HUMAN FACTORS, 1974, 16(3), 258–277

An Exploratory Study of Computer Program Debugging[1]

JOHN D. GOULD[2] *and* PAUL DRONGOWSKI, *Carnegie-Mellon University, Pittsburgh, Pa.*

This experiment represents a new approach to the study of the psychology of programming, and demonstrates the feasibility of studying an isolated part of the programming process in the laboratory. Thirty experienced FORTRAN programmers debugged 12 one-page FORTRAN listings, each of which was syntactically correct but contained one non-syntactic error (bug). Three classes of bugs (Array bugs, Iteration bugs, and bugs in Assignment Statements) in each of four different programs were debugged. The programmers were divided into five groups, based upon the information, or debugging "aids", given them. Key results were that debug times were short (median = 6 min.). The aids groups did not debug faster than the control group; programmers adopted their debugging strategies based upon the information available to them. The results suggest that programmers often identify the intended state of a program before they find the bug. Assignment bugs were more difficult to find than Array and Iteration bugs, probably because the latter could be detected from a high-level understanding of the programming language itself. Debugging was at least twice as efficient the second time programmers debugged a program (though with a different bug in it). A simple hierarchical description of debugging was suggested, and some possible "principles" of debugging were identified.

INTRODUCTION

Studying computer programming requires pioneering and exploratory efforts because there is little useful, relevant data in either the psychological or computer science literature. Most early psychological work on programming was aimed at developing selection tests for programmers, but these results have been largely unrewarding (e.g., Mayer and Stalnaker, 1968). It is now as important to understand programmer performance as it is to understand system performance when attempting to improve the productivity of computing systems. Recent economic trends toward increased people costs and decreased system costs, the relative imbalance of knowledge about these two factors, and the repeated finding that productivity differences among programmers are much greater than productivity differences caused by the programming system studied (Erickson, 1966; Grant and Sackman, 1967; Schatzoff, Tsao, and Wiig, 1967) support this conclusion.

[1] This research was conducted during 1971-1972 while the first author was visiting associate professor of psychology at Carnegie-Mellon University. It was partially supported by the National Institute of Mental Health Grant MH-07722 to Professor Lee Gregg. We thank Stephen J. Boies and Lance A. Miller for many helpful suggestions.

[2] Present address from which reprints may be obtained:

IBM Research Center
P.O. Box 218
Yorktown Heights, New York 10598

The main lines of evidence on programmer performance come from normative studies of user characteristics (Boies and Gould, 1974; Bryan, 1967; Carbonell, Elkind, and Nickerson, 1968; Hunt, Diehr, and Garnatz, 1970; Knuth, 1971; Moulton and Muller, 1966; Nagy and Pennebaker, 1971; Neisser, 1964; Scherr, 1967; Walter and Wallace, 1967; Youngs, 1969; *cf.*, Gould, Doherty, and Boies, 1971, for a fuller bibliography), which have been useful in identifying behavioral characteristics and bottlenecks in programming. And, secondly, evidence comes from experimental studies comparing the performance of programmers on two different systems, usually on-line systems *vs.* off-line systems (Adams and Cohen 1969; Erickson, 1966; Gold, 1967; Grant and Sackman, 1967; Sackman, 1970; Schatzoff, Tsao, and Wiig, 1967; Smith, 1967).

The present experiment attempts to isolate one stage of the programming process (Figure 1), namely debugging, and study it under controlled laboratory conditions. It is important to understand debugging (*i.e.*, finding errors or bugs in computer programs) because programmers spend three times longer debugging their programs than initially coding them (Rubey, 1968); and debugging represents about 25–50% of the cost of large software projects (Boehm, 1973; Delaney, 1966).

Program bugs can be divided into (a) syntactic bugs (*i.e.*, those that a compiler or assembler detects); and (b) non-syntactic bugs (*i.e.*, the conceptual, logical, semantic, or even clerical bugs that do not prevent a program from compiling and do not lead to any error messages, but do prevent the program from running as intended). Boies and Gould (1974) found that only about one-sixth of all FORTRAN, PL/I, and Assembly language programs written in a research laboratory contained syntactic errors. This, perhaps surprising, result, which includes "first submissions" as well as later versions of programs, is not limited to just one environment (Moulton and Muller, 1967). Based upon these results, and the work of Youngs (1969) on how experienced programmers eliminate bugs, Boies and Gould (1974) concluded that syntactic bugs did not represent a bottleneck in the programming process, and that research should therefore try to understand how non-syntactic errors are debugged.

The present study is an initial attempt to study one piece of this problem. Experienced programmers were asked to debug 12 1-page FORTRAN listings written by others. Each listing contained one non-syntactic bug. As elaborated below, programmers in real life frequently debug programs written by others.

One variable studied was the value of three classes of information or aids. This provides a vehicle for grossly assessing how strategies in diagnostic problem-solving by highly trained subjects are affected by the information available to them. One group of programmers was given a listing only (control group). It was assumed that this group would serve as a control or baseline with which the performance of other groups could be compared. A second group was given a listing, a printout of some example input data, and a printout of the resulting output of the program (I/0 group). A third group was given the same as the second group, plus the output that would result if the program ran correctly (I/0 + Correct Output group). The assumption here was that programmers would compare the two outputs, using a means-end problem-solving strategy (*cf.*, Newell and Simon, 1972, p. 415) to identify differences, and then use these resulting differences

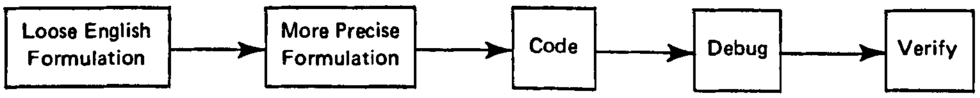

Figure 1. Stages in the programming process (without feedback loops).

to find the bug. A fourth group was given a listing and was told the class of bug that the program contained (Class of Bug group). The assumption here was that programmers would use this information to reduce their search for the bug by focussing selectively upon certain lines of the program.

A second variable studied was the class of bug. Two classes, array (allocation) bugs and iteration bugs, were chosen because conversations with consulting programmers indicated these were the most difficult to locate. The third class, assignment statement bugs, was chosen because assignment statements contain a high proportion of both syntactic and nonsyntactic errors (Moulton and Miller, 1967; Youngs, 1969).

A third variable studied was the effect of debugging the same program more than once. Each subject debugged each of four programs three times (with a different bug each time).

METHOD

Thirty volunteer, experienced FORTRAN programmers were each paid $7.50 to serve in this experiment. All were connected with Carnegie-Mellon University, most as students. At least half had served as consulting programmers in industry or at the university. Most of the remainder were undergraduate computer science majors having taken at least three computer science courses.

The four programs were modifications of FORTRAN statistical data analysis programs contained in the IBM Scientific Subroutine package (see Appendix I), and called MOME, TALL, TABI, and CORR. Fortran was chosen because of its wide usage and its familiarity to the experimenters. After modification, the programs contained 29, 37, 59, and 58 executable statements, respectively. Together with a 10-line description of the program's function, a Dimension statement, and Comment statements, the listings of each program were one to two pages long. The distributions of statement types in these four programs were about what Knuth (1971) found in his empirical analyses of submitted programs at Lockheed and at Stanford, except that in the present programs the Read, Write, and Format statements were removed. Seventy-one percent (*vs.* Knuth's 69% of the statements were Assignment, IF, or GO TO statements. There were fewer GO TO's than in Knuth's findings (4% *vs.* 6%), but there were more DO statements (19% *vs.* 4.5%), suggesting that the data flow in the present programs consisted of relatively more loops and less jumping around.

The use of these test programs provides the advantages of experimental control in that each subject received the same programs to debug. To some readers, debugging programs written by others might seem to have little relation to the real world. However, programmers in production environments frequently debug other people's programs. Employee turnover (promotions, re-assignments, resignations) require this. Consulting programmers' jobs depend upon their ability to debug other people's programs. On-site diagnosis of computer system failures require this type of debugging. Significant errors, found to exist in the field, are repaired by others. In writing large system programs, debugging is often separate from coding. Indeed, reading other people's codes is sometimes part of programmer training. At an abstract level, nearly all of the world's diagnostic problems (*e.g.*, medical, political, mechanical) are done by people other than those that created the situation requiring diagnosis.

