a suitable small ($N \approx 4$) sample such that equations (5.1) and (5.2) give dramatically different results on a machine which uses only, say, six significant digit arithmetic.

5.3. Analyze the storage requirements for the median program of Figure 5.3.

5.4. Find an algorithm to compute the sample correlation coefficient

$$r = \frac{\sum_{i=1}^{N} (X[i]-\bar{X})(Y[i]-\bar{Y})}{N\, s_X\, s_Y}$$

which is optimal in storage needed.

5.5. Modify the median program (Figure 5.3) so that it will also handle samples of even size (2n) for which the median $X_{0.5}$ equals the average of the two middle observations.

5.6. A bank has computerized its savings accounts; in the computer program, an array B[1:N] contains the balance of account number i ($1 \leq i \leq n$) in element B[i]. Initially, there were n = 1000 savings accounts; experience predicts an 8% increase in the number of accounts each year. If each array element requires one memory location and there are N total memory locations available, how many years before the bank's program overflows memory? In particular, how many years before $n > N = 20000$?

5.7. Using the storage scheme described in this section, how sparse must an $N \times N$ matrix be to save storage?

5.8. Give the explicit function evaluated by the recursive procedure:

```
integer procedure f(x); value x; integer x;
f:= if x = 1 then 1 else x ↑ 2 + f(x-1);
```

What is the maximum recursion depth of this procedure for given $x \geq 1$? Write a nonrecursive version of this procedure and compare storage analyses.

5.9. Design general purpose MC ALGOL procedures to handle packing/unpacking of a Boolean vector B[1:n] stored 26 elements to an X-8 word, with 1 ⟷ <u>true</u> and 0 ⟷ <u>false</u>.

5.10. How can you estimate $\pi(x)$, the number of primes $\leq x$, without computing the primes up to x? Approximate $\pi(10^5)$. Code the primes to 10^5 so that they occupy less than 400 words of X-8 memory.

5.11. Write and storage analyze two ALGOL programs, one recursive and the other non-recursive, to evaluate the highest (greatest) common divisor of two positive integers, m and n. Use the test cases: (m,n) = (10,30), (60,14), (50,231), (261,0), (0,27), (38,57).

5.12. Analyze an efficient storage scheme for handling double precision symmetric matrices in ALGOL.
Same problem for hermitian matrices.

SOLUTIONS

5.1. Have program initialize all elements to zero and then read in only nonzero elements.

5.3. n+1 elements are saved in all cases.

5.6. $n_k = 1.08^k n$; $n_k > N \implies k > \frac{\ln N/n}{\ln 1.08} = \frac{\ln 20}{.0334} \approx 39$.

5.7. $3K < N^2 \implies \geq 33\ 1/3\%$ zero elements.

5.8. $f(x) = \sum_{i=1}^{x} i^2 = \frac{x(x+1)(2x+1)}{6}$

Maximum recursion depth = x.

```
integer procedure f(X); value X; integer X;
f:= (X×(X+1) × (2×X+1))/6;
```

Storage requirements:

(i) Recursive - X locations for procedure arguments.

(ii) Nonrecursive - 1 location for variable X.

5.10. $\pi(x) \sim x/\ln x$ as $x \to \infty$

$\pi(10^5) = 9592$

Use packing or Boolean array B[1:9592] with

B[i] = true iff i prime.

5.11.
```
integer procedure HCD(n,m); value n,m; integer m,n;
HCD:= if m > n then HCD(m,n)
      else if m = 0 then n
           else HCD(m, remainder(n,m));
comment remainder(n,m) = remainder of n/m;


integer procedure Euclid(n,m); value n,m; integer n,m;
begin integer r;
      if m > n then begin r:= m; m:= n; n:= r end;
      for r:= remainder(n,m) while r ≠ 0 do
          begin m:= n; n:= r end;
      Euclid:= n
end;
```

(Note first if is not necessary.)

$HCD(10,30) \equiv HCD(30,10) \equiv HCD(10,0) \equiv 10.$

5.12. A(double precision symmetric) = (most sig. \ least sig.)

H(complex) hermitian $\Longleftrightarrow$ $(H^T)^* = H$

H = (real part \ complex part)

$H^* = H^T \Longrightarrow H_{ii} = \text{real}, \quad H_{ij} = a + ib, \quad H_{ji} = a - ib.$

6. Measures of program performance

The aim of software engineering is, according to F.L. Bauer, "to obtain economically software that is reliable and works efficiently on real machines". In section 3 we considered software to be "reliable" when accompanied by a correctness proof. In sections 4 (Frequency analysis) and 5 (Storage analysis) we investigated the two most important and common aspects of software efficiency: running time and memory. However, there are numerous other aspects (such as robustness, portability, ease of expression, accuracy, and adaptability) of software which serve as measures of "goodness" or performance.

In order to provide some indication of an algorithm's merits relative to existing algorithms in the field, in this section we will define (if only through examples) some of these measures of performance and we will exhibit the kinds of questions you should ask to evaluate them and to make comparisons between them. Clearly, a program is "improved" when one aspect (time, memory, portability, accuracy, etc.) is done better while nothing else deteriorates. The trouble is that there are usually tradeoffs between these performance aspects. When you use double-precision arithmetic to increase precision, for instance, then running time increases. If you program in a high-level language in order to improve the portability of your programs, then you may lose the time efficiency of machine-dependent procedures.

Nevertheless, it is worthwhile to ask whether an algorithm offers possibilities for improvement. Such a question usually leads to the question "What is best under what circumstances?" A library procedure should surely meet high standards of performance, whereas with a novel, one-shot problem one is more likely to settle for any program that works. Recall the situation in section 2 where "best" for computing y^n depended upon whether division was allowed. In sorting, the number of comparisons is often used to measure performance; but when records are large you may be concerned more with minimizing interchanges than comparisons.

Absolute measures of performance are fine when available ("procedure sqrt computes the square root correctly to six significant digits") but comparative (benchmark) measures of performance are more common ("with a sample of student jobs, QUICKTRAN compiled an average of 60% faster than

FORTRAN version 6"). Indeed, strict theoretical bounds on performance may be over-pessimistic compared to actual practical performance. That theoreticians are always constructing weird counterexamples to prove that some algorithm is either not constructable or else not efficient in the general case has led Van der Poel to exclaim: "nearly all interesting problems of practical value are unsolvable!"

Before some measures of program performance are defined and discussed, I want to make several disclaimers: The following list of performance aspects is not complete nor detailed. For example, Sammet (1971) and Van der Poel (1972) study the problems of measuring and comparing programming languages in far more depth than I would even attempt here. Further, my condensed descriptions such as "robustness" and "ease of expression" are not necessarily standard terminology and are difficult to define rigorously.

Robustness. This is a measure of program stability, i.e. "how does the program behave under different data?" Suppose that a library procedure is advertised to solve a quadratic equation. Then it would be robust if it could handle the case of complex (imaginary) roots. Perhaps the procedure only complains (with an error message) rather than processes an equation with complex roots, but even this complaining is better than trying to divide by zero or outputting nonsense.

A statistical program (such as an analysis of variance procedure) that can cope with missing data (unbalanced ANOVA designs) would be termed "robust". A least squares procedure which handles nonlinear equations and a regression analysis program that will seek "the best stepwise fit" are also robust programs.

"Adaptability" is another measure of program performance to be covered later; it is mentioned here because it is closely related to robustness. Whether, for example, a program that permits transformation of the data should be called more robust (stable) or more adaptable (general) is an open, but rather academic, question.

Once again, by "robustness" we mean the stability of an algorithm. The question to ask is: How does this algorithm perform on problems harder or different than those for which the results are guaranteed good? A numerical integration technique with "graceful degradation" on problems containing singularities would qualify as robust.

Portability. This is a measure of the transferability of a program, i.e. "Will the program run on another machine?" Suppose a program is written in COBOL, a relatively machine-independent language. Can you then "easily" transport that program from a CDC 6000 computer to an IBM 360/65 computer? If the answer is yes, than that program would be called portable.

A language which is precisely defined, and for which translators exist that conform to the language specifications, is a great asset for producing portable software. Until recently, "FORTRAN" stood for a variety of languages on a variety of machines. Hence portability was a serious problem for FORTRAN users. Kahan (1971) documents the silly variability between FORTRAN compilers; for example, the two statements

```
X = 1.0 + 3/2

Y = 1.0 + (3/2)
```

sometimes yield two different results (namely X becomes 2.50, while Y equals 2.00).

The precision of a language's definition (ALGOL 68 and PL/1 are "well-defined" languages in my opinion) and the conformity of a language compiler to its specifications greatly influence portability. The United States Navy has a COBOL certifier; any compiler that successfully compiles and runs this collection of test programs is certified as a "COBOL compiler".

Portability has obvious tradeoffs with running time. That is why random number generators are usually coded in machine language, i.e. for the sake of speed, portability is ignored. When portability is not ignored, language and/or machine dependence must be avoided. Identical plot procedures for the printer, pen plotter, and cathode ray tube permit portability between plotting devices.

To discuss repeating a program on the same machine leads to related considerations of "reliability" (mentioned later).

In review, by "portability" we mean the transferability of a program. Questions are: Is this program machine dependent? Can I easily repeat the program using the same computer, a different machine, a different program library, another compiler, etc.?

<u>Ease of expression</u>. When stating an algorithm in a programming language, the economy and clarity of representation are relevant. Questions are: Are long mnemonic identifiers possible? Is recursion available? Can I define new operators? Does the language include graphical output and generalized input? For array arithmetic must I program loops or is explicit evaluation (as in APL\360) permitted? And so on.

Whether a certain notation is "clearer" can be debatable. Take recursion for example. A procedure in ALGOL 60

```
real procedure SUM(A,n); array A; integer n;
```

to compute $\sum_{i=1}^{n} A_i$ can be written iteratively

```
begin integer i; real S;
      S:= A[1];
      for i:= 2 step 1 until n do
      S:= S + A[i];
      SUM:= S
end;
```

or else recursively

```
SUM:= if n=1 then A[n] else A[n] + SUM(A,n-1);
```

and until you learn to "think recursively", the iterative statement may seem clearer. Because FORTRAN does not allow recursion, FORTRAN programmers often fail to realize the elegant possibilities in a recursive algorithm. The lack of compound statements in FORTRAN means that the FORTRAN programmer must constantly use goto's, an undesirable statement for structured programming.

See the solution to exercise 2.9 for a recursive definition of an operator in ALGOL 68.

There exist programming languages which permit easier expression of a certain class of problems. For examples, SNOBOL for string manipulation, LISP for list processing, APL for array operations, and COGO for plane geometry computations in surveying.

Default options can help you to write fewer marks. PL/1 is designed around default options and although ALGOL 68 is the opposite of a typeless language, it also includes default options in, for example, the for statement ("by 1", "while true", etc. can be omitted).

