OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

Program comprehension differences in debugging

Murthi Nanja
Curtis Cook
Computer Science Department
Oregon State University
Corvallis, Oregon 97331

88-60-11

Program comprehension differences in debugging

MURTHI NANJA and CURTIS COOK
Department of Computer Science, Oregon State University, Corvallis, OR 97331

Many studies have shown that expert programmers are more effective at debugging programs than novice programmers. These studies have suggested that the major reason for this difference is due to the experts' superior comprehension of the program. This paper reports two experiments which investigated expert and novice student programmers program understanding during the debugging process and the hypothesis that the degree of program comprehension is a good predictor of debugging performance. In both experiments subjects used a microcomputer to debug a Pascal program with three logic errors. Subjects' understanding of the program was measured early in the debugging process and again at the conclusion of the debugging session. The results showed that program comprehension plays a vital role in the program debugging process.

1. Introduction

Debugging, the location and correction of program errors after their existence has been established, is a very common programming task. It is estimated that debugging accounts for 25% to 50% of the time in the development of a large computer system (Boehm, 1981; Glass, 1983). Students in programming courses spend a considerable fraction of their programming time tracking down and fixing errors. It is therefore surprising that relatively little is known about debugging and that debugging is only a small part of programming courses.

Almost all debugging studies have concentrated on performance differences between novice and expert programmers. They have shown that experts locate and correct errors faster than novices, find more errors, rarely introduce new errors, make more effective use of the information available, and are much more likely to use an on-line debugger. Typically in these studies the subject task was to locate or locate and correct a single error seeded in the program and the performance measures were whether or not the error was correctly identified and fixed and the time to do so (Gould, 1975; Gould & Drongowski, 1974). Since only the correctness score and time were recorded, these studies provided almost no information about the subject's debugging process - what debugging strategies were employed, how, when, or where they were applied.

However, several recent protocol analysis studies (Jeffries, 1982; Vessey, 1985; Gugerty & Olson, 1986; Nanja & Cook, 1987) have investigated the debugging process of expert and novice programmers. Protocol analysis involves presenting a subject with a problem and asking him or her to "think aloud" while solving the problem (Ericsson & Simon). This verbal protocol is then analyzed in an attempt to obtain a "trace" of the problem solving steps taken by the subject. Through careful analysis, it is often possible to develop a coherent explanation of the problem solving steps by the subject.

The debugging protocol studies showed that experts and novices do many of the same activities, but the experts do them much more effectively. Experts obtained a hierarchical understanding of the program because they read it in the order in which it would be executed while novices read the program like they would read prose - from beginning to end in physical order (Jeffries, 1982; Nanja & Cook, 1987). Novices nearly matched the experts in debugging errors for which there was an error message and a line number; experts, however, were much more successful in correcting logic errors where the only information about the error was the discrepancy between the actual and correct output. Almost all novices and a few experts introduced new errors. The experts immediately undid the errors they introduced while novices did not and had to debug these errors as well (Gugerty & Olson, 1986; Nanja & Cook, 1987).

In these protocol studies subjects' program comprehension seemed to account for most of the debugging performance differences between experts and novices. However, in these studies the degree of program understanding or when the subjects gained the understanding (from an initial study of the program) was not clear. The protocol study reported in this paper addresses these questions. The subject's understanding of the program is measured early in the debugging process and again at the conclusion of the debugging session. For the first measure of understanding a program reconstruction task was administered immediately after the first program modification. The second measure of understanding was a comprehension quiz administered at the conclusion of the debugging session that asked questions to test both low and high understanding of the program. The results indicate that subjects who were the most effective debuggers (located and corrected the errors faster) had a better understanding of the program both after their initial reading of the program and at the conclusion of the debugging session than those who were unsuccessful. Subjects who did not locate and correct all the bugs performed poorly on both the program

reconstruction task and the comprehension quiz.

2. Method

The two experiments investigate the hypothesis that the degree of initial program comprehension is a good predictor of debugging performance. We measured the subjects' initial program comprehension by using program reconstruction task (Shneiderman 1980). The subjects were presented listings of a defective program with three errors, input data file, incorrect output, desired (correct) output, and the program on a micro diskette. The task was to correct the defective program using an Apple Macintosh computer. After their initial program reading comprehension and immediately after their first modification to the program, all program listings and other materials were taken away and they were asked to reconstruct the original program. After the program reconstruction task subjects resumed debugging. Finally, at the conclusion of the debugging session, the subjects were given a nine question comprehension quiz.

The program recall/reconstruction task used in this experiment is different from other similar tasks reported in the literature in four respects. First, subjects recalled a defective program. Second, subjects were not told prior to the experiment that they were to recall a defective program during debugging session. Third, comments in the defective program described the overall function of the program. Fourth, subjects were presented with a program template which contained the declaration part of the defective program.