Each modified program was compiled with an IBM 1800 compiler to eliminate any syntactic error due to modification. Following this, the programs were used to analyze some artificial data. Formatted listings were prepared of the input to and output from each program (to be given to the two I/0 groups). Following this, the I/0 statements were removed from the programs themselves. Each of the four programs was then bugged with each of the three different classes of bugs, and again re-compiled. This produced 12 different listings. Only one statement was modified in each bugged program. Appendix 2 shows the 12 lines that were bugged, and this information allows the reader to reconstruct the 12 listings used.

Assignment bugs were planted by changing a variable in an equation or assignment statement. Array bugs were defined as "causing an array to exceed its dimensioned value". One way of planting these was by changing the subscript of an array. Iteration bugs were defined as "causing an inappropriate number of iterations over a particular calculation". One way of planting these was by changing the upper bound of a DO loop. These classes are, of course, not mutually exclusive. The intent was to study a variety of bugs at the expense of a taxonomy of bug-types. The definitions of Array and Iteration bugs were similar. Indeed, a year after completion of this manuscript, we discovered that what was treated during the experiment as an Iteration Bug and as an Array Bug for one of the programs (TABI) should have been just the opposite. Re-analysis of the main results, (a) with the TABI program excluded and (b) with the definitions (properly) reversed, showed no appreciable difference from the results reported here.

Six programmers were randomly assigned to each of four different groups. All were told that: (1) the listing was syntactically correct; (2) it contained one non-syntactic error in one line; (3) the I/0 statements were removed but the data were stored in the computer correctly; and (4) they should try to find the bug as fast as possible. The class of bug group was told whether the bug was an iteration, array, or assignment bug.

A fifth group was added after the experiment had started because it appeared that the three experimental groups were not debugging any faster than the control group, and because it appeared in most cases that when a programmer found the error, he also knew the correct repair action to take. In this fifth group, programmers were given a listing only and told which line the error was in (line number group). Their task was to indicate what was wrong with the line.

The experiment was designed so that the effect of seeing a program more than once could be evaluated separately from the main effects of programs, class of bugs, and debugging aids. The 12 listings that each of 30 programmers debugged were divided into four consecutive "blocks" of three each. Within a block was one listing of each bug type, each contained in a different program. Thus, each consecutive group of these listings contained three different programs and three different bug types. The six different orders of bug types within a block were each used an equal number of times (20). This, of course, essentially randomized the order of programs within a block. The average lag between the time a programmer debugged a particular program once and the time he debugged it again (but with a different bug in it), was about three other listings. Each programmer debugged all 12 listings, usually in four 30–120 min. sessions over a period of about two weeks. Within a session, each programmer usually debugged one block of three listings.

The programmers were timed with a stop watch, and results were recorded in tenths of min. The programmers were allowed to quit debugging a listing if they had not found the error in 15 min., or if they gave up before 45 min. They were then shown the error. Experience showed that extreme frustration resulted by this time. When the programmer believed that he had found the bug, the watch was stopped and he told the experimenter. The experimenter merely told him if he was correct or not. If he was incorrect, the watch was started again and he began debugging again.

The programmers were asked to move a felt-tip ink pen over the listing to indicate the sequence of statements to which they were attending. Pre-experimental work suggested that this was a feasible and non-interfering method to obtain gross information about the sequence of debugging behavior. Each programmer was also asked, following his debugging of a listing, how he found the bug; notes were kept on the answers.

The fact that subjects did not always find the bug (*cf.*, Figure 2) caused a problem in data analysis, both in choosing the appropriate measure of central tendency (mean, median, mode) and in choosing the appropriate variance analyses and significance tests. The data remained bi-modal under variance transformations, and with various metrics combining debug times and numbers of wrong guesses. All significance tests were based upon the reciprocals of debug times (called "speeds" in Figures 3–6). Reciprocals allow the assumption of infinitely long debugging times for the 27 cases in which the bug was not found. Results were about the same, however, when either 1/45 or zero was used for those cases. Results are presented in terms of (1) the "adjusted mean debug" times, excluding the cases in which the bug was not found; (2) the number of cases in which a bug was not found; (3) the median debug times; and (4) the reciprocal of debug times—the speed of debugging.

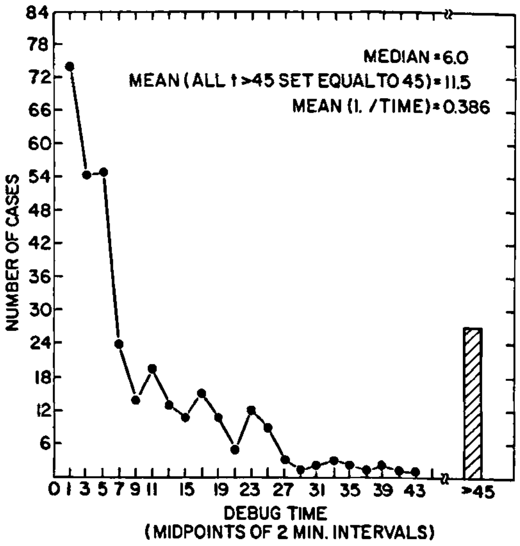

Figure 2. Debug times on the 360 listings. Bar at right represents the 27 cases in which the bug was never found.

RESULTS AND DISCUSSION

Figure 2 shows the distribution of debug times. The median time for programmers to find a bug in a listing was 6.0 min. Two-thirds of the bugs that were identified were found within 10 min. and seven-eighths were found within the first 20 min. Subjects did not detect a constant percent per minute of remaining bugs. On 27 of the 360 listings, programmers did not find the bug, as shown by the bar at the right of the figure.

These times seem fast[3] when compared with: (1) the times required to find nonsyntactic bugs in real-life; (2) the hours of debug times reported in experiments when programmers wrote and debugged their own programs (*e.g.*, Grant and Sackman, 1967); (3) our initial expectations which were weighted heavily by verbal predictions of professional programming personnel; and (4) programmers' statements that the task was difficult, and that they did not know enough statistics to understand the programs. Subjects reported that they normally do not debug this fast. They suggested that they would normally submit the program many times, using an interactive computer system, before they found the bug. (However, an

[3] Whether or not these times are "fast" is open to interpretation. We made one attempt to assess this by observing actual instances of experienced consulting programmers helping other real-life programmers debug their listings. Observations were made during seven 2-3 hour periods. The main observation from these few sessions was that the skilled consultant answered a majority of the questions brought to him, including some over the telephone, fairly quickly and simply. These questions were usually less complex than debugging a listing, however. In 10 or so cases which involved helping debug another person's listing, a median time of six min. was an acceptable estimate for the length of time required to identity what was *believed* to be the bug. Of course, the programmer then had to verify this on a computer. We thank Roland Carn for making these observations.

experiment just completed (Gould, 1974) showed that when another group of subjects was given access to a large interactive system to debug these same listings, they did not use it.) No subjects volunteered the suggestion that their experience seemed to imply, namely, that debugging with strong self-imposed pressure and few aids can, at least in some cases, be more efficient but probably more painful than their normal procedures.

A high motivational level undoubtedly contributed to the fast debugging times. Each programmer felt under pressure to perform well, even though the experimenters tried to provide a relaxed environment. Each appeared as if his reputation as an experienced programmer was being tested. Not finding a bug seemed very frustrating.

Debugging times were also undoubtedly affected by subjects' knowledge that there was always one and only one bug present, and that it was in a single line of the program. Knowing this, subjects could constrain their search strategy by judging the correctness of any statements under the assumption that all others were correct. An interesting future study would be to vary the number of bugged lines and so inform subjects. McClurkin and Naughton (1973) have studied subjects debugging listings with 15 bugs in them.

Practice Effects

Each programmer debugged the same program (but with different bugs in it) three times and, as shown in Figure 3, debugging was about twice as fast after subjects had seen a program once before ($F_{(2,50)} = 10.44; p < .01$). Table 1 is a summary of the analysis of variance of the reciprocals of the time scores; similar analyses were done on errors (defined below).