Good documentation as well as custom designed punched-cards/printer-paper make program preparation easier.

We mention the possibility of replacing a difficult analytic solution to a problem with a simulation (Monte Carlo) study. The birthday problem in exercise 6.6 is easier to simulate than to derive and compute the analytic formula.

In conclusion, "ease of expression" is often a debatable performance aspect in programming. Some people think English (such as used in COBOL) allows clearest expression of computer algorithms, while the other extreme is (perhaps) the APL\360 programmer who delights in producing the "one-liner":

```
0.05×+/(*-X)÷1+X×X←0.1×,(⍳110)∘.-0.5×1+ 1 ¯1 ×3*¯0.5
```

to evaluate by quadrature the integral:

$$\int_0^\infty \frac{e^{-x}dx}{1+x^2} \ .$$

Accuracy. Rounding and/or truncation, coupled with the finite representation of real numbers, in a computer lead to accuracy questions. For on a computer it is possible to have A+B = A even though B does not equal zero. Nor does the associative law, (A+B) + C = A + (B+C), always hold. When one computes 30! as

```
30 * 29 * 28 * ... * 3 * 2 * 1
```

or as

```
1 * 2 * 3 * 4 * ... * 28 * 29 * 30
```

on a computer, the answers differ (assuming eight-significant digit arithmetic with rounding after multiplication) by 10^{24}.

Another source of accuracy problems is error introduced by using approximate rather than exact data. In some problems a very slight change in the data produces a major shift in the output. For example, the system of simultaneous linear equations

$$2x - y = 1$$
$$2.001x - y = 2$$

has exact solution $x = 1000$, $y = 1999$. But a 0.05% change in the x coefficient of equation two (2.0010 becomes 2.0000) makes the system insolvable! And a 0.1% change (2.0010 becomes 1.9990) produces the solution $x = -1000$, $y = -2001$. Thus input must be accurate to avoid such instability.

Kahan (1971) gives numerous examples of serious troubles with existing FORTRAN and PL/1 dialects that stem from round-off, real number representations, overflow, and so forth.

An error message such as "Negative argument for square root" often actually means "To machine accuracy, matrix A at line 96 is not positive definite". The latter message is preferred because it pinpoints the real problem in the matrix inversion procedure.

Besides being aware of pitfalls in computation (Forsythe 1970) in order to select the best algorithm (See, for instance, the paper by Young and Cramer on choosing sum and sum-of-product algorithms), one must constantly be alert to accuracy situations that are potentially dangerous. As an example, when writing a foreign currency conversion program you should ask what would happen if there were a "small accuracy bug" in it. Big troubles can grow from little errors.

To be safer, multiple precision or interval arithmetic can be employed. They give more accurate results at the expense of time and memory.

If you never question the accuracy of your computer output, you deserve the nonsense you will sometimes get ...

Reliability. This is a measure of the protection a program has against operator, machine, program, and user failures. As always, a program is as reliable as its input, so input should always be verified. Similarly, calculations may be purposely duplicated to insure reliability (e.g. after computing c:= a÷b, the product c×b can be compared to a). Elaborate internal checks are mandatory in critical situations such as real-time space flights controlled by computers.

Maintaining the integrity of data structures is a key problem nowadays. Consider the typical newspaper report (Computerworld, vol. V, no. 52, page 11):

"Computer tampering was said to have been necessary in the theft of 217 Penn. Central Railroad boxcars. The cars were discovered on the tracks and yards of a tiny Illinois railroad. According to attorneys, someone 'had to put the fix' on the Penn Central's computers to shuttle the boxcars to the railroad and to 'make them disappear'."

Security against run-time errors will be provided by a good programming system in the forms of subscript range checking, mismatched parameter-argument values checking, exceptional arithmetic conditions checking, parity checking, and more.

When evaluating the reliability of a software system, always assume the worst will happen: operator drops program deck, card reader shuffles input, user submits wrong input tape (header labels help eliminate this error), control cards mispunched, etc.

Adaptability. This measures the generality of a program, i.e. "what is the effect of a slight problem change?"

Changes in data types (real to complex, vector to matrix, single precision to multiple precision) after the initial program is running, happen so frequently that it should be anticipated. Use parameters for array bounds, constants, etc. instead of fixed constants that are suitable only to the current situation. Use a language which allows complex arithmetic, multiple precision, etc. without great changes to the program.

A good programmer appreciates how problem statements tend to change drastically each time the program appears to be "near completion". A general program will be more easily adapted to such changes.

Closing remarks. We have now completed our considerations of the important questions of performance for a computer program. Through measurements of performance and computational efficiency, significant savings in computational effort can be achieved. However, for the user who will not take the time and trouble to search out the "best" program, the above techniques for evaluating performance characteristics of proposed computer algorithms will be of no value. Only a responsible user who questions the validity of the output and the efficiency of the processing will demand and compare analyzed computer programs. Quality scientific software will only become available when both programmers and users begin to worry about the analysis of computer programs.

EXERCISES

6.1. Construct and analyse a program which reads in temperatures in either degrees Fahrenheit or degrees Centigrade and outputs the temperature in the opposite scale.

6.2. In many matrix applications, such as the solution of simultaneous equations, it is required to check whether or not the matrix is symmetric, i.e. whether or not A[i,j] = A[j,i] for all i and j. Construct and analyse an algorithm to check an n×n matrix A for symmetry.

6.3. A program uses G + M storage locations and runs in approximately A/M + B time units. Choose M which gives the optimum product of space times time.

6.4. Suppose that airplane P_i has identifying number i and coordinates (x_i, y_i, z_i) for $i = 1,2,\ldots,n$. Construct and analyze a program which checks for a specified safe minimum distance, Dmin, between planes.

6.5. An output procedure used 2989 digits to number the pages of printed output. How many pages were output?

6.6. In a room containing n persons let Qn be the probability that there are two or more persons with the same birthday. It can be shown that

$$Qn = 1 - \frac{365!}{(365-n)!\ 365^n} \qquad \text{for } n = 0,1,2,\ldots,365.$$

Compute and tabulate (or plot) Qn versus n for n = 0(1)100.

6.7. Estimation of the size of an animal population from recapture data. Suppose that n_1 fish caught in a lake are marked by red spots and released. After a while a new catch of r fish is made, and it is found that k among them have red spots. We are interested in estimating the number n of fish in the lake. If $q_k(n)$ equals the probability that the second catch contains exactly k red fish, then

$$q_k(n) = \frac{\binom{r}{k}\binom{n-r}{n_1-k}}{\binom{n}{n_1}}, \quad \text{where } \binom{a}{b} = \frac{a!}{b!(a-b)!}$$

For $n_1 = r = 1000$ and $k = 100$ find the particular value of n for which $q_k(n)$ attains its largest value, since for that n our observations (100 red fish among the second sample of 1000 fish) would have the greatest probability.
This value is called the maximum likelihood estimate of n.

6.8. An asymptotic approximation (for large n) to n! is given by Stirling's formula:

$$n! \sim \sqrt{2\pi}\, n^{n+\frac{1}{2}}\, e^{-n}.$$

How accurate is Stirling's formula?

6.9. A programmer claims that the birthday probability Q_n in problem 6.6 above seems to be approximately equal to

$$Q_n \approx \frac{n(n-1)}{730}$$

Do you agree?

6.10. How many different bridge hands can a bridge player obtain? How many ways can a bridge deck be dealt into four hands (North, West, South, and East)? (Hint: Use logarithms).

6.11. An accident assurance company finds that 0.001 of the population of Amsterdam auto owners drive their car into a canal each year. Assuming that the company has insured 10,000 Amsterdammers who own autos and who were selected at random, compute the probability that not more than 3 of the company's policyholders will drive into some canal in a given year:

$$\sum_{k=0}^{3} \binom{n}{k} p^k (1-p)^{n-k}, \quad \text{where } p = 10^{-3} \text{ and } n = 10^4.$$

6.12. KLM airline finds that 4 percent of the persons making reservations on the Amsterdam-to-London flight will not show up for the flight. If their policy is to sell to 75 persons reserved seats on a plane that has exactly 73 seats, then compute the probability that there will be a seat to London for every person who shows up:

$$\sum_{k=0}^{73} \binom{n}{k} p^k(1-p)^{n-k}, \quad \text{where } p = 0.96 \text{ and } n = 75.$$

(Hint: Use the binomial formula, $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$.)

6.13. Let Q_k be the probability that in a group of 500 people (chosen at random) exactly k will have birthday on April 25. Clearly

$$Q_k = \binom{n}{k} p^k(1-p)^{n-k}, \quad \text{where } n = 500 \text{ and } p = \frac{1}{365} .$$

The Poisson approximation to the binomial probability Q_k is

$$Q_k \approx e^{-\lambda} \frac{\lambda^k}{k!}, \quad \text{where } \lambda = np.$$

Compute and compare these formulas for $k = 0(1)n$.

6.14. Consider a Guilder-tossing situation with constant probability p $(0<p<1)$ for Queen Juliana's head. Let $x_i = 1$ (0) indicate that the Queen's head did (did not) occur on the i-th toss. Thus the probability (likelihood) for an <u>ordered</u> sample of size 100 is:

$$\text{Prob}\{(x_1,x_2,\ldots,x_{100})\} = p^{x_1}(1-p)^{1-x_1} \ldots p^{x_{100}}(1-p)^{1-x_{100}}$$

$$\text{for } x_i = 0 \text{ or } 1 \ (1\leq i\leq 100).$$

If $\sum_{i=1}^{100} x_i = 47$ was observed, compute the maximum likelihood estimate of p.

6.15. Same problem as 6.14, except now order is not important but you are interested in:

H_{100} = total number of heads in 100 tosses.

$$\text{Prob}\{H_{100} = h\} = \begin{cases} \binom{100}{h} p^h(1-p)^{100-h}, & \text{if } h = 0,1,\ldots,100 \\ 0, & \text{otherwise.} \end{cases}$$

Compute the maximum likelihood estimate of p when h = 47 is observed.

6.16. Two persons, "You" and "Me", have initial (prior) opinions about the parameter:

t = Average temperature in Centigrade at the exact North Pole.

These prior probabilities are:

You: t is Normal (mean -9, precision 1/36)

Me : t is Normal (mean 3, precision 1/4)

Remember that "x is Normal (mean μ, precision ρ)" means x has the probability density function

$$\frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} = \sqrt{\frac{\rho}{2\pi}}\, e^{-\frac{\rho}{2}(x-\mu)^2}$$

when the precision ρ equals $\frac{1}{\sigma^2}$, the reciprocal of the variance.

Four different normally distributed samples of size n are drawn to better determine t:

Sample	n	sample mean $\bar{x}$	sample precision $\frac{1}{s^2}$
1	1	0,1	1/4
2	4	0,06	1
3	9	0,001	9/4
4	16	0	4

It can be shown (by Bayes' theorem) that after such a sample is drawn, "You" must have a normally distributed posterior opinion with:

posterior mean = weighted mean of the datum value and the prior mean, weighted with their precisions.