3. Experiment 1

3.1. SUBJECTS

The subjects in this experiment were six novices and six expert Pascal programmers, all volunteers. The novices were just finishing their second term of a Pascal programming course at Oregon State University. The experts group was composed of graduate students in computer science at Oregon State University. All experts were very experienced student programmers. All subjects had previous experience with MacPascal programming environment on Macintosh computer.

3.2. MATERIALS

The program was a 50-line Pascal program that read in 19 integer data from a file, sorted them in ascending order using bubble sort, and searched for five key values in the sorted list using a binary search routine. It was written in a structured fashion with indenting and meaningful names, contained three lines of comments describing the program, but had no in-line comments. It was identical to the one used in Nanja and Cook (1987) except it was implemented as a monolithic program rather a modular program and seeded with only logic errors, not semantic errors. Since modular programming facilitates program comprehension of subjects by allowing them to concentrate on a small portion of a program and to encode that portion into higher level semantic concepts (chunks) (Shneiderman & Mayer, 1979), we felt that a monolithic program would provide a better test of comprehension. All subjects were familiar with the binary search routine and the bubble sorting algorithm used in the program.

There were three logic errors in the program: (a) off-by-one error -- the number of iterations through a loop is counted incorrectly, (b) assignment statement error -- a variable is assigned incorrect value, and (c) predicate error -- incorrect conditional expression. These errors were selected because they are errors commonly made by students and require thorough understanding of the problem domain.

The listing of the defective program, with three errors, is shown in Appendix A1. Each of these errors could be corrected by modifying only one statement. Correct versions of these statements are shown in the defective program listing as in-line comments.

3.3. PROCEDURE

Subjects performed the debugging task individually. They were given a print-out of the defective table lookup program and a printed copy of input data file, both of which were also available to them on a micro diskette. They were also given a copy of the correct output and incorrect output (see Appendix A2). Subjects were not told how many or what type of errors the program contained or that each error could be repaired by changing only one statement. Further, subjects were informed that they could debug the program at their own pace and use any debugging techniques.

During the debugging session, an experimenter recorded what program objects the subjects were examing and only asked about error hypotheses. For the purpose of this study, we considered the follow-

ing program objects: listing of input data file, printed copy of the expected output, listing of program, Pascal program window, error message window, program output (text) window, and observe window. Activities performed by subjects were categorized as follows:

- examine listing of input data
- examine listing of expected output
- examine program listing on screen
- examine error window
- examine program output window
- examine observe window
- hand simulate program segment (in program listing)
- enter input data
- compare expected output and actual output
- use on-line debugging tool
- run program
- modify program segment

After program modification(s) and before they ran the program subjects were asked to state their hypotheses. The subject's debugging process was recorded as an episode outline his/her activities on program objects. This data gathering procedure was identical to the procedure used in our previous study (Nanja & Cook, 1987).

To measure subjects program understanding, immediately after the first modification to the program, the program and other materials were taken away and the subjects were given 20 minutes to create a program on-line that was as close as possible to the original one. Further, they were instructed not to change the algorithm(s) used in the defective program. The subjects were not told beforehand that a reconstruction task would be administered during the debugging session. For the reconstruction task the subjects began with a program template that contained only the declaration part of the program. The program template is shown in Appendix A3. During their recall task, an experimenter recorded the order in which program statements were recalled, the time spent in each of the three routines of the program, and the total number of references to the declaration part contained in the program template. After the program reconstruction

task subjects resumed their debugging session.

Finally, after the debugging session, subjects were given the correct version of the program and asked to answer a nine question comprehension quiz about the program. The quiz is shown in Appendix A4. The purpose of this test was to measure subjects' understanding of the program at the conclusion of the debugging session.

3.4. RESULTS

Although the defective program was implemented as a monolithic program, it contained three distinct functions: read 19 integer data items from the input file (ReadData), sort all integer data items using bubble sort (BubbleSort), and search for five key values in the sorted list (BinarySearch). The grade for the recalled program was based on separate grading for each function. Two grading schemes, one based on functional correctness of the program and the other based on verbatim recall of the program, were employed.

For functional correctness grading the three functions were broken into a number of parts (chunks), each consisting of one or more program statements and representing a familiar pattern in the specified algorithm. Program parts in each of these functions are:

ReadData function:

- initialize index or count variables
- iterate to read in a set of integer values
- read an integer value and assign its value to an array element
- increment the index variable by one
- count the total number of input values

BubbleSort function:

- iterate to count the number of passes
- iterate array comparison for each pass
- compare two successive array elements
- interchange two array elements

BinarySearch function:

- search for five key values
- read in a key value
- initialize the beginning (low) and the end (high) of the array elements
- iterate search process
- compute the value of the middle index
- test for equality and set appropriate index values
- terminate search process

The functional recall score is the total number of program parts subjects recalled correctly. Table 1 shows for novices and experts the total number of parts recalled, the number of parts recalled correctly, and the number of times the part was missing in the recall. If a subject recalled a different algorithm (e.g. linear search instead of binary search) it was treated as missing all parts. The functional score for each subject is also shown in Table 2