The distributions of debug times were similar on the second and third tries and more positively skewed than on the first try. The narrow-striped bars of Figure 3 show the

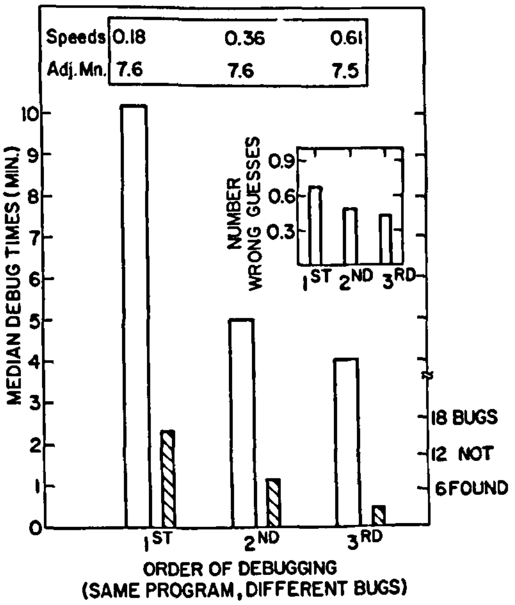

Figure 3. Median debug times (wide bars), number of bugs not found (striped bars), number of wrong guesses per programmer per listing about the location of the bug (insert right), speeds (mean reciprocal time), and adjusted mean debug times (above) for each of the three occasions that programmers debugged the same program (different bug each time). Adjusted mean debug times are based upon all the cases in which a bug was found, i.e., 104, 112, and 117, respectively.

number of listings on which programmers did not find the bug in 45 min. From a serious 13% on the first try, they dropped to 3% by the third try. The decrease in median debug times, although less striking, was still present when the cases in which programmers did not find the bug were removed from this calculation. The insert in Figure 3 shows that the mean number of false alarms, or occasions per program that programmers incorrectly stated the location of the bug, also decreased with practice ($F_{(2,50)} = 35.81; p < .001$). These false-alarms will henceforth be called "errors". They took the form of "I think the bug is in line ______ because of....", when in fact that line was correct. Because it was desired to tell a programmer only whether or not he had correctly identified the error each time he stopped debugging, no attempt was made to obtain detailed information about the errors.

The conclusion, based upon time and accuracy scores, is that debugging performance was at least twice as efficient the second time a programmer debugged a program as the first time. An interesting but presently unanswerable question is to what extent the following factors led to this more efficient debugging: (1) programmer's understanding of the data representation; (2) their memory for the statements themselves; (3) their mental representation of the flow of control in the program; or (4) their general expectations of the bugs to look for. Subjects rarely drew representations of the data on their listings. They never indicated to the experimenters, even when asked, a memory of instructions from a previous version of a program which then helped them debug a subsequent version of the same program. Programmers' tracings indicated they were clearly aware of the simple data flow in these programs. Programmers reported that after debugging one or two listings they had a much better idea of "what to look for".

TABLE 1

Summary Table of Analysis of Variance on the Reciprocals of the Time Scores (1/min.)

Source of Variance	df	Mean Square	F
1. Bugs (B)	2	4.9936	20.62**
2. Programs (P)	3	.9081	2.75*
3. Information Aids (A)	4	4.4096	4.61**
4. B X P	6	1.1201	2.66**
5. B X A	8	1.1638	4.81**
6. P X A	12	.9311	2.82**
7. B X P X A	24	.6198	1.47
8. S/A	25	.9564	
9. B X S/A	50	.2422	
10. P X S/A	75	.3305	
11. P X B X S/A	150	.4204	
Total	359		
12. Order (O)***	2	5.4213	10.44**
13. O X S/A	50	.5194	

* $p < .05$
** $p < .01$
*** Lines 12 and 13 are the variance in lines 1 and 9 re-analyzed for effects of order, or debugging the same program more than once, rather than for the effect of bugs.

Differences Among Programmers

The major finding about programmer differences concerns the number of errors programmers made. Table 2 shows that the ranges of errors, for the six programmers in each group, were between 4.5:1 and 15:1. Whereas these ranges are large, only 2 of the 30 programmers averaged one or more errors per program. The significance of this error measure is that it is an amplification factor for actual debugging times. An error might be roughly analogous to a resubmission of a program in which the programmer is confident that he has found the bug, when in fact he has not. The number of errors was, however, less than the number of submissions that other studies have found (Nagy and Pennebaker, 1971; Schatzoff, *et al.*, 1967).

TABLE 2

Ranges of Individual Differences among Programmers

	Wrong Guesses Per Listing	Median Time Scores (min.)	Mean Reciprocals (1/Debug Time)	Bugs Not Found In A Listing
No Aid	.1–1.5	5.4–10.6	.18– .31	0–2
Class of Bug	.1– .8	1.2– 9.6	.16– .66	0–2
I/0	.2– .9	6.3–16.9	.12– .26	1–2
I/0 + Correct 0	.1– .8	3.7–14.7	.12– .48	0–2
Line Number	.1– .7	1.9– 3.5	.45–1.47	0–1

Table 2 also shows that the range of differences among median debug times and the range of differences among mean debug speeds were about 2:1 for three of the groups and about 5:1 for the other two groups. The variance of the speed scores among the 30 programmers is about equal to the between-program variance (P) and about 20% as large as either the between-bugs (B) or between-information aids (A) sources of variance (Table 1).

On the one hand, a positive correlation between errors and debug times might be expected on the basis of individual differences. That is, maybe some programmers are fast and accurate, whereas others are slow and inaccurate. However, a negative correlation between errors and debug times might be expected on the basis that some programmers frequently guess about a bug's location and then use the resulting feedback to help isolate the actual bug. The experimenters never detected this latter strategy. A plot of the data revealed no strong correlation between errors and debug times, thus indicating no support for either possibility.

Programs

Figure 4 shows the important point that program, independent of bug, caused significant variation in debugging performance ($F_{(3,75)}$ = 2.75; $p < .05$). Programs that required longer debugging times also had a higher probability that the bug would never be found (striped bars).

About the same number of errors occurred on each program, regardless of debugging times. Thus, although these programs were relatively homogeneous (*i.e.*, FORTRAN data analysis programs, all about one-page long, linear in data flow, and had "good programming practice"), they differentially affected debugging efficiency. The magnitude of the program effect was not large in comparison to that caused by different types of bugs, however, but would probably be proportionately greater had pro-

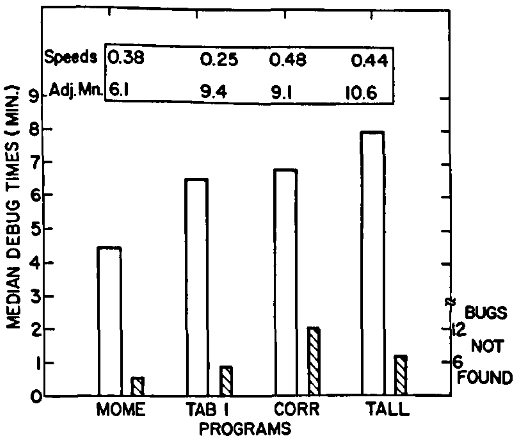

Figure 4. Median debug times (wide bars), number of bugs not found (striped bars), speeds (mean reciprocal time), and adjusted mean debug times (above) on each of four different programs.

grammers written and debugged their own programs. As discussed below, there was a Program X Bug interaction.

Bugs

Figure 5 shows how the class of bug affected debugging times ($F_{(2,50)}$ = 20.62; $p < .001$) and errors ($F_{(2,50)}$= 15.51; $p < .01$). The major finding was that time scores were about five times longer and errors were about twice as frequent on Assignment bugs as on the other classes of bugs. In addition, about five times more Assignment bugs were not found. The debug times and errors for Array and Iteration bugs were about the same, in accordance with the fact that they were very similar bugs.