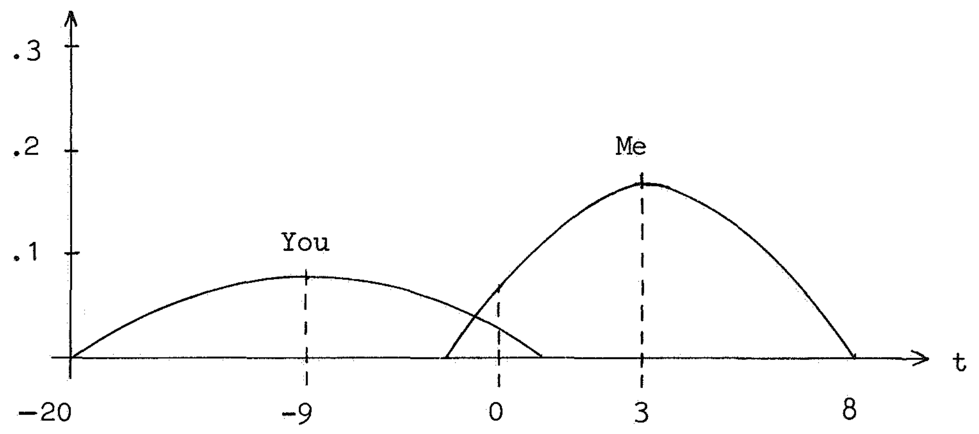
$$= \frac{(-9)(\frac{1}{36}) + \bar{x}(\frac{1}{s^2})}{(\frac{1}{36}) + (\frac{1}{s^2})} .$$

posterior precision = prior precision + datum precision

$$= (\frac{1}{36}) + (\frac{1}{s^2}).$$

Similarly for "Me".

Compute and plot the four posterior opinions (corresponding to each of the 4 samples) for both You and Me. For example, before the first sample is observed, the plot would be:



6.17. The least squares straight line, $y = mx + b$, which fits the data:

i	y_i	x_i
1	9.12	2000
2	20.44	4000
3	32.47	6000
4	46.15	8000
5	55.82	10000
6	70.40	12000

consists of that slope m and intercept b which minimize the quantity:

$$\sum_{i=1}^{10} [y_i - (mx_i+b)]^2.$$

Compute and plot this least squares line. How good does it fit the data?

6.18. A soccer star's picture is enclosed in each packet of cigarettes you buy. How many packets must you buy before you complete a set of ten pictures? Do a simulation first; then try to derive an analytic formula.

SOLUTIONS

6.3. Time $t = A/M + B$

Space $s = G + M$

Minimise $f \equiv ts = AC/M + A + BG + BM$ with respect to M.

$$f_M = B - AG/M^2 = 0 \Longrightarrow M^2 = AG/B.$$

6.5. Preliminary estimate: 999 pages needs

$$99 + 2 \times 90 + 3 \times 900 = 2889 \text{ pages.}$$

Thus, for y pages,

$$2889 + 4 \times (y-999) = 2989$$

$$\therefore y = 1024.$$

6.6. See ALGOL program and plotter output on following pages.

6.7. The largest integer less than $\frac{n_1 r}{k}$. In this case, 9999.

6.8. The percentage error decreases steadily and Stirling's approximation is remarkably accurate even for small n:

n	n!	Stirling's formula	% error
1	1	0.9221	8
2	2	1.919	4
5	120	118.019	2
10	3628800	3598600	0.8
100	*	*	0.08

6.9. Yes, for n = 0,1,...,20.

6.10. $\binom{52}{12} = 635{,}013{,}559{,}600.$

$$\binom{52}{13}\binom{39}{13}\binom{26}{13}\binom{13}{13} = \frac{52!}{(13!)^4} \approx 5.36 \times 10^{18}$$

6.11. 0.010

6.12. Compute $1 - \sum_{k=74}^{75} \binom{75}{k} 0.96^k \, 0.04^{75-k}$

6.13.

k	Qk	Poisson approximation
0	0.2537	0.2541
1	0.3484	0.3481
2	0.2388	0.2385
3	0.1089	0.1089
4	0.0372	0.0373
5	0.0101	0.0102
6	0.0023	0.0023

All errors are in the fourth decimal place.

6.14. 0.47

6.15. Same as 6.14, that is $h/100 = 0.47$.

6.16. See ALGOL program and plotter output on following pages.

6.17. $m = 0.006089$ and $b = 3.556$.

6.18.
```
integer procedure random;
comment Produces a pseudo-random digit from 0 to 9;
for i:= 0 step 1 until 9 do A[i]:= 0;
r:= 0; comment r = number of different cards so far;
for c:= 1, c+1 while r ≤ 9 do
begin i:= random;
      if A[i] = 0 then begin r:= r+1;
                             A[i]:= 1
                       end
end;
print(c); comment c = total number of cards;
```

You may also compute the waiting time, w, for each card. Then

$E[w_i] = 10/(11-i)$

$\mathrm{Var}[w_i] = 10(i-1)/(11-i)^2$.