****

insert Table 1 here

****

****

insert Table 2 here

****

As can be seen from Table 1 and Table 2, there were considerable differences between the expert and novice programmers. Experts recalled more parts (mean = 15.5 for experts, and mean = 11.8 for novices), more parts correctly (mean = 13.5 for experts, mean = 8.5 for novices), and had fewer missing parts (mean = 0.5 for experts, and mean = 4.2 for novices) than novices. For each of the three functions experts recalled more parts than novices and a higher percentage of the parts recalled were correct. Both experts and novices performed better in recalling parts contained in ReadData and BubbleSort functions

than in BinarySearch function. Experts recalled nearly twice as many parts of BinarySearch as novices. Furthermore, novices averaged nearly three missing parts in BinarySearch while only one expert missed one part.

For verbatim recall grading, a statement was graded as correct if it was identical to the original except for indentation, and spacing. If subjects used different variable names, changed the statement order, or omitted one or more statements, it was counted as incorrect. All in-line comments in their recall were ignored. In addition, if a subject recalled correctly a corrected version of an incorrect statement (e.g. count := index) of the defective program, it was considered as incorrect. One point was awarded for each correctly recalled statement. A subject's verbatim score was the sum of the points.

****

insert Table 3 here

****

Table 3 shows the total number of program statements recalled verbatim by subjects. As can be seen from this table, expert programmers recalled more program statements verbatim than novice programmers (mean percent = 40.9% for novices, mean percent = 50.9% for experts). Most of this difference was reflected in their recall of the BinarySeach routine. It is interesting to note that both experts and novices recalled the begin-end pairs (syntactic beacons) better than any other statements in the program.

The verbatim score of each subject is shown in Table 4. Surprisingly novices had higher verbatim score than experts in the ReadData function (mean = 4.3 for novices, and mean = 3.5 for experts), but they scored less than experts in the BubbleSort and BinarySearch functions (mean = 3.3 for novices, and mean = 4.3 for experts in BubbleSort function, mean = 6.7 for novices, and mean = 10.0 for experts in BinarySearch function). Note that experts' score is substantially higher than that of novices in the BinarySearch function.

****

insert Table 4 here

****

Both experts and novices began their recall task approximately 6 minutes into the debugging session and, except for BinarySearch function, recalled almost the same number of program statements. In addition, there were little differences in the mean overall recall time between the experts and novices.

The purpose of the post session comprehension quiz was to test subjects' high and low level understanding of the program. There were nine questions in the comprehension quiz, five of which had two parts, and one point was awarded for each correct answer. This yielded a maximum comprehension quiz score of 14 points. Subjects' post session program comprehension quiz score along with recall verbatim score, functional score (total number parts recalled correctly), and number of errors corrected are shown in Table 5. As can be seen from this table novices scored lower than experts (mean = 10.8 for novices, and mean = 12.8 for experts) on the comprehension quiz. Note that the two novices (N1 and N6) who corrected all three errors had the two highest verbatim recall, functional recall, and post session quiz scores among the novices. Notice also that the expert (E3), who failed to correct all three errors had the lowest verbatim, functional, and post session quiz scores among the experts. Correlation coefficient were calculated between the debugging performance of subjects (total number of errors corrected) and program recall performance (functional score, verbatim score), and post comprehension score ($r = 0.77$, $r = 0.82$, and $r = 0.86$, respectively). These relatively high correlation coefficients seem to indicate that subjects' program comprehension is related to their debugging performance.

****

insert Table 5 here

****

The error correction order of experts and novices are shown in Table 6. The four novices (N2, N3, N4, and N5) and one expert (E3), who failed to correct all errors, all failed to find the error in the BinarySearch function. Note that all subjects except E2 immediately verified each modification. E2 modified the ReadData and BubbleSort functions before verifying the modifications were correct.

****

insert Table 6 here

****

4. Experiment 2

4.1. SUBJECTS

A different group of expert and novice programmers at Oregon State University participated in this experiment. There were six subjects in each group. As in the previous experiment, the novices were enrolled in the second sequence of an introductory Pascal programming course, and the experts group was composed of graduate students in computer science department.

4.2. MATERIALS

The program used in this experiment was identical to the one used in the previous experiment (see Appendix A1).

4.3. PROCEDURE

The procedure was identical to the previous experiment, except the reconstruction task was administered 12 minutes into the debugging session and a cloze procedure comprehension quiz was employed at the end of the debugging session. After 12 minutes of program debugging all materials were taken away and a 20 minutes program reconstruction task was administered. After this program reconstruction task subjects resumed their debugging session. Finally, after the debugging session, a cloze procedure comprehension test (Cook, et al, 1983) was given to subjects. The cloze procedure is a "fill-in-missing-parts" procedure. Tokens in the program being tested are systematically deleted and replaced by a blank space. A subject's ability to fill in the blanks is related to the extent to which the program is understood. The cloze procedure version of the program is shown in Appendix A5.