It appears that there is nothing especially difficult about finding bugs that occur in assignment statements *per se*. Two so-called "Array" bugs (MOME-ARRAY and CORR-ARRAY, *cf.*, Appendix 2) were in assignment statements, and these led to a violation of FORTRAN language rules in that they caused an array to be exceeded. These were found relatively quickly. However, when the bug

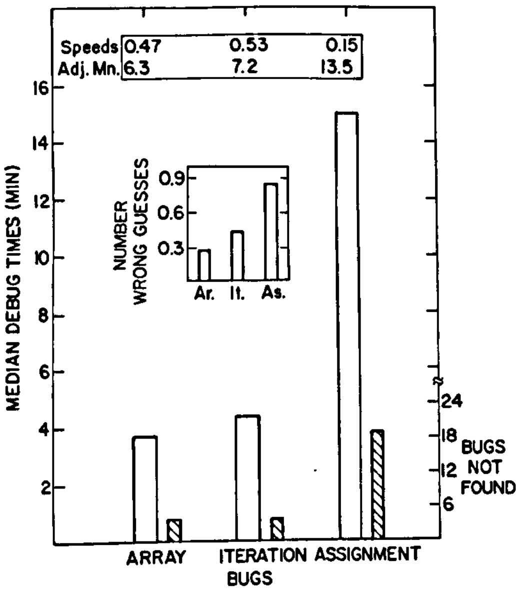

Figure 5. Median debug times (wide bars), number of bugs not found (striped bars), number of wrong guesses per programmer per listing about location of the bug (middle insert), speeds (mean reciprocal time) and adjusted mean debug times (above) on each of the classes of bugs.

appeared in an assignment statement and caused no change in the program flow, allocation, or iteration, the bug was harder to detect.

An interesting question is whether debugging performance is the same on a program, regardless of the bug in it; or, stated differently, is debugging performance the same for a class of bugs, regardless of the programs in which they occur? The answer seems to be "yes" for Array and Iteration bugs. There was a Bug X Program interaction for the time scores ($F_{(6,150)} = 2.66$; $p < .05$) and for errors ($F_{(6,150)} = 4.94$; $p < .001$). Plots of the data, however, suggested that these interactions were primarily due to two Assignment bugs. These two Assignment bugs (in TALL and CORR programs, Appendix 2) led to the longest debugging times (Table 4), the most errors (1/3 of all errors, Table 3), and the most cases (half) in which bugs were not caught. It should be noted in interpreting this interaction that, unlike most factorial experiments, the same level of Bug did not appear with each program. For example, the Array bug had a different form in each program.

General Debugging Strategy

Programmers, regardless of the group they were in, frequently appeared to employ a simple hierarchical debugging strategy. Since the programmers knew there were no syntactic errors present, they did not look for these, although in practice these are the first bugs they eliminate (Youngs, 1969, p. 121). Programmers first seemed to look for violations of the grammar of the programming language. These were violations that the compiler could not detect but that would provide clues to programmers about the bug. Examples are endless loops, arrays exceeded, foolish flow of control, and incorrect iteration. Evidence that programmers searched for this type of non-syntactic bug first comes from: (1) faster debugging times for listings containing such bugs; (2) verbal statements of programmers; (3) inferences drawn from their tracings; and (4) faster debugging times of assignment statements that contained such bugs (MOME and CORR Array bugs, Appendix 2) than assignment statements that contained so-called Assignment bugs.

If programmers did not detect an error at this level, they then delved into the substance of the program. They had to begin understanding what the program was intended to do and what the variables stood for. This means that they had to concentrate especially on assignment statements. It appears from the data that the programmers first concentrated on short assignment statements, and, if they still had not discovered the bug, they then moved on to longer assignment statements. There was a perfect rank-order correlation between the length (number of characters) of the assignment statements that contained the four Assignment bugs and the time required to debug them.

TABLE 3

Number of Incorrect Guesses (Errors) about Location of Bug (Each Cell Based upon 6 Listings)

	MOME			*TABI*			*CORR*			*TALL*			
	Array	*Iter.*	*Assign.*	*Array*	*Iter.*	*Assign.*	*Array*	*Iter.*	*Assign.*	*Array*	*Iter.*	*Assign.*	*Total*
No Help	4	5	10	3	7	1	4	5	4	2	3	8	56
Cl. Bug	4	1	4	0	8	4	0	1	10	0	0	0	32
I/O	1	1	2	1	6	4	7	2	4	1	0	5	34
I/O + CO	0	2	5	2	0	3	0	0	7	3	5	8	35
Line Numb.	2	2	3	0	2	1	0	0	11	1	0	7	29
	11	11	24	6	23	13	11	8	36	7	8	28	186

(Clearly, however, there are other measures of the psychological "length" of a statement than only the number of characters.)

The names given to variables in the listings contained information that provided the programmer with some understanding of the program, and these names probably had mnemonic value, also. In an informal demonstration, several other people were given one of the listings to debug in which the names of all the variables were changed to the symbols V1, V2, V3, *etc.* The results suggested that debug times were about twice as long.

Aids

Figure 6 shows the debug times for each information aid. The key result was that what almost anyone would assume to be useful debugging "aids" were of no help at all. Debugging times, although variable, were no faster for the I/O, I/O + correct output, and class of bug groups than for the no aid group. Thus, the no aid (control) group did not serve as a baseline to assess the value of the "aids", as was planned. Rather, programmers were highly flexible in adapting their debugging strategies to

TABLE 4

Median Debug Times (Minutes) (Each Cell Based Upon 6 Listings)

	MOME			*TABI*			*CORR*			*TALL*		
	Array	*Iter.*	*Assign.*	*Array*	*Iter.*	*Assign.*	*Array*	*Iter.*	*Assign.*	*Array*	*Iter.*	*Assign.*
No Help	4.2	4.5	10.2	3.0	13.7	8.3	7.9	8.0	20.3	5.4	1.5	24.7
Cl. of Bug	2.2	3.1	12.6	4.6	21.3	8.6	1.6	5.3	21.5	1.3	1.8	15.0
I/O	2.4	7.9	8.9	5.2	14.3	16.1	16.6	12.2	4.5	9.3	8.2	31.7
I/O + Co	2.6	5.4	14.1	11.5	4.8	20.0	5.9	3.4	25.8*	32.1	7.3	27.0
Line Number	2.4	1.8	4.7	2.0	3.5	6.1	1.0	0.9	38.1	1.9	1.0	17.8

* The median was between a time score and a case in which the bug was not found. In the latter case 45 min. was used, and the mean of these two scores was calculated to estimate the median debug time.

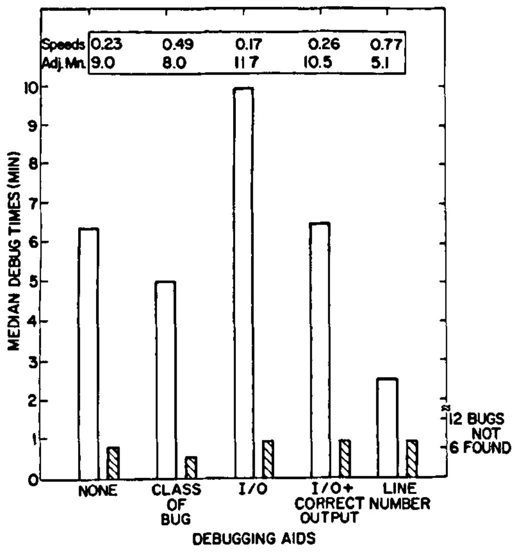

Figure 6. Median debug times (wide bars), number of bugs not found (striped bars), speeds (mean reciprocal time), and adjusted mean debug times (above) for the five groups of programmers.

the kinds of information they were given. This adaptability is characteristic of human information processing, as Simon (1969) has pointed out. Although one might suggest that this general lack of an "aids" effect was due merely to the task being easy, this is hard to reconcile with some programmers' verbal statements that the task was hard and that they used the "aids" they were given.

The no help group made about 60% more errors than did the other four groups (Table 3). The number of bugs not found (striped bars) was the same for the five groups. The overall significant F-test for Aids in Table 1 was due to the fact that the line number group's time scores differed significantly from the control group's time scores (Duncan Range Test (1955); $p < .05$).