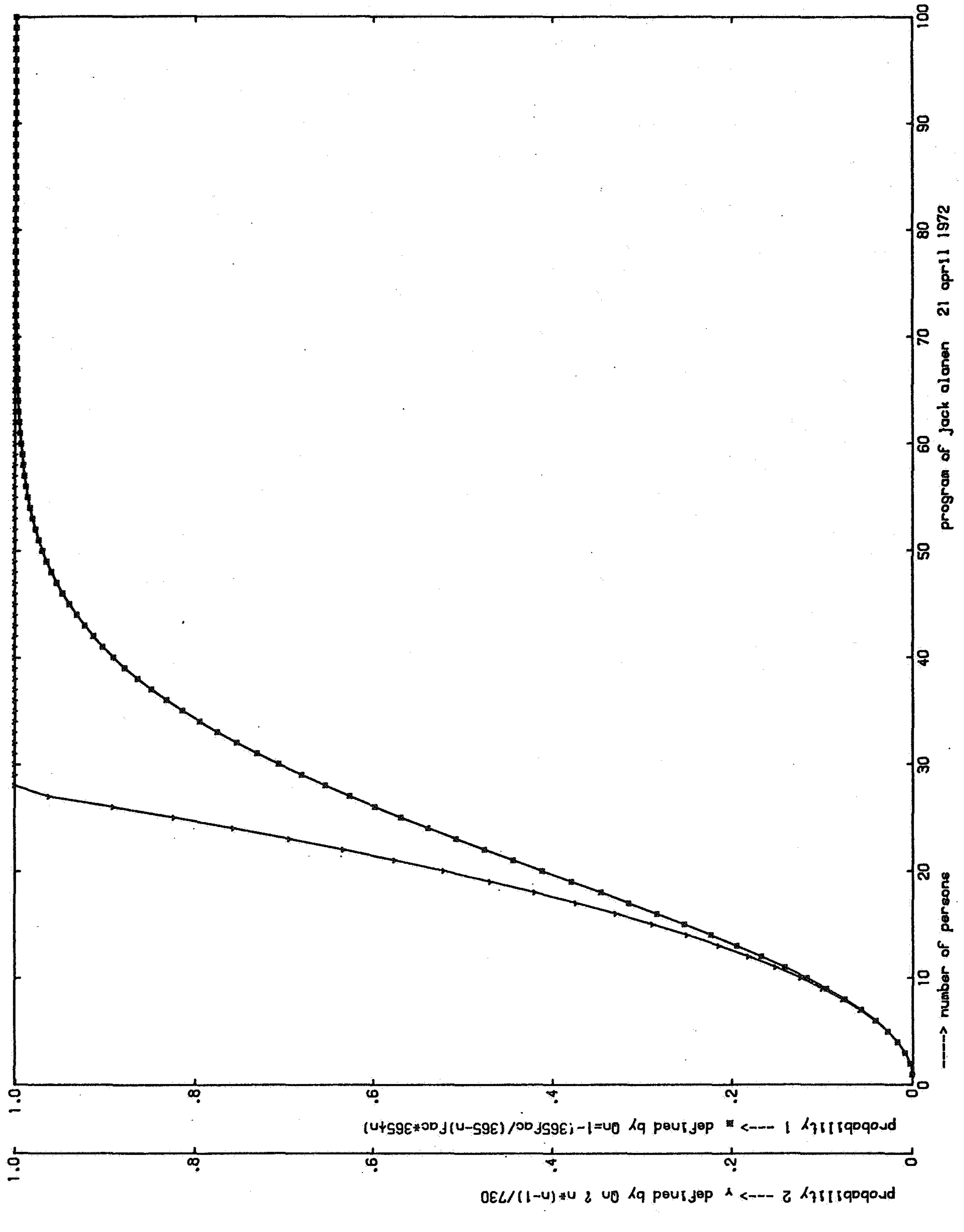
Total mean waiting time ≈ 29.3 (variance 125.7).
The analytic formula involves the negative binomial distribution.

```
D2296V.1,JACK ALANEN,P1000

'BEGIN'
'COMMENT'*****************************************************************************************************
           *                                                                                                *
           *    PROGRAM BY JACK ALANEN, MC EXT. 38, 21 APRIL 1972, TO SOLVE THE TWO PROBABILITY PROBLEMS:    *
           *    1. IN A ROOM CONTAINING  K  PERSONS LET  QK  BE THE PROBABILITY THAT THERE ARE TWO OR MORE PERSONS WITH THE SAME  *
           *         BIRTHDAY.  NOW                                                                         *
           *                                                                                                *
           *    QK = 1 - (1 - 1/365)*(1 - 2/365)* ... *(1 - (K-1)/365)   FOR  K=1,2,... 365,                *
           *                                                                                                *
           *    COMPUTE AND PLOT  QK  VERSUS  K  FOR  K=1(1)N.                                              *
           *    2. THE PROBABILITY IN PROBLEM 1 IS SUPPOSED TO APPROXIMATELY EQUAL  K*(K-1)/730.  DO YOU AGREE?  *
           *                                                                                                *
           **************************************************************************************************

           'COMMENT' BASIC STRUCTURE OF THIS PROGRAM IS:
                     READ IN INPUT (N).
                     PRINT HEADINGS.
                     'FOR' K:=1 'STEP' 1 'UNTIL' N 'DO' 'BEGIN'
                                                         COMPUTE QK.
                                                         PRINT K-TH LINE.
                                                        'END'.
                     PLOT PROBABILITIES.
                     STOP.;


'INTEGER' N;

           'COMMENT' BASIC PROCEDURE FOR PRINTER OUTPUT;
           'PROCEDURE' OUT(STRING,NAME); 'STRING' STRING;
           'BEGIN' NLCR; PRINTTEXT(STRING); PRINTTEXT(" = "); PRINT(NAME) 'END';


READ IN INPUT:       N:=READ; OUT("MAX NUMBER OF PEOPLE IN ROOM =N",N);
                     'IF' N<0 'THEN' N:=0 'ELSE' 'IF' N>365 'THEN' N:=365;

'BEGIN'   'REAL' 'ARRAY' QK[1:N]; 'REAL' TERM;

           'COMMENT' PARAMETERS TO PLOT PROCEDURE;
           'INTEGER' I,MARK,DELTAMARK,MODE,MAXX,MAXY,K;
           'REAL' XMIN,XMAX,DX,YMIN,YMAX,DY;


PRINT HEADINGS:

                     NLCR;
                     SPACE(3);
                     PRINTTEXT("K");
                     SPACE(2);
                     PRINTTEXT("QK=1-365FAC/((365-K)FAC*365**K)");
                     SPACE(2);
                     PRINTTEXT("QK=?=K*(K-1)/730");
                     SPACE(2);
                     PRINTTEXT("RELATIVE PERCENT ERROR");
                     NLCR;
```

```
COMPUTE QK: ;'COMMENT'   Q1=0=1-(1-(1-1)/365)
                         Q2=1/365=1-(1-1/365)
                         Q3=1-(1-1/365)*(1-2/365)
                         ...
                         QK=1-(1-1/365)*(1-2/365)*...*(1-(K-1)/365);


               TERM:=1;
               'FOR' K:=1 'STEP' 1 'UNTIL' N 'DO' 'BEGIN' TERM:=TERM*(1-(K-1)/365);  QK[K]:=1-TERM;
                                   PRINT K TH LINE:    ABSFIXT(3,0,K);
                                                       SPACE(3);
                                                       ABSFIXT(2,11,QK[K]);
                                                       SPACE(15);
                                                       ABSFIXT(2,11,K*(K-1)/730);
                                                       SPACE(3);
                                                       FIXT(2,11,100*(QK[K]-K*(K-1)/730)/QK[K]);
                                                       NLCR;
                                                'END';

PLOT PROBABILITIES:
               ;'COMMENT' PLOT  X[I] = I  VERSUS  Y[I] = QK[I]  FOR  I=1(1)N,  SEE PLOT PROCEDURE WRITEUP IN LR1,1, SECTION J,6.2.2,
                         PAGES 4-19 AND 4-20, APRIL 1971;

               MARK:=5;  'COMMENT' SYMBOL (*) NUMBER 139 IN TABLE 6.5.2;
               DELTAMARK:=1;
               MODE:=0066;
               XMIN:=0;
               XMAX:=100;
               DX:=10;
               MAXX:=3000;
               YMIN:=0.0;
               YMAX:=1.0;
               DY:=0.2;
               MAXY:=2500;


               PLOTPICTURE(I,QK[I],I,N,
                         MARK,DELTAMARK,MODE,
                         XMIN,XMAX,DX,MAXX,
                         "----> NUMBER OF PERSONS                                             PROGRAM OF JACK ALANEN   21 APRIL 1972",
                         YMIN,YMAX,DY,MAXY,
                         "PROBABILITY 1 ---> ⊥139⊥ DEFINED BY ⊥53⊥N=1-(365FAC/(365-N)FAC*365⊥69⊥N)",
                         PLOTLINE);

               'COMMENT' PLOT (ON TOP OF PREVIOUS GRAPH)  X[I]=I  VERSUS  Y[I] = I*(I-1)/730  FOR  I=1(1)N;

               MARK:=6;   'COMMENT' SYMBOL 140 (Y) IN TABLE 6.5.2;
               MODE:=2066;
               PLOTPICTURE(I,'IF' I*(I-1)/730 > 1 'THEN' 1 'ELSE' I*(I-1)/730,I,N,
                         MARK,DELTAMARK,MODE,
                         XMIN,XMAX,DX,MAXX,
                         "",
                         YMIN,YMAX,DY,MAXY,
                         "PROBABILITY 2 ---> ⊥140⊥ DEFINED BY ⊥53⊥N ⊥122⊥ N*(N-1)/730",
                         PLOTLINE);


'END'
'END'
```

```
MAX NUMBER OF PEOPLE IN ROOM =N =              +100
    K  QK=1-365FAC/((365-K)FAC*365**K)  GK=?=K*(K-1)/730  RELATIVE PERCENT ERROR
    1         .00000000000               .00000000000    +,1776646197514⏨+629
    2         .00273972603               .00273972603      -,00000000143
    3         .00820416588               .00821917808      -,18298262236
    4         .01635591247               .01643835616      -,50406052719
    5         .02713557370               .02739726027      -,96436721133
    6         .04046248365               .04109589041     -1,56541741016
    7         .05623570309               .05753424658     -2,30910864437
    8         .07433529235               .07671232877     -3,19772256608
    9         .09462383389               .09863013699     -4,23392599438
   10         .11694817771               .12328767123     -5,42077153257
   11         .14114137832               .15068493151     -6,76169759899
   12         .16702478883               .18082191781     -8,26052771595
   13         .19441027523               .21369863014     -9,92146885536
   14         .22310251200               .24931506849    -11,74910863001
   15         .25290131976               .28767123288    -13,74841109909
   16         .28360400525               .32876712329    -15,92471093532
   17         .31500766529               .37260273973    -18,28370569256
   18         .34691141787               .41917808219    -20,86144589639
   19         .37911852603               .46849315068    -23,57432267716
   20         .41143838358               .52054794521    -26,51905266536
   21         .44368833516               .57534246575    -29,67265987306
   22         .47569530766               .63287671233    -33,04245430551
   23         .50729723432               .69315068493    -36,63600706542
   24         .53834425791               .75616438356    -40.46112175367
   25         .56869970397               .82191780822    -44,52580201533
   26         .59824082013               .89041095890    -48,86821513614
   27         .62685928226               .96164383562    -53,40665167250
   28         .65446147234              1.03561643836    -58,23948118015
   29         .68096853748              1.11232876712    -63.34510420199
   30         .70631624272              1.19178082192    -68,73190078696
   31         .73045463373              1.27397260274    -74.40817593818
   32         .75334752705              1.35890410959    -80,38210246328
   33         .77497185417              1.44657534247    -86,66166192677
   34         .79531636462              1.53698630137    -93,25458439835
   35         .81438323887              1.63013698630    -,1001682879126⏨+  3
   36         .83218210638              1.72602739726    -,1074098185998⏨+  3
   37         .84873400822              1.82465753425    -,1149857925551⏨+  3
   38         .86406732106              1.92602739726    -,1229023405650⏨+  3
   39         .87821966437              2.03013698630    -,1311650568387⏨+  3
   40         .89123180962              2.13698630137    -,1397789528866⏨+  3
   41         .90315161148              2.24657534247    -,1487484176415⏨+  3
   42         .91403047156              2.35890410959    -,1580771848401⏨+  3
   43         .92392235566              2.47397260274    -,1677683085331⏨+  3
   44         .93288536855              2.59178082192    -,1778241474559⏨+  3
   45         .94097589947              2.71232876712    -,1882463587711⏨+  3
   46         .94825234337              2.83561643836    -,1990359014675⏨+  3
   47         .95477440283              2.96164383562    -,2101930494604⏨+  3
   48         .96059797266              3.09041095890    -,2217174141684⏨+  3
   49         .96577960932              3.22191780822    -,2336079760971⏨+  3
   50         .97037357958              3.35616438356    -,2458631246965⏨+  3
   51         .97443199333              3.49315068493    -,2584807055625⏨+  3
   52         .97800450933              3.63287671233    -,2714580738284⏨+  3
   53         .98113811348              3.77534246575    -,2847921524872⏨+  3
   54         .98387696276              3.92054794521    -,2984794942462⏨+  3
   55         .98626228882              4.06849315068    -,3125163454809⏨+  3
   56         .98833235486              4.21917808219    -,3268987108772⏨+  3
   57         .99012245934              4.37260273973    -,3410224173559⏨+  3
```

```
  58       .99166497939             4.52876712329     -.355683176013610+   3
  59       .99298944842             4.68767123287     -.372076640929110+   3
  60       .99412266087             4.84931506850     -.387798463851210+   3
  61       .99508879861             5.01369863014     -.403644343957510+   3
  62       .99590957490             5.18082191781     -.420210072119710+   3
  63       .99660438683             5.35068493151     -.436691569234010+   3
  64       .99719047897             5.52328767123     -.453684918450810+   3
  65       .99768310731             5.69863013698     -.471186391271210+   3
  66       .99809570464             5.87671232877     -.488792467640710+   3
  67       .99844004298             6.05753424657     -.506699850348310+   3
  68       .99872639125             6.24109589041     -.524905474118010+   3
  69       .99896366631             6.42739726028     -.543406509870710+   3
  70       .99915957596             6.61643835616     -.562200364719110+   3
  71       .99932075318             6.80821917808     -.581284678261710+   3
  72       .99945288064             7.00273972603     -.600657315783210+   3
  73       .99956080556             7.20000000000     -.620316358942510+   3
  74       .99964864445             7.40000000000     -.640260094496410+   3
  75       .99971987817             7.60273972603     -.660487001610010+   3
  76       .99977743745             7.80821917808     -.680995738208310+   3
  77       .99982377924             8.01643835616     -.701785126798810+   3
  78       .99986095458             8.22739726027     -.722854140126110+   3
  79       .99989066840             8.44109589040     -.744201886986410+   3
  80       .99991433195             8.65753424658     -.765827598420010+   3
  81       .99993310851             8.87671232877     -.787730614502010+   3
  82       .99994795292             9.09863013698     -.809910371874510+   3
  83       .99995964569             9.32328767123     -.832366392125410+   3
  84       .99996882215             9.55068493151     -.855098271062610+   3
  85       .99997599733             9.78082191781     -.878105668932210+   3
  86       .99998156699            10.01369863014     -.901388301585810+   3
  87       .99998592540            10.24931506849     -.924945932555010+   3
  88       .99998928017            10.48767123287     -.948778366016210+   3
  89       .99999186407            10.72876712329     -.972885440602010+   3
  90       .99999384236            10.97260273973     -.997267023968510+   3
  91       .99999536520            11.21917808219     -.102192300810910+   4
  92       .99999652073            11.46849315068     -.104685330528610+   4
  93       .99999739769            11.72054794521     -.107205784457510+   4
  94       .99999806075            11.97534246575     -.109753656890210+   4
  95       .99999856017            12.23287671233     -.112328943255910+   4
  96       .99999893492            12.49315068492     -.114931639911410+   4
  97       .99999921505            12.75616438357     -.117561743965010+   4
  98       .99999942365            13.02191780822     -.120219253133410+   4
  99       .99999957840            13.29041095890     -.122904165621510+   4
 100       .99999969275            13.56164383562     -.125616480024210+   4
```


probability 1 ---> * defined by Qn=1-(365fac/(365-n)fac*365↑n)
probability 2 ---> + defined by Qn ≈ n*(n-1)/730
----> number of persons
program of jack alanen 21 april 1972
0
.2
.4
.6
.8
1.0
0
10
20
30
40
50
60
70
80
90
100