4.4. RESULTS

Since this experiment is almost identical to the previous experiment, the programs recalled by subjects were graded using the same verbatim and functional grading schemes described in the previous experiment. Even though the recall task was administered at end 12 minutes, almost all results were the same or similar to those found in the previous experiment.

****

insert Table 7 here

****

****

insert Table 8 here

****

Functional recall scores for all subjects are shown in Tables 7 and 8. Again as in the previous experiment, experts recalled more parts of each of the three functions than novices (mean = 13.5 for novices, and mean = 15.8 for experts) and more often they recalled them correctly (mean = 10.3 for novices, and mean = 13.1 for experts). Furthermore, experts were much better than novices in recalling parts in the BinarySearch function. Also, note that all experts except E1 recalled all 16 parts of the program, while only two novices (N4 and N6) recalled all parts. Subjects' recall performance was better than those in Experiment 2 because they had more time to study and work with the program.

****

insert Table 9 here

****

****

insert Table 10 here

****

Tables 9 and 10 show the total number of statements recalled verbatim by subjects. Expert programmers recalled more program statements verbatim than novice programmers (mean percent = 48.5 for novices, and mean percent = 59.0% for experts). As in Experiment 1, both experts and novice programmers recalled syntactic beacons better than other program statements and recalled almost the same number of statements (mean = 32 for novices, and mean = 34 for experts).

Subjects' post session program comprehension score is shown in Table 11. One point was awarded for a correct entry in each blank space in the cloze version of the program. This yielded a maximum of 11 points. Novices scored less in their comprehension task than experts (mean = 8.8 for novices, and mean = 9.2 for experts) and took more time to complete the post session comprehension task (i.e. cloze procedure) than experts (mean = 8.3 minutes for novices, and mean = 6.2 minutes for experts). Note that the two novices (N4 and N6) who corrected all three errors had the two highest verbatim, functional, and post session quiz scores among the novices. The only expert (E6) who did not correct all three errors had the lowest post session quiz score among the experts.

****

insert Table 11 here

****

Subjects' error correction order is shown in Table 12. All experts except E6 successfully located and corrected all errors, while only two novices (N4 and N6) corrected all errors. Experts spent more time in reading the program prior to their first modification to the program (mean = 4.8 minutes for novices, and mean = 7.7 minutes for experts). The error correction order of subjects was similar to the one found in the previous experiments.

****

insert Table 12 here

****

Experts were more efficient than novices in debugging the program (mean debug time = 35.0 minutes for novices, and mean debug time = 29.2 minutes for experts). Since other debugging performance measures, debugging strategies, and activities were similar to those found in Experiment 1, they are

not discussed in this section.

5. Discussion

In Experiments 1 and 2 subjects' program comprehension was measured during and after the debugging session. The results shows a strong connection between subjects' program comprehension scores and the number of errors corrected. This corroborates the importance of program comprehension implied in previous debugging studies. The results also replicated many findings obtained in the protocol study reported by Nanja and Cook (1987). For example, expert programmers were more efficient in locating and correcting errors; experts corrected almost all errors and did so at a much faster rate than novices. However, experts' multiple error correction strategy reported in our previous protocol study was not supported by these experimental data. One plausible explanation for this behavior may be due to the fact that only three logic errors and no semantic errors were seeded in the program.

One surprising observation was that all subjects recalled the correct version of the incorrect assignment error in swap part of the BubbleSort routine. A few experts also recalled the off-by-one error in the ReadData routine correctly. It is interesting to note that none of experts or novices included the correct version of the predicate error in the BinarySearch routine in their recall. This seems to indicate that both experts and novices chunk common related statements such as the swap routine and remember them as a unit, rather than treating each statment as an independent unit to be remembered separately. Furthermore they did not seem to realize these statements were incorrect in the program because almost all subjects failed to locate and correct these error statements immediately after their recall task. We do not have a good explanation for their behavior other than that subjects seemed to assume that relatively difficult sections of the code contain the error. All subjects in both experiments initially thought that the BinarySearch function was entirely responsible for the anomalous behavior of the program and first modified one or more statements in this particular function.

Both expert and novice programmers recalled almost the same number of statements in program reconstruction task. This result is inconsistent with large difference reported by Wiedenbeck (Wiedenbeck, 1986), Shneiderman (Shneiderman, 1976), and Shneiderman and Mayer (Shneiderman & Mayer, 1979). However, in the previous studies the program presented to subjects did not contain comments describing

the overall function of the program and for the recall task they were not provided a program template containing the declarations. Our experts were more successful than novices in recalling the entire program. Novices often recalled many redundant statements such begin-end pairs, incorrect statements, and statements that are not relevant to the algorithms used in the program. Furthermore, even though they were told to reconstruct the program, two novices in the first experiment replaced the binary search with a linear search routine in their recall.