Although the line number group was faster at debugging than the others, the important point is that they still require 25–50% of the time that the other group required. This result suggests that a very real part of finding a non-syntactic error involves acquiring knowledge of the intended state of the program. The result does not imply, however, that 25–50% of normal debug time is devoted solely to this.

The powerful effect of Assignment bugs was primarily responsible for the significant Bugs X Aids interaction ($F_{(8,275)} = 4.81; p < .01$). All five groups required about the same amount of time on the Assignment bugs, whereas on the other two bugs the groups were somewhat different. One hypothesis was that one aid might be better for one kind of bug whereas another aid might be better for another class of bug. The evidence in the Bugs X Aids interaction did not support this, however.

No aid group. In every case, programmers started at the top of a listing and proceeded downwards. The first time they saw a program they read the description of it carefully. On the second and third occasions, they did not spend much time on this description. However, this probably does not entirely account for faster debugging. There were no selective starting points, *e.g.*, in the middle of the listing, at the beginning of a loop, *etc.* Once they started at the top, they generally went through the program carefully and completely. For example, the first time one programmer saw one program, he spent 40 min. on the first pass through it. Typically, programmers reported that they were not sure of what the program actually was designed to do, although they understood the rules of the programming language. They sometimes seemed to understand the semantics of the program in a local sense. Perhaps appropriate analogies are how some professional proof-readers or high-speed typists interact with text: they execute their jobs relatively oblivious to the substantive or conceptual content of the text, and yet pick up phrases or sentences that, although syntactically correct, "don't make sense"

On about half of the 72 listings, programmers found the bug on their first pass through the listing. On the other half, programmers would typically start at the top again and proceed downward. Programmers in this and the other groups usually did not draw a picture of the data representation or write out calculations on their listings. They attempted to ascertain the correctness of complicated assignment statements only as a last resort. Programmers were not at all bothered by the absence of I/O statements in the listings.

Class of bug group. One might expect that telling programmers what the class of bug was would increase their selectivity in debugging, and that they would focus upon only a subset of statements. Based upon the programmers' tracings, there was clear evidence on 15 of the 72 listings of such selectivity. At least 50% of the remainder of the listings showed clear evidence of no selectivity, whereas it was not possible to make accurate judgments on the remainder. This selectivity was not sufficient to reduce debugging time significantly, however, although it may have been related to the errors programmers made, which clustered on a few listings (Table 3).

I/O group. This group was observed to use the output listings, but not to spend much time on them. (There were no Read, Write, or Format statements in the programs themselves, however.) Nobody wrote on either their input or output listings. Based upon tracings, this group seemed more selective in the statements they attended to than did the no aid group. If they did not find the bug relatively quickly, their tracings showed that they made several passes through selected parts of the listing. The usefulness of the output data probably depended upon the bug-program combination. In one case (CORR-Assignment, Appendix 2), a vector (*e.g.*, of means) was printed out as 0.0, and programmers reported this was useful; in other cases programmers reported that the I/O was not of much use. Half of the 34 errors of this group were on 3 of the 12 listings (Table 3), but no special relationship among these was discernible.

An interesting anecdote, which occurred because one programmer in this group was given by mistake the same listing that he debugged a week earlier, suggests the time score data may be reliable. He required 5.0 min. to debug it the first time, and 4.2 min. to debug the same bug, in the same program, the second time.

I/O plus correct output group. The programmers typically read the description above each listing, then compared the actual output and the correct input, and then concentrated on the code. The important question was how this group differed from the I/O group. It was assumed that this group would use a "means-end" problem-solving strategy (Newell and Simon, 1972). In this strategy a person finds a difference between a current and desired situation and then acts to reduce it. The only observable difference between this group and the others was that programmers occasionally wrote out calculations, *e.g.*, values of variables at each iteration of a loop, which suggests they were using this mean-end strategy. One programmer wrote notes to himself, one programmer circled the differences between the actual and correct output listings, while the others reported that they at least observed them.

Line number group. Based upon their tracings, these programmers were more selective in the statements to which they attended. Frequently they first attended to the line containing the bug, then attended to a Dimension statement and other explanatory statements of data structure. In general, they spent most of their time on these statements and in the local region of the error. The important conclusion from this group is that, in most cases, a programmer must be able to discover the intended or goal state of a program (or problem) before he can discover what is wrong with the present state of the program.

GENERAL DISCUSSION

A general principle in programming and debugging may be that programmers focus their attention on a local region of code, which is defined by both geographical factors (*e.g.*, neighboring statements) and conceptual factors. This is supported by general limitations on human memory and processing capacity (*cf.*, Lindsay and Norman, 1972, for a recent introduction to this subject). Suggestive support is provided in this experiment. For example, the CORR Assignment bug required the most time to detect (Table 4). The statement that was bugged (line 990) was 19 lines away from statement 1710, which provided the clue or symptom for the bug. This separation, which was greater than that for any other bug, may have exceeded usual "locality" limits.

The indication of a possible hierarchical debugging strategy and a general tendency to put off attacking complex parts of code suggest what might be called an "ease into it" strategy. This conclusion is consistent with Nagy and Pennebaker's (1971) finding that student programmers, while debugging, typically make a small change, resubmit the program, hope that it works right, find another bug, make another small change, resubmit it, *etc.* This repetitive cycle is characteristic, as opposed to programmers trying to eliminate several bugs at once.

Subjects demonstrated (based upon their "fast" debug times and "few" errors) and acknowledged verbally that sitting down and struggling by oneself with code can often lead to more efficient debugging than their usual methods of non-interactive (*e.g.*, a series of re-submissions; writing out several variables) or interactive debugging. Certainly these results cannot apply to the debugging of all programs. Nevertheless, the significance of this "sweat and blood" approach, which demands disciplined motivation and concentration, is that it provides an empirical template with which a programmer can evaluate the efficiency of his own present methods of debugging. We loosely speculate that this "sweat and blood" approach includes the "ease into it" strategy. We suspect that the only ingredient it adds to the strategy is a serious self-imposed pressure. The programmer, in effect, locks himself into an imaginery room until he finds the bug.

Since subjects were not debugging "as usual", it might be argued that these results have little generality to the real world. On the contrary, however, these results suggest that debugging strategies are under the control of programmers, and that they are influenced by the information and time available, the programmer's motivation, and his experience with the program. The results indicate that the concept of "debugging as usual" has hardly even description validity.

The fact that Array and Iteration bugs were easier to detect than Assignment bugs is inconsistent with the reason these were studied in the first place (the reason being that consulting programmers stated these were the hardest for people to debug). Evidently, when debugging a program written by oneself, a programmer does not use the same strategy of first looking for potential violations of high-level programming language conventions that subjects in this experiment seemed to use when debugging programs written by others.

The finding that the line number group debugged only about twice as fast as the other groups, even though they were given almost the ultimate in a debugging aid, provides a small bit of evidence for some very risky speculation. While there is considerable on-going research in computer sciences to provide professional programmers with better programming tools, could it be that productivity increases of no more than a factor of two can ever be achieved with this approach? In view of Boehm's (1973) estimates of programming costs, this is not very significant. Of course, the present experiment is merely exploratory in this regard, and direct research is needed to determine theoretical limits of productivity increases.

There are several important questions that seem critical to generalizing the results of this experiment. One is the extent to which results on 1-page listings generalize to longer programs. Conversations with programmers indicate that they can sometimes quickly isolate a bug to within 40-50 statements, although these statements may not be as "local" (*cf.*, above) as those in the listings studied, whereas other times they cannot.

A second question is raised by the fact that programmers in this experiment realized that there was always one and only one bug, whereas in reality this degree of certainty does not exist. Subjects reported using this information in their debugging strategies. Indeed, occasionally a subject continued to debug even though he felt a bug was not present!

A related question has to do with one-bug programs, as studied here, *vs.* multiple-bug programs, as often really occur. To the extent that programmers look for only one bug at a time, these data may have some relation to both cases.

Another question is the one raised earlier, namely, how the debugging of programs written by others relates to the real world. At an abstract level, the results relate to most diagnostic situations since the diagnostician usually does not create the system or product he debugs. In programming, there are, of course, many applications in which programmers debug programs written by others. An experiment analogous to the present one but with programmers debugging errors (either those created by themselves or by the experimenter) in their own programs would now be useful. Any major differences in results would be instructive in indicating, for example, how coding and debugging are not independent. In the absence of data to the contrary, however, we would cautiously use the main findings of this study as guidelines in thinking about the case in which programmers debug their own programs also.