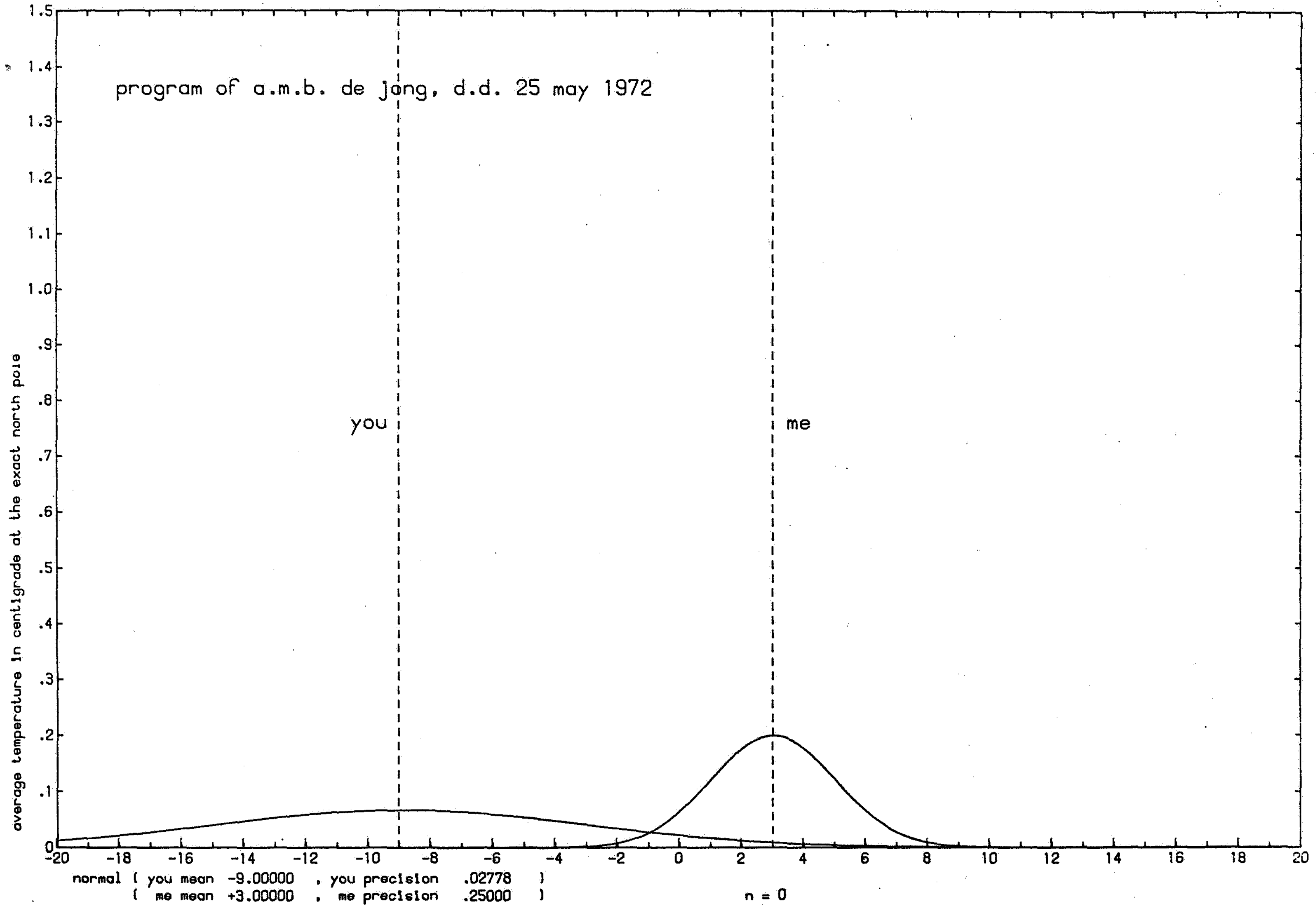
```
D2296V.014, DE JONG, T1000, R250, P1000

   'BEGIN' 'COMMENT' ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                     <PROBLEM 3 IN STATISTIC. THREE OF JACK ALANEN, >
                     <D.D. 4 MAY 1972                               >
                     <COMPUTE BY BENJAMIN DE JONG                   >
                     <THE PROBLEM:                                  >
                     <TWO PERSONS, 'YOU' AND 'ME' HAVE INITAL(PRIOR)>
                     <OPINIONS ABOUT THE PARAMETER:                 >
                     <T= AVERAGE TEMPERATURE IN CENTIGRADE AT THE   >
                     <EXACT NORTH POLE                              >
                     <THE TWO PRIOR PROBABILITIES ARE:              >
                     <YOU: T IS NORMAL(MEAN -9,PRECISION 1/36)      >
                     <ME: T IS NORMAL(MEAN  3,PRECISION 1/4  )      >
                     <THEN WE HAD FOUR TESTING SAMPLES              >
                     <AND WE CHANCHING AFTER EACH SAMPLE THE MEAN   >
                     <AND THE PRECISION (BY BAYES' THEOREM)         >
                     <FOR THE MEAN WE NEED THE FORMULA:             >
                     <(OLD MEAN*OLD PRECISION+SAMPLE MEAN*SAMPLE    >
                     <PRECISION)/(OLD PRECISION*SAMPLE PRECISION)   >
                     <FOR THE PRECISION WE NEED THE FORMULA:        >
                     <OLD PRECISION * SAMPLE PRECISION              >
                     <WE PLOT THESE FIVE OPINIONS WHICH NEEDING THE >
                     <FORMULA:PLOTPICTURE WHICH PROCEDURE IS DESCRI->
                     <BED IN LR 1.1 SECTION 4.6.2.9.                >
                     <FOR THE FUNCTION WE NEED:                     >
                     <DENSITY FUNCTION = SQRT(PRECISION/(2*PI))*    >
                     <EXP(-(PRECISION/2)*(X-MEAN) ** 2)             >
                     vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

    ;
    'REAL' YOUMEAN, YOUPRECISION, MEMEAN, MEPRECISION, SAMPLEMEAN,
    SAMPLEPRECISION, OLDMEAN, OLDPRECISION, PI,N;
    'INTEGER' NUMBER, X, Z, K;
    'REAL'  'ARRAY' NN,SAMPLEM,SAMPLEP[1:4];
    'BOOLEAN' CHECK;

    'COMMENT' AT FIRST WE READ THE PRIOR PROBABILITIES;

    YOUMEAN:= READ;
    YOUPRECISION:= READ;
    MEMEAN:= READ;
    MEPRECISION:= READ;
    PI:= 3.1415926535808;
    CHECK:= 'TRUE';

    'BEGIN'

       'PROCEDURE' OUTPUT(MEAN, PRECISION); 'VALUE' MEAN, PRECISION;
       'REAL' MEAN, PRECISION;
       'BEGIN'
          'IF'  Z  'DIV' 2 * 2 = Z 'THEN'
          'BEGIN' SPACE(1); ABSFIXT(2,0,N);
                  SPACE(3); FIXT(1,6,MEAN);
                  SPACE(2); FIXT(2,10,PRECISION);
                  SPACE(80-PRINTPOS); FIXT(1,6,MEAN - OLDMEAN);
                  SPACE(2); FIXT(2,10,PRECISION - OLDPRECISION);
                  CARRIAGE(0); Z:= Z + 1
   'END'
   'ELSE'
   'BEGIN' SPACE(37); FIXT(1,6,MEAN);
           SPACE(2); FIXT(2,10,PRECISION);
           SPACE(113 - PRINTPOS); FIXT(1,6.MEAN - OLDMEAN);
           SPACE(2); FIXT(2,10,PRECISION - OLDPRECISION);
           NLCR; Z:= Z + 1
   'END';
'END' OUTPUT;

'PROCEDURE' DRAW(X,Y,I); 'VALUE' X, Y, I; 'REAL' X, Y; 'INTEGER' I;

'BEGIN' 'COMMENT' FOR THE PROCEDURES NEED IN THIS PROCEDURE LOOKED
                  AT LR 1.1 SECTION 4.6.1.1, 4.6.1.3, 4.6.1.4,
                  4.6.2.4;

   'IF' I < 2 'THEN'
   'BEGIN' PLOT(YOUMEAN, 0, 2);
           PLOT(YOUMEAN,1.5,4);
           SHAPE(0, 42, 0);
           COORD(YOUMEAN-1.5,.75, 'TRUE');
           PLOTTEXT("YOU");
           PLOT(MEMEAN, 0, 2);
           PLOT(MEMEAN,1.5,4);
           COORD(MEMEAN+0.5,.75, 'TRUE');
           PLOTTEXT("ME");
           COORD(-18,1.35, 'TRUE');
           PLOTTEXT("PROGRAM OF A.M.B. DE JONG, D.D. 25 MAY 1972");
           SHAPE(0, 28, 0);
           COORD(-15, -.0625, 'TRUE');
           FIXPLOT(3,5,YOUMEAN);
           COORD(-15, -.09375, 'TRUE');
           FIXPLOT(3,5,MEMEAN);
           COORD(-7.5, -.0625, 'TRUE');
           ABSFIXPLOT(1,5,YOUPRECISION);
           COORD(-7.5, -.09375, 'TRUE');
           ABSFIXPLOT(1,5,MEPRECISION);
           'IF' K > 0 'THEN'
           'BEGIN' COORD(16,-.06250, 'TRUE');
              FIXPLOT(3,5,SAMPLEM[K]);
              COORD(16,-.09375, 'TRUE');
              FIXPLOT(3,5,SAMPLEP[K]);
              SHAPE(0,28,0);
              COORD(10,1.35,'TRUE');
              PLOTTEXT("CHANGING AFTER SAMPLE");
              SHAPE(0,28,0);
              COORD(16,1.35,'TRUE');
              ABSFIXPLOT(2,0,K);
           COORD( 4, -0.06250 , 'TRUE');
           ABSFIXPLOT(2,0,K);
           COORD(4,-.09375, 'TRUE');
           ABSFIXPLOT(2,0,N);
           'END';
   'END';
   PLOTCURVE(X, Y, I);
'END' DRAW;

'REAL' 'PROCEDURE' NORMALFUNCTION;
'BEGIN' 'REAL' HULPP,HULPM;
   'IF' CHECK 'THEN'
```