As expected, in these experiments, expert programmers' verbatim recall score was superior to that of novice programmers. In both experiments, subjects used a program template provided by the experimenter to reconstruct the original defective program. It is interesting to note the following about their program recall: (a) expert programmers used variable names included in the template more appropriately than novice programmers. For example, almost all experts used variable name "index" as an index variable while novices used it as a variable to store the number of data elements read from the input file. (b) novices frequently declared one new variable name (i.e. boolean : found;) in the program and attempted to use this variable in the BinarySearch routine. Novices, who attempted to recall BinarySearch, recalled the version of the BinarySearch routine given in the textbook prescribed for the class. As a result they recalled a different version of the BinarySearch program. (c) none of the expert programmers attempted to recall a different algorithm other than the ones used in the defective program, while several novice programmers recalled a linear search algorithm instead of binary search algorithm. These observations may partly explain why expert programmers were superior than novice programmers in recalling program statements.

It seems reasonable to conclude that the higher recall scores by both expert and novice programmers in the second experiment were attributable to the additional time spent studying and working with the program before the reconstruction task. On the average the subjects in Experiment 1 began their recall task after their first modification (six minutes into the debugging session), while subjects in Experiment 2 recalled the program at 12 minutes into the program debugging session. This extra time allowed subjects in Experiment 2 to modify, on the average, more program statements than subjects in Experiment 1.

****

insert Table 13 here

****

All subjects in both experiments initially attempted to comprehend the program by reading and studying the program listing of the defective program, input data, expected output, and actual output. Typically expert programmers spent more time in initial reading of the program than novice programmers. This observation is consistent with the finding reported in our previous study. In the previous protocol study, however, three novices, three intermediates, and one expert did not attempt to understand the program through initial reading and immediately ran the program. This can be explained by the type of errors in the program. Both semantic and logic errors were seeded in the program used in the previous study, whereas only logic errors were studied in the experiments reported in this paper. Hence it seemed that the semantic error message along with the line number in the program where the error occurred prompted novices and intermediates to run the program immediately without trying to understand the program.

In the 20-minutes reconstruction task in Experiments 1 and 2, the experimenter gathered information about the order in which subjects recalled individual program statements. During recall all subjects frequently referred to the variable declaration part of the program template. In general experts' recall of the program was smoother as novices frequently modified previously recalled program statements. Typically, all experts recalled a group of statements, paused for a few seconds, and then resumed recalling. This seems to indicate that experts chunk a group of statements in the program and recall them as a single unit. Experts' program chunking was also quite evident from their in-line comments in the recalled program. For example, experts (E2, E5, and E6 in Experiment 1, and E4 in Experiment 2) inserted either a blank line or an in-line comment at the beginning of each function while recalling the program. During recall none of the novices inserted any blank lines or in-line comments between each function in the program. This observation plus novices' modification of previously recalled statements seems to indicate that they do less chunking than experts.

Various debugging measures such as initial reading time of the program, total debug time, and total number of errors corrected were recorded for all experts and novices. Experts and novices spent almost equal amounts of time reading the program prior to their first modification. None of the novices or experts immediately ran the program without doing any initial program reading. Even though two novices (N2 and N5) spent relatively more time than others in initial reading of the program, they failed to correct all errors in the program. All other debugging performance measures (number of runs, number of modifications, and

number of errors introduced) of experts and novices were the same or similar to those reported by Nanja and Cook (1987).

Our previous study found that only experts used on-line debugging aids to trace variables and novices and intermediates inserted debug write statements. In this study novices inserted many debugging write statements inside the loop constructs, while almost all experts used the on-line debugging aids.

In summary, experts and novices performed identical activities during debugging, but experts seemed to carry them out more efficiently and successfully than novices.

6. Conclusions

Previous debugging studies that compared novices and experts inferred that better program comprehension was the major reason for the superior performance of experts. The results of Experiments 1 and 2 demonstrate that experts do obtain a better understanding of the program.

Program understanding may not be that important for locating and correcting semantic errors because the error message with line number information often helps programmers easily identify the cause of the error. But logic errors are more difficult to locate and correct because the only information available about the error is the discrepancy between the actual output and the expected output. Hence for logic errors, it is important that programmers understand not only the overall function of the program but also very detailed statement-level understanding of the program. Hence for debugging logic errors it seems like the level of program understanding clearly separates the expert programmers from novice programmers.

This study also indicated that the primary reasons for the expert programmers' superiority in locating and correcting logic errors was their (a) greater understanding of every program segment, (b) ability to isolate the error to the program segment where the error manifested itself, and (c) ability to select the correct hypothesis. Novice programmers, on the other hand, had difficulty isolating the error to a program segment. Hence they corrected only a few of the logic errors and spent more time debugging the program than experts.