With these several questions in mind, probably the proper perspective with which to view the present results is that (1) programming, as done by 10^5–10^6 people in the world today (Computer Yearbook, 1972), involves very complex cognitive processes that have not been studied much, and (2) this experiment demonstrates the feasibility of studying, in a controlled way, one aspect of programming. As such, the data should have at least heuristic value in formulating hypotheses for further study. Some of the questions raised could be profitably investigated. Real understanding of programming awaits a behavioral theory of how people generate plans or sequences of action, and how these are tested and modified. Programming probably requires as much rigor in this regard as any other cognitive task.

REFERENCES

Adams, J. and Cohen, L. Time-sharing vs. instant batch processing: An experiment in programmer training. *Computers and Automation*, March, 1969, 30–34.

Boehm, B. W. Software and its impact: A quantitative assessment. *Datamation*, May, 1973, 48–59.

Boies, S. F. and Gould, J. D. Syntactic errors in computer programming. *Human Factors*, 1974, 16, 253–257.

Bryan, G. E. JOSS: 20,000 hours at the console–A statistical summary. Rand Corporation Memo RM-5359-PR, August, 1967.

Carbonell, J. R., Elkind, J. I., and Nickerson, R. S. On the psychological importance of time in a time sharing system. *Human Factors*, 1968, 10, 135-142.

Computer yearbook 72. Detroit: Computer Yearbook Company, 1972.

Delaney, W. A. Predicting the costs of computer programs. *Data Processing Magazine*, October, 1966, 32–34.

Duncan, D. B. Multiple range and multiple *F* Tests. *Biometrics*, 1955, 11, 1–42.

Erickson, W. F. A pilot study of interactive versus noninteractive debugging, SDC Technical Report, TM-3296, December, 1966.

Gold, N. M. A methodology for evaluating time-shared computer system usage. Unpublished Ph.D. thesis, Massachusetts Institute of Technology, August, 1967.

Gould, J. D., Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Systems*, 1974 (in press).

Gould, J. D., Doherty, W. J., and Boies, S. J. Bibliography of behavioral aspects of on-line computer programming. IBM Research Report, RC-3513, 1971.

Grant, E. E. and Sackman, H. An exploratory investigation of programmer performance under on-line and off-line conditions. *IEEE.Transactions on Human Factors in Electronics*, 1967, HFE-8, 33–48.

Hunt, E., Diehr, G., and Garnatz, D. Who are the users? An analysis of computer use in a university computer center. University of Washington, Computer Service Group Technical Report No. 70-09-05, September, 1970.

Knuth, D. E. An empirical study of FORTRAN programs. IBM Research Report, RC-3276, March, 1971.

Lindsay, P. H. and Norman, D. A. *An introduction to psychology.* New York: Academic Press, 1972.

Mayer, D. B. and Stalnaker, A. W. Selection and evaluation of computer personnel–the research history of SIG/CPR. *Proceedings of A.C.M.,* 1968, 657-670.

McClurkin, P. and Naughton, J. Report on pilot code reading: Experiment #1. Unpublished manuscript, IBM Advanced Technology Group, Gaithersburg, Maryland.

Moulton, P. E. and Muller, M. E. DITRAN-A compiler emphasing diagnostics. *Communications of A.C.M.,* 1967, 10, 45–52.

Nagy, G. and Pennebaker, M. A step toward automatic analysis of logically undetectable programming errors. IBM Research Report, RC-3407, 1971.

Neisser, U. MAC and its users. M.I.T. Memorandum MAC-M-185, September, 1964.

Newell, A. and Simon, H. A. *Human problem solving.* Englewood Cliffs, New Jersey: Prentice-Hall, 1972.

Rubey, R. J. A comparative evaluation of PL/I. *Datamation,* December, 1968, 20–25.

Sackman, H. *Man-computer problem solving: Experimental evaluation of time-sharing and batch processing.* New York: Auerbach, 1970.

Schatzoff, M., Tsao, R., and Wiig, R. An experimental comparison of time-sharing and batch processing. *Communications of the ACM,* 1967, 10, 261-265.

Scherr, A. An analysis of time-shared computer systems. Research Monograph No. 36, M.I.T. Press, Cambridge, Massachusetts, 1967.

Simon, H. A. Human problem solving: The state of the theory in 1970. *American Psychologist,* 1971, 26, 145–159.

Simon, H. A. *The sciences of the artificial.* Cambridge, Massachusetts: M.I.T. Press, 1969.

Smith, L. B. A comparison of batch processing and instant turnaround. *Communications of the A.C.M.,* 1967, 10, 495-500.

Walter, E. W. and Wallace, V. L. Further analysis of a computing center environment. *Communications of the ACM,* 1967, 10, 266–272.

Appendix 1

```
C     ............................................................MOME  05
C     PURPOSE OF PROGRAM                                          MOME  10
C           ASSUME A SET OF DATA POINTS,EACH OF WHICH FALLS INTO  MOME  20
C           ONE OF MANY CATEGORIES.  F(I) IS THE NUMBER OF DATA POINTS MOME  30
C           IN CATEGORY I.   THE CATEGORIES ARE SPACED AT EQUAL   MOME  40
C           INTERVALS ALONG A SINGLE DIMENSION.  THIS PROGRAM FINDS MOME  50
C           THE FIRST FOUR MOMENTS OF THESE DATA POINTS.          MOME  60
C                                                                 MOME 360
C     ............................................................MOME 370
C     INSTRUCTIONS TO SUBJECTS                                    MOME  90
C           ASSUME THAT DATA ARE CORRECTLY IN CORE IN THE FORM--  MOME 100
C            20 EQUAL INTERVAL CATEGORIES OF FREQUENCY DATA,F(1)-F(20) MOME 110
C            UBO(1) = LOWER BOUND OF ALL CATEGORIES               MOME 130
C            UBO(2) = SIZE OF CLASS INTERVAL                      MOME 140
C            UBO(3) = UPPER BOUND OF ALL CATEGORIES               MOME 150
C     ............................................................MOME 170
C                                                                 MOME 380
      DIMENSION F(20), UBO(3),ANS(4)                              MOME 400
C                                                                 MOME 410
      DO 100 I=1,4                                                MOME 420
  100 ANS(I)=0.0                                                  MOME 430
C                                                                 MOME 440
C     CALCULATE THE NUMBER OF CATEGORY INTERVALS                  MOME 450
C                                                                 MOME 460
      N=(UBO(3)-UBO(1))/UBO(2)+0.5                                MOME 470
C                                                                 MOME 480
C     CALCULATE TOTAL FREQUENCY                                   MOME 490
C                                                                 MOME 500
      T=0.0                                                       MOME 510
      DO 110 I=1,N                                                MOME 520
  110 T=T+F(I)                                                    MOME 530
C                                                                 MOME 540
  115 NOP=5                                                       MOME 560
  120 JUMP=1                                                      MOME 570
C                                                                 MOME 600
C         FIRST MOMENT                                            MOME 610
C                                                                 MOME 620
  150 DO 160 I=1,N                                                MOME 630
      FI=I                                                        MOME 640
  160 ANS(1)=ANS(1)+F(I)*(UBO(1)+(FI-0.5)*UBO(2))                 MOME 650
      ANS(1)=ANS(1)/T                                             MOME 660
C                                                                 MOME 670
      GO TO (350,200,250,300,200), NOP                            MOME 680
C                                                                 MOME 690
C         SECOND MOMENT                                           MOME 700
C                                                                 MOME 710
  200 DO 210 I=1,N                                                MOME 720
      FI=I                                                        MOME 730
  210 ANS(2)=ANS(2)+F(I)*(UBO(1)+(FI-0.5)*UBO(2)-ANS(1))**2       MOME 740
      ANS(2)=ANS(2)/T                                             MOME 750
      GO TO (250,350), JUMP                                       MOME 760
C                                                                 MOME 770
C         THIRD MOMENT                                            MOME 780
C                                                                 MOME 790
  250 DO 260 I=1,N                                                MOME 800
      FI=I                                                        MOME 810
  260 ANS(3)=ANS(3)+F(I)*(UBO(1)+(FI-0.5)*UBO(2)-ANS(1))**3       MOME 820
      ANS(3)=ANS(3)/T                                             MOME 830
      GO TO (300,350), JUMP                                       MOME 840
C                                                                 MOME 850
C         FOURTH MOMENT                                           MOME 860
C                                                                 MOME 870
  300 DO 310 I=1,N                                                MOME 880
      FI=I                                                        MOME 890
  310 ANS(4)=ANS(4)+F(I)*(UBO(1)+(FI-0.5)*UBO(2)-ANS(1))**4       MOME 900
      ANS(4)=ANS(4)/T                                             MOME 910
  350 CALL EXIT                                                   MOME 920
      END                                                         MOME 930
```