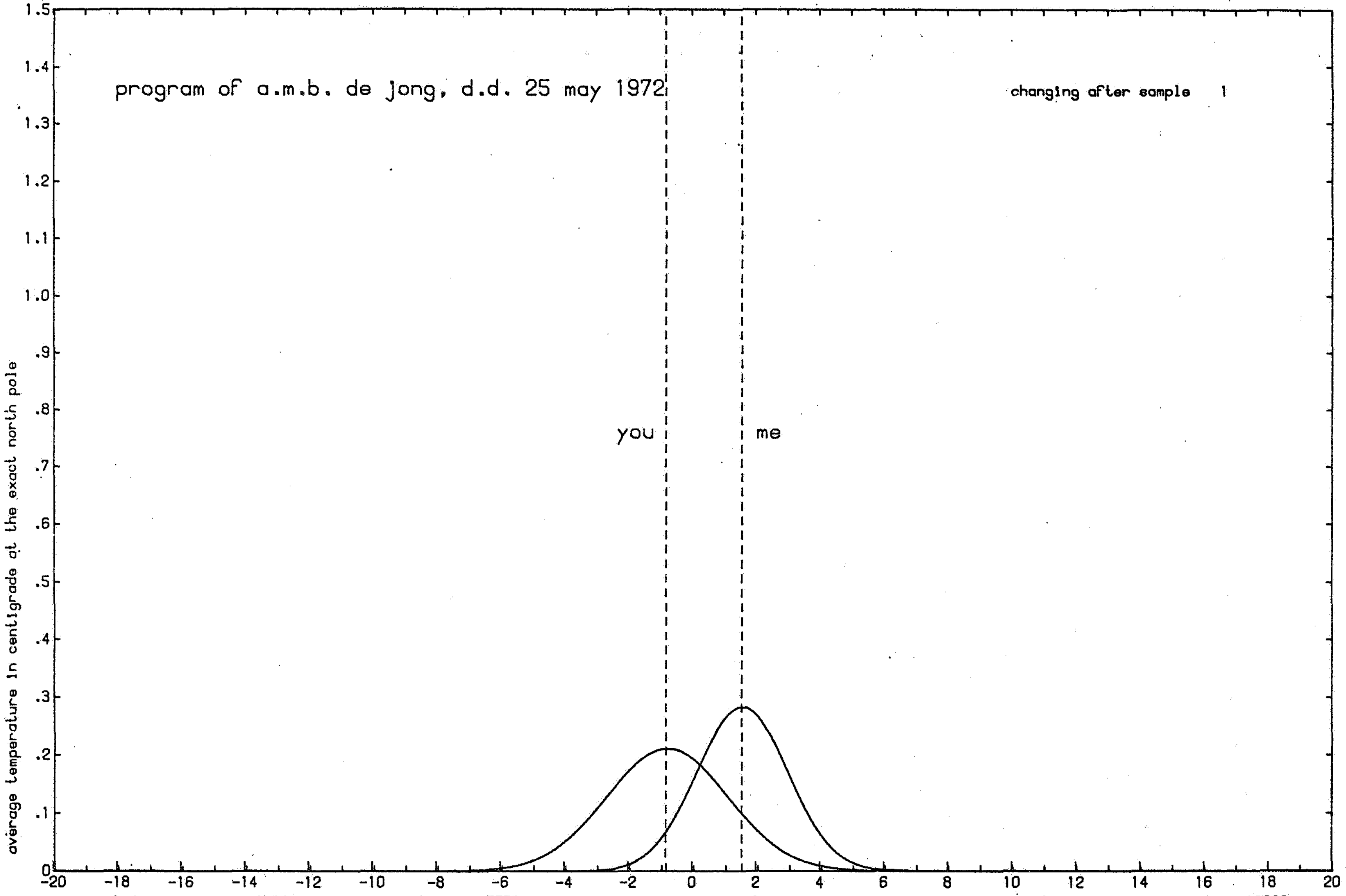
```
        'BEGIN' HULPP:= YOUPRECISION; HULPM:= YOUMEAN 'END'
        'ELSE'
        'BEGIN' HULPP:= MEPRECISION; HULPM:= MEMEAN 'END';
        NORMALFUNCTION:= SQRT(HULPP / (2*PI)) * EXP(-( HULPP / 2) * ( 1/40 * X -20.025 - HULPM) **2);
      'END' NORMALFUNCTION;

      'COMMENT' HEADING OF OUTPUT;

      SPACE(103); PRINTTEXT("CHANGING - TABLE"); NLCR;
      NLCR; SPACE(3); PRINTTEXT("N");
      SPACE(6); PRINTTEXT("YOUMEAN");
      SPACE(6); PRINTTEXT("YOUPRECISION");
      SPACE(5); PRINTTEXT("MEMEAN");
      SPACE(5); PRINTTEXT("MEPRECISION");
      SPACE(80 - PRINTPOS); PRINTTEXT("YOUMEAN");
      SPACE(6); PRINTTEXT("YOUPRECISION");
      SPACE(10); PRINTTEXT("MEMEAN");
      SPACE(5); PRINTTEXT("MEPRECISION");
      NLCR;
      SPACE(3); PRINTTEXT("0");
      SPACE(4); FIXT(1,6,YOUMEAN);
      SPACE(2); FIXT(2,10,YOUPRECISION);
      SPACE(2); FIXT(1,6,MEMEAN);
      SPACE(2); FIXT(2,10,MEPRECISION);
      NLCR;
      Z:= 2;

      'COMMENT' WE PLOT THE FIRST PROBABILITY WITH PROCEDURE PLOTPICTURE;

      PLOTPICTURE(1/40*X-20.025, NORMALFUNCTION,
       X ,1601, 1,0 , 0066,-20, 20, 1, 3600,
      "NORMAL ( YOU MEAN              , YOU PRECISION           )
        ( ME MEAN              , ME PRECISION             )                        N = 0",
      0,1.5,0.1,2400,
      "AVERAGE TEMPERATURE IN CENTIGRADE AT THE EXACT NORTH POLE",
      DRAW);
      CHECK:= ¬ CHECK;
      PLOTPICTURE(1/40*X-20.025, NORMALFUNCTION,
       X ,1601, 1, 0, 2077, -20, 20, 1, 3600,
      "",
      0,1.5,0.1,2400,
      "",
      PLOTCURVE);
      CHECK:= ¬ CHECK;

      'COMMENT' WE READ THE TEST SAMPLES,
                AT FIRST HOW MUCH SAMPLES
                THEN FOR EACH SAMPLE:
                FIRST : HOW MUCH OBSERVANCES,
                SECOND: SAMPLE MEAN,
                THIRD : SAMPLE PRECISION;

      NUMBER:= READ;
      'FOR' K:= 1 'STEP' 1 'UNTIL' NUMBER 'DO'
      'BEGIN' NN[K]:= N:= READ;
              SAMPLEM[K]:= SAMPLEMEAN:= READ;
              SAMPLEP[K]:= SAMPLEPRECISION:= READ;
        OLDMEAN:= YOUMEAN;
        OLDPRECISION:= YOUPRECISION;
        YOUMEAN:= (YOUMEAN * YOUPRECISION + SAMPLEMEAN * SAMPLEPRECISION ) / (YOUPRECISION + SAMPLEPRECISION);
      YOUPRECISION:= YOUPRECISION + SAMPLEPRECISION;

     'BEGIN' 'COMMENT' WE PRINT THIS PART ;'END';

      OUTPUT(YOUMEAN, YOUPRECISION);
      OLDMEAN:= MEMEAN;
      OLDPRECISION:= MEPRECISION;
      MEMEAN:= (MEMEAN * MEPRECISION + SAMPLEMEAN * SAMPLEPRECISION)/ (MEPRECISION + SAMPLEPRECISION);
      MEPRECISION:= MEPRECISION + SAMPLEPRECISION;

      'BEGIN' 'COMMENT' WE PRINT THIS PART; 'END';

      OUTPUT(MEMEAN, MEPRECISION);

      'BEGIN' 'COMMENT' WE PLOT CHANGING MEAN AND PRECISION OF YOU AND ME; 'END';

      PLOTPICTURE(1/40*X-20.025, NORMALFUNCTION,
       X ,1601, 1, 0, 0066, -20, 20, 1, 3600,
      "NORMAL ( YOU MEAN          , YOU PRECISION          )                    SAMPLE =                                  SAMPLE MEAN
=
     ( ·ME MEAN            ,  ME PRECISION            )                    N =                                  SAMPLE PRECISION = ",
      0,1.5,0.1,2400,
      "AVERAGE TEMPERATURE IN CENTIGRADE AT THE EXACT NORTH POLE",
      DRAW);
      CHECK:= ¬ CHECK;
      PLOTPICTURE(1/40*X-20.025, NORMALFUNCTION,
       X ,1601, 1, 0, 2077, -20, 20, 1, 3600,
      "",
      0,1.5,0.1,2400,
      "",
      PLOTCURVE);
      CHECK:= ¬ CHECK;
   'END'
'END';

'COMMENT' PRINTING OF TESTRESULTS;

CARRIAGE(12);
'FOR' NUMBER:= 1 'STEP' 1 'UNTIL' 48 'DO'
'BEGIN' PRINTTEXT("_); SPACE(1) 'END'; NLCR;
PRINTTEXT("|); SPACE(2); PRINTTEXT("SAMPLE"); SPACE(1);
PRINTTEXT("|); SPACE(3); PRINTTEXT("N"); SPACE(1);
PRINTTEXT("|); SPACE(2); PRINTTEXT("SAMPLE MEAN"); SPACE(1);
PRINTTEXT("|); SPACE(2); PRINTTEXT("SAMPLE PRECISION"); SPACE(1);
PRINTTEXT("|); NLCR;
'FOR' NUMBER:= 1 'STEP' 1 'UNTIL' 48 'DO'
'BEGIN' PRINTTEXT("_); SPACE(1); 'END'; NLCR;
'FOR' NUMBER:= 1 'STEP' 1 'UNTIL' 4 'DO'
'BEGIN' PRINTTEXT("|); SPACE(1);
   ABSFIXT(3,0,NUMBER); SPACE(3);
   PRINTTEXT("|); SPACE(1);
   ABSFIXT(2,0,NN[NUMBER]);
   PRINTTEXT("|); SPACE(3);
   FIXT(1,4,SAMPLEM[NUMBER]); SPACE(3);
   PRINTTEXT("|); SPACE(5);
   FIXT(2,4,SAMPLEP[NUMBER]); SPACE(5);
   PRINTTEXT("|); NLCR
'END';
'FOR' NUMBER:= 1 'STEP' 1 'UNTIL' 48 'DO'
'BEGIN' PRINTTEXT("_); SPACE(1) 'END'; NLCR;
```

```
      'FOR' NUMBER:= 1 'STEP' 1 'UNTIL' 48 'DO'
      'BEGIN' PRINTTEXT("_"); SPACE(1) 'END'; NLCR;

    'END';
```