References

- Boehm, B. W. (1981). *Software engineering economics*, Prentice-Hall, Inc..
- Cook, C. R., Bregar, W. S., and Foote, D. (1983). A preliminary investigation of the use of the cloze procedure as a measure of program understanding. *Information Processing & Management*, Vol. 20, No. 1-2, 199-208.
- Ericsson, K. A., and Simon, H. A. (1980). Verbal reports as data. *Psychological Review*, Volume 87, 215-251.
- Glass, R. L. (1982). *Modern programming practices - A report from industry*, Prentice-Hall, Inc..
- Gould, J. D. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7, 151-182.
- Gould, J. D., and Drongowski, P. (1974). An exploratory study of computer program debugging. *Human Factors*, 16, 258-277.
- Gugerty, L., and Olson, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. *Empirical Studies of Programmers*, Soloway, E., and Iyengar, S. (Eds.). Ablex Inc., Norwood, New Jersey, 13-27.
- Jeffries, R. A. (1982). Comparison of debugging behavior of novice and expert programmers. Paper presented at the 1982 meetings of the American Educational Research Association.
- Jeffries, R. A. (1982). Computer program debugging by experts. Paper presented at the 1982 meetings of the Psychonomic Society.
- Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, Volume 23, 459-494.

TABLE 1

Functional score of novices and experts by program parts

Program parts	Novices			Experts		
	Number of parts recalled	Number of parts recalled correctly	Number of missing parts	Number of parts recalled	Number of parts recalled correctly	Number of missing parts
Read Data						
part1	5	4	1	6	6	0
part2	6	5	0	6	6	0
part3	6	4	0	6	6	0
part4	6	6	0	6	6	0
part5	2	1	4	4	3	2
Total	25	20	5	28	27	2
Percent	83%	80%	17%	93%	96%	7%
Bubble Sort						
part1	6	3	0	6	6	0
part2	5	2	1	6	5	0
part3	6	4	0	6	5	0
part4	6	6	0	6	6	0
Total	23	15	1	24	22	0
Percent	96%	65%	4%	100%	92%	0%
Binary Search						
part1	2	2	4	5	5	1
part2	4	4	2	6	6	0
part3	4	2	2	6	6	0
part4	4	3	2	6	4	0
part5	3	4	3	6	5	0
part6	3	0	3	6	2	0
part7	3	1	3	6	4	0
Total	23	16	19	41	32	1
Percent	55%	70%	45%	98%	78%	2%
Overall						
Total	71	51	25	93	81	3
Percent	74%	72%	26%	97%	87%	3%

TABLE 2
Functional score of novices and experts

Subjects	Number of parts recalled	Number of parts recalled correctly	Number of missing parts
Novices			
N1	16	12	0
N2	9	6	7
N3	16	10	0
N4	6	5	10
N5	9	5	7
N6	15	13	1
Total	71	51	25
Mean	11.8	8.5	4.2
Percent	74%	72%	26%
Experts			
E1	16	13	0
E2	15	15	1
E3	15	10	1
E4	16	14	0
E5	16	14	0
E6	15	15	1
Total	93	81	3
Mean	15.5	13.5	0.5
Percent	97%	87%	3%

TABLE 3
Total number of statements recalled verbatim by novices and experts

Program Statement	Novices	Experts
index := 1;	3	0
while not eof(infile) **do**	3	4
begin	6	6
readln(infile, a[index]);	3	2
index := index + 1;	4	2
end;	6	6
count := index;	1	1
for i := 1 **to** count - 1 **do**	2	2
for j := 1 **to** count - 1 **do**	1	2
if a[j] > a[j + 1] **then**	2	4
begin	6	6
temp := a[j];	3	6
a[j + 1] := a[j];	0	0
a[j] := temp;	0	0
end;	6	6
for i := 1 **to** numkey **do**	1	2
begin	3	5
write('key = ');	4	2
readln(key);	4	6
low := 1;	3	6
high := count;	2	2
while low <> high **do**	1	2
begin	3	6
middle := (low + high) **div** 2;	2	5
if key >= a[middle] **then**	1	3
high := middle	2	1
else	3	5
low := middle + 1;	2	2
end;	3	6
if key = a[low] **then**	0	1
arrayindex := low	0	0
else	1	1
arrayindex := 0;	1	0
writeln('key = ', key, ' value = ', arrayindex);	1	0
end;	3	5
Total	86	107
Mean	14.3	17.8
Percent	40.9%	50.9%

TABLE 4
Verbatim score of novices and experts

Subjects	Function name			
	ReadData	BubbleSort	BinarySearch	Total
Novices				
N1	6	5	13	24
N2	5	3	2	10
N3	4	4	10	18
N4	3	3	2	8
N5	3	0	3	6
N6	5	5	10	20
Total	26	20	40	86
Mean	4.3	3.3	6.7	14.3
Experts				
E1	4	4	8	16
E2	3	4	11	18
E3	3	4	4	11
E4	6	5	12	23
E5	0	3	11	14
E6	5	6	14	25
Total	21	26	60	107
Mean	3.5	4.3	10.0	17.8