Appendix 1 (*continued*)

```
C                                                                       TALL  10
C     ..................................................................TALL  20
C        PURPOSE OF PROGRAM                                             TALL  50
C            TO CALCULATE TOTAL,MEAN,STANDARD DEV., MINIMUM, MAXIMUM    TALL  70
C            FOR EACH VARIABLE IN A SET (OR A SUBSET) OF OBSERVATIONS   TALL  80
C     ..................................................................TALL  85
C        INSTRUCTIONS TO SUBJECTS                                       TALL  77
C            ASSUME AN OBSERVATION MATRIX,NO X NV.  NO = NUMBER OF      TALL  90
C            OBSERVATIONS = 30, AND NV = NUMBER VARIABLES = 20.  ASSUME TALL 100
C            THESE SCORES ARE CORRECTLY READ INTO CORE IN A VECTOR,     TALL 110
C            A(1) - A(600).                                             TALL 120
C            S(1)-S(30) = INPUT VECTOR INDICATING SUBSET OF A. ONLY     TALL 150
C                   OBSERVATIONS WITH A NON-ZERO S(J) ARE CONSIDERED.   TALL 160
C                   VECTOR LENGTH IS NO.                                TALL 170
C            TOTAL(1)-TOTAL(20) = OUTPUT VECTOR OF TOTALS OF EACH VAR.  TALL 180
C            AVER(1)-AVER(20) = OUTPUT VECTOR OF AVER'S OF EACH VAR.    TALL 200
C            SD(1)-SD(20) = OUTPUT VECTOR OF STD. DEVS. OF EACH VAR.    TALL 220
C            VMIN(1)-VMIN(20) = OUTPUT VECTOR OF MINIMA OF EACH VAR.    TALL 260
C            VMAX(1)-VMAX(20) = OUTPUT VECTOR OF MAXIMA OF EACH VAR.    TALL 280
C     ..................................................................TALL 450
C                                                                       TALL 460
      DIMENSION A(600),S(30),TOTAL(20),AVER(20),SD(20),VMIN(20),VMAX(20)TALL 480
C                                                                       TALL 490
      NV = 20                                                           TALL 502
      NO = 30                                                           TALL 504
C        CLEAR OUTPUT VECTORS AND INITIALIZE VMIN,VMAX                  TALL 500
C                                                                       TALL 510
      IER=0                                                             TALL 511
      DO 1 K=1,NV                                                       TALL 520
      TOTAL(K)=0.0                                                      TALL 530
      AVER(K) = 0.0                                                     TALL 540
      SD(K) = 0.0                                                       TALL 550
      VMIN(K)=1.0E10                                                    TALL 560
    1 VMAX(K)=-1.0E10                                                   TALL 570
C                                                                       TALL 580
C        TEST SUBSET VECTOR                                             TALL 590
C                                                                       TALL 600
      SCNT=0.0                                                          TALL 610
      DO 7 J=1,NO                                                       TALL 620
      IJ=J-NO                                                           TALL 630
      IF(S(J)) 2,7,2                                                    TALL 640
    2 SCNT=SCNT+1.0                                                     TALL 650
C                                                                       TALL 660
C        CALCULATE TOTAL, MINIMA, MAXIMA                                TALL 670
C                                                                       TALL 680
      DO 6 I=1,NV                                                       TALL 690
      IJ=IJ+NO                                                          TALL 700
      TOTAL(I)=TOTAL(I)+A(IJ)                                           TALL 710
      IF(A(IJ)-VMIN(I)) 3,4,4                                           TALL 720
    3 VMIN(I)=A(IJ)                                                     TALL 730
    4 IF(A(IJ)-VMAX(I)) 6,6,5                                           TALL 740
    5 VMAX(I)=A(IJ)                                                     TALL 750
    6 SD(I)=SD(I)+A(IJ)*A(IJ)                                           TALL 760
    7 CONTINUE                                                          TALL 770
C                                                                       TALL 780
C        CALCULATE MEANS AND STANDARD DEVIATIONS                        TALL 790
C                                                                       TALL 800
      IF (SCNT)8,8,9                                                    TALL 801
    8 IER=1                                                             TALL 802
      GO TO 15                                                          TALL 803
    9 DO 10 I=1,NV                                                      TALL 810
   10 AVER(I)=TOTAL(I)/SCNT                                             TALL 820
      IF (SCNT-1.0) 13,11,13                                            TALL 821
   11 IER=2                                                             TALL 822
      DO 12 I=1,NV                                                      TALL 823
   12 SD(I)=0.0                                                         TALL 824
      GO TO 15                                                          TALL 825
   13 DO 14 I=1,NV                                                      TALL 826
   14 SD(I)=SQRT(ABS((SD(I)-TOTAL(I)*TOTAL(I)/SCNT)/(SCNT-1.0)))        TALL 830
   15 CALL EXIT                                                         TALL 840
      END                                                               TALL 850
```

Appendix 1 (*continued*)

```
C
C     ..............................................................
C     PURPOSE OF PROGRAM
C              TO TABULATE FOR ONE VARIABLE IN AN OBSERVATION MATRIX (OR
C              A MATRIX SUBSET),THE FREQUENCY AND PERCENT FREQUENCY OF
C              OCCURRENCE OF VARIOUS SCORES IN GIVEN CLASS INTERVALS.  IN
C              ADDITION, IT CALCULATES FOR THE SAME VARIABLE THE TOTAL,
C              AVERAGE, STANDARD DEVIATION, MINIMUM, AND MAXIMUM VALUES.
C     ..............................................................
C     INSTRUCTIONS TO SUBJECTS
C              ASSUME AN OBSERVATION MATRIX,NO X NV.  NO = 30. NV = 20.
C              ASSUME THE SCORES ARE CORRECTLY READ INTO CORE IN A VECTOR,
C              A(1) - A(600).
C              S(1) - S(30) = INPUT VECTOR INDICATING SUBSET OF A TO BE
C              ANALYZED.
C              UBO(1)-UBO(3) = INPUT VECTOR GIVING LOWER LIMIT,NUMBER OF
C              INTERVALS, AND UPPER LIMIT OF VARIABLE TO BE TABULATED
C              IN UBO(1),UBO(2), AND UBO(3) RESPECTIVELY.
C     ..............................................................
      DIMENSION A(600),S(30),UBO(3),FREQ(30),PCT(30),STATS(5),WBO(3)
      NO = 30
      NV = 20
      NOVAR = 5
      DO 5 I=1,3
    5 WBO(I)=UBO(I)
C
C        CALCULATE MIN AND MAX
C
      VMIN=1.0E10
      VMAX=-1.0E10
      IJ=NO*(NOVAR-1)
      DO 30 J=1,NO
      IJ=IJ+1
      IF(S(J)) 10,30,10
   10 IF(A(IJ)-VMIN) 15,20,20
   15 VMIN=A(IJ)
   20 IF(A(IJ)-VMAX) 30,30,25
   25 VMAX=A(IJ)
   30 CONTINUE
      STATS(4)=VMIN
      STATS(5)=VMAX
C
C        DETERMINE LIMITS
C
      IF(UBO(1)-UBO(3)) 40,35,40
   35 UBO(1)=VMIN
      UBO(3)=VMAX
   40 INN=UBO(2)
C
C        CLEAR OUTPUT AREAS
C
      DO 45 I=1,INN
      FREQ(I)=0.0
   45 PCT(I)=0.0
      DO 50 I=1,3
   50 STATS(I)=0.0
C
C        CALCULATE INTERVAL SIZE
C
      SINT=ABS((UBO(3)-UBO(1))/(UBO(2)-2.0))
C
C        TEST SUBSET VECTOR
C
      SCNT=0.0
      IJ=NO*(NOVAR-1)
      DO 75 J=1,NO
      IJ=IJ+1
      IF(S(J)) 55,75,55
   55 SCNT=SCNT+1.0
C
C        DEVELOP TOTAL AND FREQUENCIES
C
      STATS(1)=STATS(1)+A(IJ)
      STATS(3)=STATS(3)+A(IJ)*A(IJ)
      TEMP=UBO(1)-SINT
      INTX=INN-1
      DO 60 I=1,INTX
      TEMP=TEMP+SINT
      IF(A(IJ)-TEMP) 70,60,60
   60 CONTINUE
      IF(A(IJ)-TEMP) 75,65,65
   65 FREQ(INN)=FREQ(INN)+1.0
      GO TO 75
   70 FREQ(I)=FREQ(I)+1.0
   75 CONTINUE
      IF (SCNT)79,105,79
C
C        CALCULATE RELATIVE FREQUENCIES
C
   79 DO 80 I=1,INN
   80 PCT(I)=FREQ(I)*100.0/SCNT
C
C        CALCULATE MEAN AND STANDARD DEVIATION
C
      IF(SCNT-1.0) 85,85,90
   85 STATS(2)=STATS(1)
      STATS(3)=0.0
      GO TO 95
   90 STATS(2)=STATS(1)/SCNT
      STATS(3)=SQRT(ABS((STATS(3)-STATS(1)*STATS(1)/SCNT)/(SCNT-1.0)))
   95 DO 100 I=1,3
  100 UBO(I)=WBO(I)
  105 CALL EXIT
      END
```