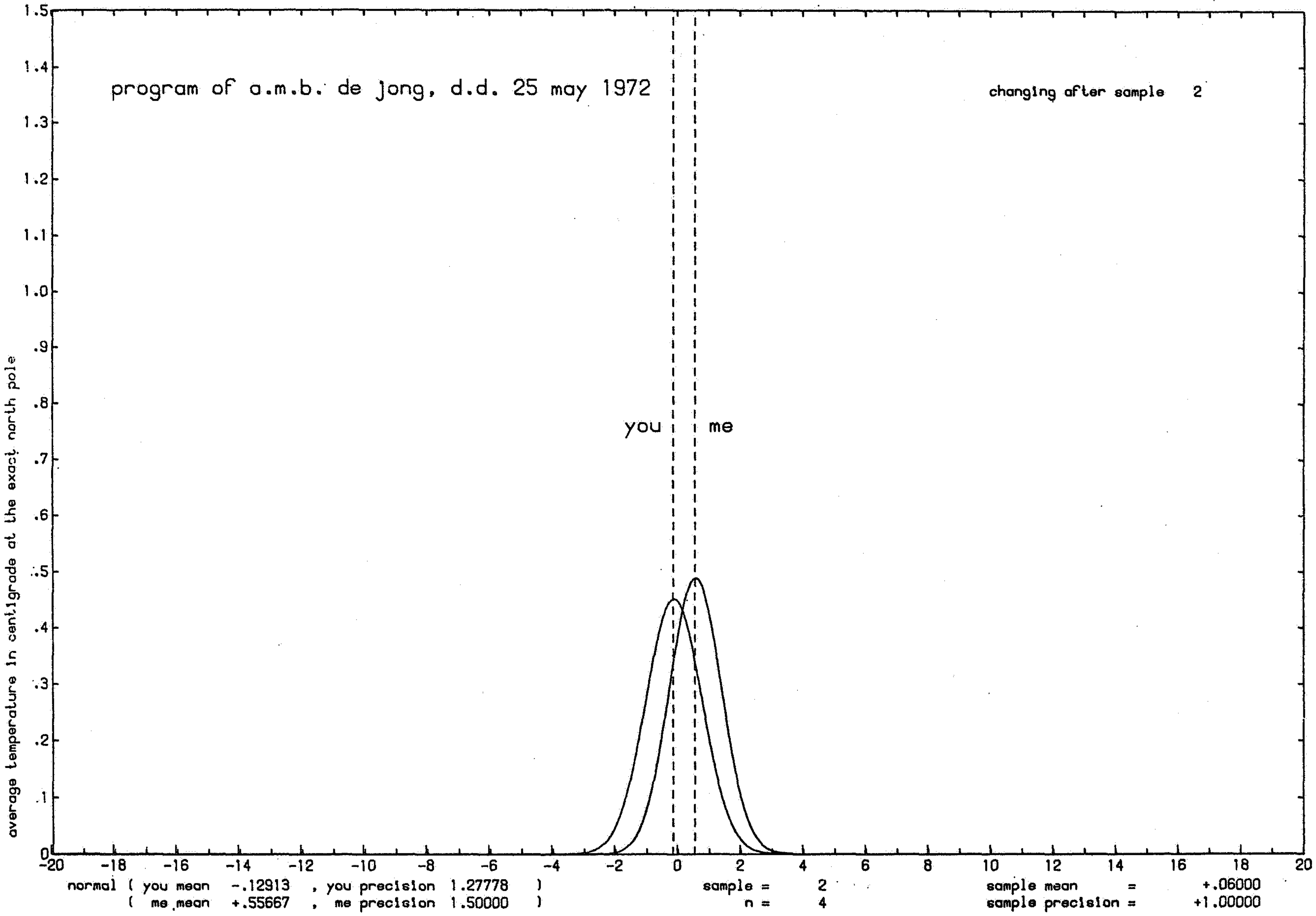
```
                                                                                                    CHANGING - TABLE

   N      YOUMEAN       YOUPRECISION      MEMEAN       MEPRECISION               YOUMEAN      YOUPRECISION           MEMEAN        MEPRECISION
   0    -9.000000     +.0277777778    +3.000000     +.2500000000
   1     -.810000     +.2777777778    +1.550000     +.5000000000               +8.190000     +.2500000000         -1.450000      +.2500000000
   4     -.129130    +1.2777777778     +.556667    +1.5000000000                +.680870    +1.0000000000          -.993333     +1.0000000000
   9     -.046134    +3.5277777778     +.223267    +3.7500000000                +.082997    +2.2500000000          -.333400     +2.2500000000
  16     -.021620    +7.5277777778     +.108032    +7.7500000000                +.024514    +4.0000000000          -.115234     +4.0000000000
```

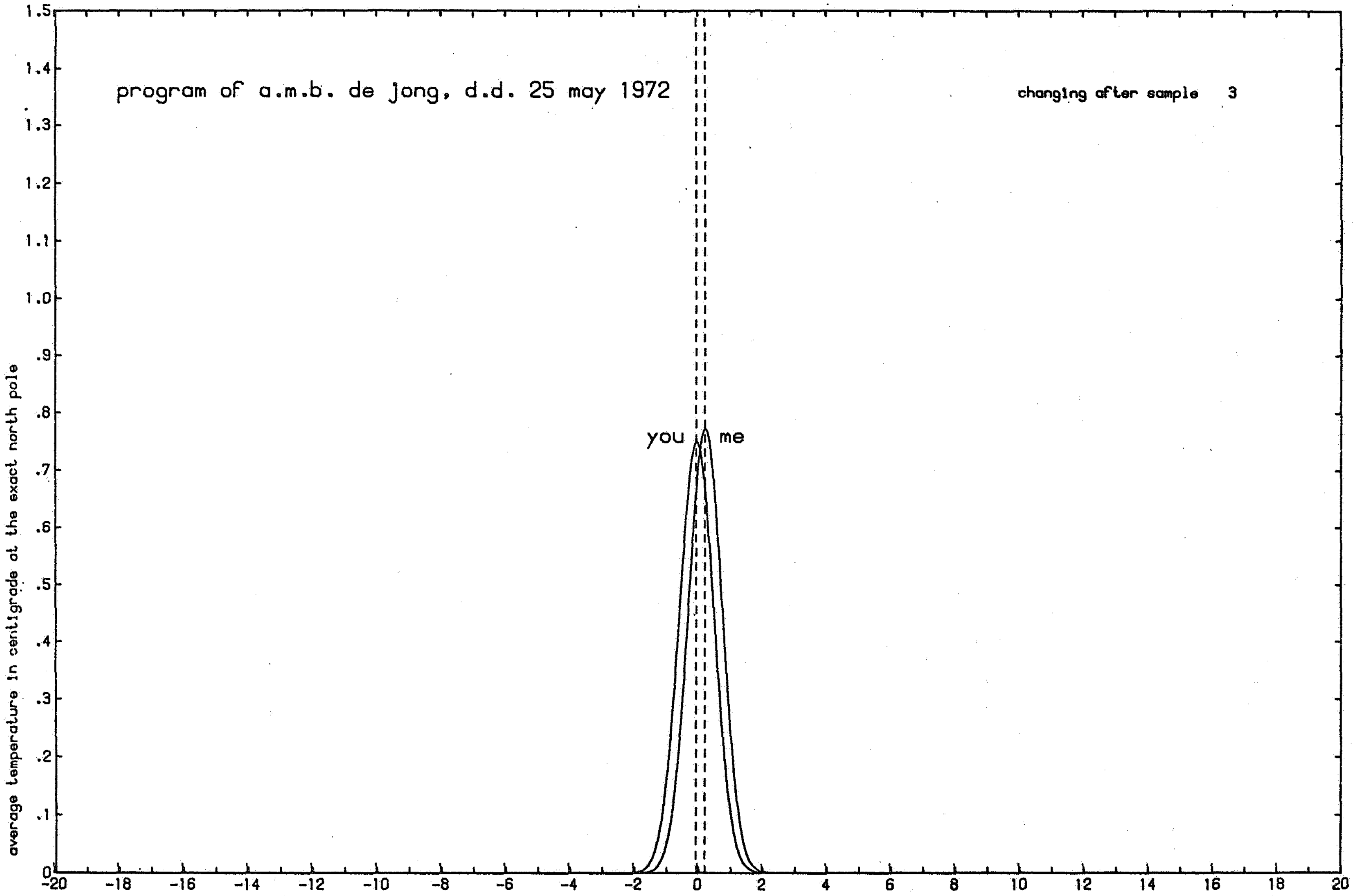
```
----------------------------------------------------
| SAMPLE |  N | SAMPLE MEAN | SAMPLE PRECISION |
----------------------------------------------------
|   1    |  1 |   +.1000    |      +.2500      |
|   2    |  4 |   +.0600    |     +1.0000      |
|   3    |  9 |   +.0010    |     +2.2500      |
|   4    | 16 |   +.0000    |     +4.0000      |
----------------------------------------------------
----------------------------------------------------
```


program of a.m.b. de jong, d.d. 25 may 1972
you
me
average temperature in centigrade at the exact north pole
normal (you mean -9.00000 , you precision .02778)
(me mean +3.00000 , me precision .25000)
n = 0