TABLE 5
Comprehension scores vs debugging performance

Subjects	Verbatim score	Functional score	Post quiz score	Number of errors corrected
Novices				
N1	24	12	14	3
N2	10	6	7	2
N3	18	10	10	1
N4	8	5	11	1
N5	6	5	8	1
N6	20	13	14	3
Total	86	51	64	11
Mean	14.3	8.5	10.7	1.8
Experts				
E1	16	13	14	3
E2	18	15	14	3
E3	11	10	10	2
E4	23	14	12	3
E5	14	14	13	3
E6	25	15	14	3
Total	107	81	77	17
Mean	17.8	13.5	12.8	2.8

TABLE 6
Error correction order

Subjects	Logic error in ReadData	Logic error in BubbleSort	Logic error in BinarySearch
Novices			
N1	1	2	3
N2	2	1	*
N3	*	1	*
N4	1	*	*
N5	*	1	*
N6	3	2	1
Experts			
E1	2	3	1
E2	2	2	1
E3	2	1	*
E4	1	2	3
E5	2	1	3
E6	1	2	3

* ---- did not correct error

TABLE 7

Functional score of novices and experts by program parts

Program parts	Novices			Experts		
	Number of parts recalled	Number of parts recalled correctly	Number of missing parts	Number of parts recalled	Number of parts recalled correctly	Number of missing parts
Read Data						
part1	6	6	0	6	6	0
part2	6	5	0	6	6	0
part3	6	5	0	6	5	0
part4	6	5	0	6	6	0
part5	4	2	2	5	3	1
Total	28	23	2	29	26	1
Percent	93%	82%	7%	97%	90%	3%
Bubble Sort						
part1	5	5	1	6	6	0
part2	6	2	0	6	4	0
part3	6	3	0	6	6	0
part4	6	6	0	6	6	0
Total	23	16	1	24	22	0
Percent	96%	70%	4%	100%	92%	0%
Binary Search						
part1	5	5	1	6	6	0
part2	5	5	1	6	6	0
part3	4	3	2	6	5	0
part4	3	2	3	6	2	0
part5	5	4	1	6	6	0
part6	4	2	2	6	4	0
part7	4	2	2	6	4	0
Total	30	23	12	42	31	0
Percent	71%	77%	29%	100%	74%	0%
Overall						
Total	81	62	15	95	79	1
Percent	84%	77%	16%	99%	84%	1%

TABLE 8
Functional score of novices and experts

Subjects	Number of parts recalled	Number of parts recalled correctly	Number of missing parts
Novices			
N1	14	9	2
N2	14	9	2
N3	11	10	5
N4	16	14	0
N5	10	6	6
N6	16	14	0
Total	81	62	15
Mean	13.5	10.3	2.5
Percent	84%	77%	16%
Experts			
E1	15	11	1
E2	16	13	0
E3	16	15	0
E4	16	13	0
E5	16	15	0
E6	16	12	0
Total	95	79	1
Mean	15.8	13.2	0.2
Percent	99%	84%	1%

TABLE 9

Total number of statements recalled verbatim by novices and experts

Program Statement	Novices	Experts
index := 1;	3	1
while not eof(infile) **do**	5	5
begin	6	6
readln(infile, a[index]);	4	4
index := index + 1;	4	4
end;	6	6
count := index;	2	1
for i := 1 **to** count - 1 **do**	3	4
for j := 1 **to** count - 1 **do**	2	4
if a[j] > a[j + 1] **then**	2	4
begin	6	6
temp := a[j];	6	6
a[j + 1] := a[j];	0	0
a[j] := temp;	0	0
end;	6	6
for i := 1 **to** numkey **do**	2	2
begin	4	6
write('key = ');	4	1
readln(key);	3	6
low := 1;	4	6
high := count;	4	3
while low <> high **do**	1	3
begin	3	6
middle := (low + high) **div** 2;	4	6
if key >= a[middle] **then**	2	2
high := middle	1	3
else	3	3
low := middle + 1;	1	3
end;	3	6
if key = a[low] **then**	2	2
arrayindex := low	0	0
else	2	3
arrayindex := 0;	0	0
writeln('key = ', key, ' value = ', arrayindex);	0	0
end;	4	6
Total	102	124
Mean	17.0	20.7
Percent	48.5%	59.0%

TABLE 10
Verbatim score of novices and experts

Subjects	Function name			
	ReadData	BubbleSort	BinarySearch	Total
Novices				
N1	4	4	3	11
N2	5	4	7	16
N3	4	4	4	12
N4	7	6	15	28
N5	4	2	3	9
N6	6	5	15	26
Total	30	25	47	102
Mean	5.0	4.2	7.8	17.0
Experts				
E1	3	4	6	13
E2	7	4	14	25
E3	4	6	13	23
E4	7	6	14	27
E5	3	6	10	19
E6	3	4	10	17
Total	27	30	67	124
Mean	4.5	5.0	11.2	20.7