Appendix 1 (*continued*)

```
C     ..................................................................CORR  20
C                                                                       CORR  30
C        PURPOSE OF PROGRAM                                             CORR  02
C            TO COMPUTE MEANS, STANDARD DEVIATIONS. SUMS OF CROSS-      CORR  06
C            PRODUCTS OF DEVIATIONS, AND CORRELATION COEFFICIENTS OF    CORR  08
C             DATA IN A N BY  M MATRIX.  N = NUMBER OF OBSERVATIONS.    CORR  10
C            M = NUMBER OF VARIABLES.                                   CORR  12
C     ..................................................................CORR 400
C        INSTRUCTIONS TO SUBJECTS                                       CORR  14
C            ASSUME DATA FROM THE N = 30 BY M = 20 MATRIX ARE CORRECTLY CORR  16
C            STORED IN CORE IN A VECTOR  X(1) -  X(600).                CORR  18
C     ..................................................................CORR  90
      DIMENSION X(600),XBAR(20),STD(20),RX(400),R(400),B(20),D(20),T(20)CORR 600
C                                                                       CORR 610
C                                                                       CORR 790
C     INITIALIZATION                                                    CORR 800
      M = 20                                                            CORR 802
      N = 30                                                            CORR 804
      DO 100 J=1,M                                                      CORR 820
      B(J)=0.0                                                          CORR 830
  100 T(J)=0.0                                                          CORR 840
      K=(M*M+M)/2                                                       CORR 850
      DO 102 I=1,K                                                      CORR 860
  102 R(I)=0.0                                                          CORR 870
      FN=N                                                              CORR 880
      L=0                                                               CORR 890
C                                                                       CORR 900
  105 DO 108 J=1,M                                                      CORR 950
      DO 107 I=1,N                                                      CORR 960
      L=L+1                                                             CORR 970
  107 T(J)=T(J)+X(L)                                                    CORR 980
      XBAR(J)=T(J)                                                      CORR 990
  108 T(J)=T(J)/FN                                                      CORR1000
C                                                                       CORR1010
      DO 115 I=1,N                                                      CORR1020
      JK=0                                                              CORR1030
      L=I-N                                                             CORR1040
      DO 110 J=1,M                                                      CORR1050
      L=L+N                                                             CORR1060
      D(J)=X(L)-T(J)                                                    CORR1070
  110 B(J)=B(J)+D(J)                                                    CORR1080
      DO 115 J=1,M                                                      CORR1090
      DO 115 K=1,J                                                      CORR1100
      JK=JK+1                                                           CORR1110
  115 R(JK)=R(JK)+D(J)*D(K)                                             CORR1120
C                                                                       CORR1660
C     CALCULATE MEANS                                                   CORR1670
C                                                                       CORR1680
  205 JK=0                                                              CORR1690
      DO 210 J=1,M                                                      CORR1700
      XBAR(J)=XBAR(J)/FN                                                CORR1710
C                                                                       CORR1720
C     ADJUST SUMS OF CROSS-PRODUCTS OF DEVIATIONS                       CORR1730
C     FROM TEMPORARY MEANS                                              CORR1740
C                                                                       CORR1750
      DO 210 K=1,J                                                      CORR1760
      JK=JK+1                                                           CORR1770
  210 R(JK)=R(JK)-B(J)*B(K)/FN                                          CORR1780
C                                                                       CORR1790
C     CALCULATE CORRELATION COEFFICIENTS                                CORR1800
C                                                                       CORR1810
      JK=0                                                              CORR1820
      DO 220 J=1,M                                                      CORR1830
      JK=JK+J                                                           CORR1840
  220 STD(J)= SQRT( ABS(R(JK)))                                         CORR1850
      DO 230 J=1,M                                                      CORR1860
      DO 230 K=J,M                                                      CORR1870
      JK=J+(K*K-K)/2                                                    CORR1880
      L=M*(J-1)+K                                                       CORR1890
      RX(L)=R(JK)                                                       CORR1900
      L=M*(K-1)+J                                                       CORR1910
      RX(L)=R(JK)                                                       CORR1920
      IF(STD(J)*STD(K)) 225, 222, 225                                   CORR1930
  222 R(JK)=0.0                                                         CORR1940
      GO TO 230                                                         CORR1950
  225 R(JK)=R(JK)/(STD(J)*STD(K))                                       CORR1960
  230 CONTINUE                                                          CORR1970
C                                                                       CORR1980
C     CALCULATE STANDARD DEVIATIONS                                     CORR1990
C                                                                       CORR2000
      FN=SQRT(FN-1.0)                                                   CORR2010
      DO 240 J=1,M                                                      CORR2020
  240 STD(J)=STD(J)/FN                                                  CORR2030
C                                                                       CORR2040
C     COPY THE DIAGONAL OF THE MATRIX OF SUMS OF CROSS-PRODUCTS OF      CORR2050
C     DEVIATIONS FROM MEANS.                                            CORR2060
C                                                                       CORR2070
      L=-M                                                              CORR2080
      DO 250 I=1,M                                                      CORR2090
      L=L+M+1                                                           CORR2100
  250 B(I)=RX(L)                                                        CORR2110
      CALL EXIT                                                         CORR2120
      END                                                               CORR2130
```

Appendix 2: The 12 Program Bugs Used in the Experiment

Program	*Bug*	*Modified Line No.*	*Modification*
TALL	Array	690	DO 6 I = 1, NO
TALL	Iteration	560-570	Lines were interchanged*
TALL	Assignment	830	(. . ./SCNT)/SCNT-1.0))
MOME	Array	750	Ans(2) = Ans(NOP)/T
MOME	Iteration	720	200 DO 210 I = 1, JUMP
MOME	Assignment	510	T = N
TAB1	Array	900	DO 50 i = 1, inn
TAB1	Iteration	11240	79 DO 80 I = 1, NOVAR
TAB1	Assignment	830	40 Inn = UBO(3)
CORR	Array	980	107T(I) = T(I) + X(L)
CORR	Iteration	1830	DO 220 J = 1, JK
CORR	Assignment	990	B(J) = T(J)

* The line label at the far right was interchanged also to eliminate this cue.