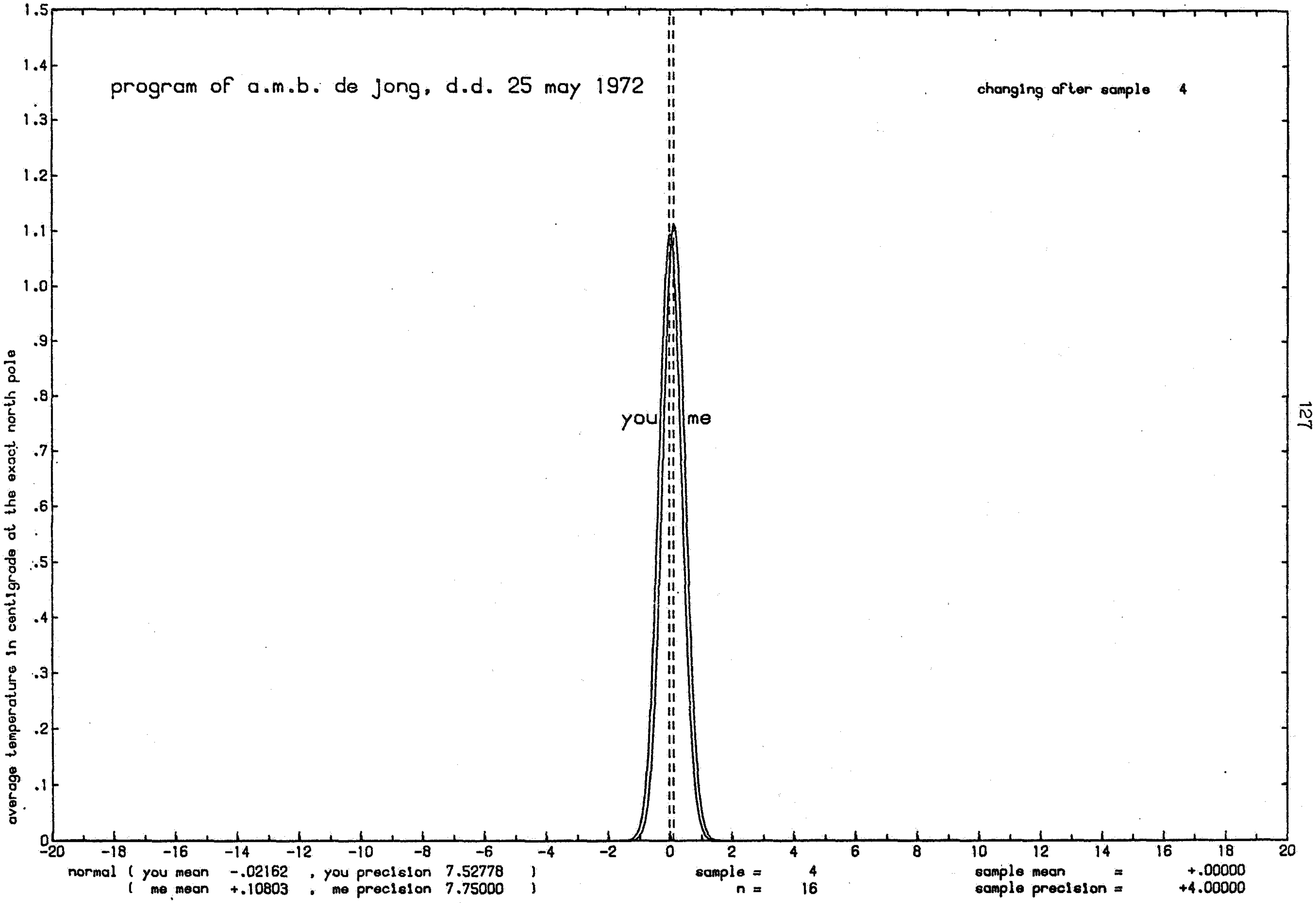


```
normal ( you mean   -.81000   , you precision   .27778   )            sample =     1                sample mean       =      +.10000
       (  me mean  +1.55000   ,  me precision   .50000   )                 n =     1                sample precision =       +.25000
```


program of a.m.b. de jong, d.d. 25 may 1972
changing after sample 2
you
me
average temperature in centigrade at the exact north pole
1.5
1.4
1.3
1.2
1.1
1.0
.9
.8
.7
.6
.5
.4
.3
.2
.1
0
-20 -18 -16 -14 -12 -10 -8 -6 -4 -2 0 2 4 6 8 10 12 14 16 18 20
normal (you mean -.12913 , you precision 1.27778)
(me mean +.55667 , me precision 1.50000)
sample = 2
n = 4
sample mean = +.06000
sample precision = +1.00000



```
normal ( you mean  -.04613  , you precision  3.52778   )            sample =     3                     sample mean      =      +.00100
       (  me mean  +.22327  ,  me precision  3.75000   )                 n =     9                     sample precision =     +2.25000
```



```
normal ( you mean   -.02162   , you precision  7.52778   )          sample =    4          sample mean      =      +.00000
       (  me mean   +.10803   ,  me precision  7.75000   )               n =   16          sample precision =     +4.00000
```

References

Alanen, J.D., 1972. Empirical study of aliquot series. Ph.D. dissertation, Yale University. Also MR 133/72, Mathematical Center report.

Anonymous, 1971. Computerworld 5 (52) 11.

Bauer, F.L., 1972. Software engineering. Lecture notes for Advanced Course on Software Engineering, Munich, Germany.

Beizer, B., 1971. On the ABM. Datamation 17 (17) 79-80.

Bellman, R., K.L. Cooke, and J.A. Lockett, 1970. Algorithms, Graphs, and Computers. Academic Press.

Berkeley, E.C., 1971. Common sense, elementary and advanced. Notebook published by Computers and Automation.

Bono, E. de, 1971. Practical Thinking, Johnathan Cape Ltd., 1-198.

Brent, R.P., 1970. Algorithms for matrix multiplication. Stanford University Report CS-70-157.

Buckley, F.J., 1971. Verification of software programs. Computers and Automation 20 (2) 23-24.

Buxton, J.N., and B. Randell, 1970. Software Engineering Techniques NATO Science Committee Report 1-164.

Day, A.C., 1970. The use of symbol state tables. The Computer Journal 13 (4) 332-339.

Dijkstra, E.W., 1968. Go to considered harmful. CACM 11 (3) 147.

Dijkstra, E.W., 1970. Notes on structured programming. Mathematics Dept., Tech. U. Eindhoven.

Dijkstra, E.W., 1971. On a methodology of design. In MC-25 Informatica Symposium Tract 37, Mathematical Center (Amsterdam).

Ellis, L.E., G. Goldstein (editor), and J.D. Tinsley, 1971. Computers and the Teaching of Numerical Mathematics in the Upper Secondary School. G. Bell and Sons.

Floyd, R.W., 1967. Assigning meanings to programs. Proc. Symp. Appl. Math. AMS 19 19-32.

Forsythe, G.E., 1970. Pitfalls in computation, or why a math book isn't enough. Stanford University Technical Report CS-147. Also in The American Mathematical Monthly 77 (9) 931-956.

Furnival, G.M., 1971. All possible regressions with less computation. Technometrics 13 (2) 403-408.

Gardner, M., 1968. Mathematical games: a short treatise on the useless elegance of perfect numbers and amicable pairs. Scientific American (March) 121-124.

Gentleman, W.M., and J.F. Traub, 1968. The Bell Laboratories numerical mathematics library project. Proc. ACM 23rd Nat. Conf. 485-490.

George, F.J., 1972. The urgent need to rethink the use of the computer in industry. Computers and Automation 20 (12) 19-24.

Glas, R.L., 1971. I believe in an anti-ballistic system. Computers and Automation 20 (9) 33.

Golomb, S.W., and L.D. Baumert, 1965. Backtrack programming. Journal ACM 12 (4) 516-524.

Goos, G., 1972. Documentation. Lecture notes for Munich course.

Gruenberger, F., and G. Jafferay, 1965. Problems for Computer Solution. Wiley.

Gruenberger, F., 1968. Program testing and validating. Datamation (July) 39-47.

Hall, M. and D.E. Knuth, 1965. Combinatorial analysis and computers. American Mathematical Monthly 72 (2) part II 21-28.

Hamming, R.W., 1971. Introduction to Applied Numerical Analysis. McGraw-Hill.

Hemmerle, W.J., 1967. Statistical Computations on a Digital Computer. Blaisdell Publishing Company.

Hill, I.D., 1972. Wouldn't it be nice if we could write programs in English - or would it? The Computer Bulletin 16 (6) 306-312.

Ignalls, D.H., 1971. FETE-A FORTRAN execution time estimator. Stanford University Report CS-71-204.

Kahan, W., 1971. A survey of error analysis. IFIP Congress 71, Ljubljana, Yugoslavia.

Knuth, D.E., 1968. The Art of Computer Programming: Fundamental Algorithms 1 Addison-Wesley.

Knuth, D.E., 1969. The Art of Computer Programming: Seminumerical Algorithms. 2 Addison-Wesley.

Knuth, D.E., 1971. Mathematical analysis of algorithms. Stanford University Report CS-71-206.

Knuth, D.E., 1972. The Art of Computer Programming: Sorting and Searching. 3 Addison-Wesley.

London, R.L., 1970. Proving programs correct: some techniques and examples. BIT 10 (2) 168-182.

London, R.L., 1970. Proof of algorithms: A new kind of certification (Certification of Algorithm 245 TREESORT 3). CACM 13 (6) 371-373.

London, R.L., 1970. Bibliography on proving the correctness of computer programs. Machine Intelligence 5, Meltzer, B. and D. Michie (eds.), Edinburgh University Press 569-580.

McCracken, D.D., 1971. Anti-ABM essay contest announced. Computers and Automation 20 (3) 33-34.

Moler, C.B., 1971. Matrix computations with FORTRAN and paging. Stanford University Report CS-71-196.

Naur, P., 1966. Proof of algorithms by general snapshots. BIT 6 (4) 310-316.

Naur, P., and B. Randell, 1969. Software Engineering NATO Science Committee Report 1-231.

Newman, T.G., and P.L. Odell, 1971. The Generation of Random Variates. Griffin.

Ord-Smith, R.J., 1971. Generation of permutation sequences: Part 2. The Computer Journal 14 (2) 136-139.

Poel, W.L. van der, 1972. A comparative study of some higher programming languages. Lecture notes at ADVANCED COURSE IN PROGRAMMING LANGUAGES AND DATA STRUCTURES, Amsterdam (12-23 June 1972).

Polya, G., 1957. How to Solve It. Doubleday.

Poole, P.C., 1972. Debugging and testing. Lecture notes of Advanced Course on Software Engineering, Technical University of Munich, Feb. 21 - March 4, 1-40.

Ralston, A., and H.S. Wilf, 1960. Mathematical Methods for Digital Computers. 1 Wiley.

Ralston, A., and H.S. Wilf, 1967. Mathematical Methods for Digital Computers. 2 Wiley.

Ralston, A., 1971. Introduction to Programming and Computer Science. McGraw-Hill. 207 (Program structure, preparation, and testing).

Redish, K.A., 1971. Comment on London's certification of Algorithm 245. CACM 14 (1) 50-51.

Rice, J.R., 1968. The go to statement reconsidered. CACM 11 (8) 538.

Rustin, R., 1971. Debugging techniques in large systems. Prentice-Hall.

Sammet, J.E., 1971. Problems in, and a pragmatic approach to, programming language measurement. Proceedings of the Fall Joint Computer Conference (AFIPS) 39 243-251.

Smith, J.M., 1972. Proof and validation of program correctness. The Computer Journal 15 (2) 130-131.

Sobieraj, R.A., 1971. Rears its ugly warhead. Datamation 17 (24) 21.

Sterling, T.D., and S.V. Pollack, 1968. Introduction to Statistical Data Processing. Prentice-Hall.

Tou, J.T., 1970. Software Engineering. 1 Academic Press.

Turski, W.M., 1971. Efficient production of large programs. Proceedings of International Workshop, Computation Centre, Polish Academy of Sciences.

Wegner, P., 1968. Programming Languages, Information Structures, and Machine Organization. McGraw-Hill.

Wells, M.B., 1971. Elements of Combinatorial Computing. Pergamon Press.

Winograd, S., 1970. On the number of multiplications necessary to compute certain functions. Communications on Pure and Applied Mathematics 23 (2) 165-179.

Wirth, N., 1971. Program development by stepwise refinement. Comm. ACM 14 (4) 221-227.

Wulf, W.A., 1971. Programming without the goto. IFIP Congress 71, booklet TA3, 84.

Youngs, E.A. and E.M. Cramer, 1971. Some results relevant to choice of sum and sum-of-product algorithms. Technometrics 13 (3) 657-665.