TABLE 11

Comprehension scores vs debugging performance

Subjects	Verbatim score	Functional score	Post quiz score	Number of errors corrected
Novices				
N1	11	9	8	1
N2	16	9	9	2
N3	12	10	8	0
N4	28	14	11	3
N5	9	6	7	1
N6	26	14	10	3
Total	102	62	53	10
Mean	17.0	10.3	8.8	1.7
Experts				
E1	13	11	11	3
E2	25	13	9	3
E3	23	15	11	3
E4	27	13	8	3
E5	19	15	9	3
E6	17	12	7	2
Total	124	79	55	17
Mean	20.7	13.2	9.2	2.8

TABLE 12
Error Correction Order

Subjects	Logic error in ReadData	Logic error in BubbleSort	Logic error in BinarySearch
Novices			
N1	*	*	1
N2	*	1	2
N3	*	*	*
N4	1	2	3
N5	*	1	*
N6	3	1	2
Experts			
E1	3	1	2
E2	3	1	2
E3	3	1	2
E4	1	2	3
E5	1	1	2
E6	2	1	*

* —— did not correct error

TABLE 13
Comparison of Experiment 1 and Experiment 2

Subjects' Performance (percent score)	Experiment 2	Experiment 3
Novices		
Verbatim score	41%	49%
Functional score	72%	77%
Post quiz score	76%	80%
Number of errors corrected	61%	56%
Experts		
Verbatim score	51%	59%
Functional score	87%	84%
Post quiz score	92%	83%
Number of errors corrected	94%	94%

Appendix A1

Defective program listing

```
{*************************************************************************}
{          The purpose of the program is to read in a set of              }
{          integer values, sort them in ascending order, and              }
{          then search for certain key values in the sorted list.         }
{*************************************************************************}

program Debug (input, output);
  const
    size = 1000;
    numkey = 5;
  type
    arraytype = array[1..size] of integer;
  var
    t : arraytype;
    i, j, temp, count, index, key, low, high, middle, arrayindex: integer;
    infile : text;
  begin
    showtext;
    reset(infile, 'debug.protocol:data);
    index := 1;
    while not eof(infile) do
      begin
        readln(infile, a[index]);
        index := index + 1;
      end;
    count := index;     { count := index - 1; }
    for i := 1 to count - 1 do
      for j := 1 to count - 1 do
        if a[j] > a[j + 1] then
          begin
            temp := a[j];     { temp := a[j + 1]; }
            a[j + 1] := a[j];
            a[j] := temp;
          end;
    for i := 1 to numkey do
      begin
        write('key = ');
        readln(key);
        low := 1;
        high := count;
        while low <> high do
          begin
            middle := (low + high) div 2;
            if key >= a[middle] then     { if key <= a[middle] then }
              high := middle
            else
              low := middle + 1;
          end;
        if key = a[low] then
```

```
            arrayindex := low
          else
            arrayindex := 0;
        writeln('key = ', key, ' value = ', arrayindex);
      end;
end.
```

Appendix A2

Expected output

key = 4567	value = 19
key = 0	value = 3
key = -5	value = 2
key = 234	value = 0
key = 77	value = 8

Incorrect output

key = 4567	value = 0
key = 0	value = 0
key = -5	value = 0
key = 234	value = 0
key = 77	value = 0

Appendix A3

Program template

```
program Debug (input, output);
  const
    size = 1000;
    numkey = 5;
  type
    arraytype = array[1..size] of integer;
  var
    t : arraytype;
    i, j, temp, count, index, key, low, high, middle, arrayindex: integer;
    infile : text;
  begin
    showtext;
    reset(infile, 'debug.protocol:data);
  end.
```

Appendix A4

Postsession quiz

- What type of sort and search routines are used in this program? Circle appropriate responses.

  Sort: 1. selection sort 2. bubble sort 3. insertion sort
  Search: 1. linear search 2. binary search 3. quadratic search

- If the statement number 17 was changed to "**read(infile, a[index]);**", what would happen during program execution? (Be specific)

- If the statement number 23 was changed to "**if a[j] < a[j+1] then**", what values would the program print out for each of these input key values?

  key = 4567 value = _____
  key = 77 value = _____

- If the statement number 22 was changed to "**for j := count-1 downto i do**", would the results of the program be changed?

  Yes or No. Briefly explain why or why not.

- If the statement number 37 was changed to "**middle := (low+high) / 2;**", what would happen during program execution? (Be specific)

- If the statement number 38 was changed to "**if key < a[middle] then**", what values would the program print out for each of these input key values?

key = 4567 value = _____
key = 670 value = _____

- For each of these input key values, what values will be printed out?

key = 33 value = _____
key = 999 value = _____

- In order to get these output values, what input key values would you type in?

value = 12 key = _____
value = 0 key = _____

- How would you modify this program to search for 7 key values?

Appendix A5

Cloze procedure

```
01 program Debug (input, output);
02   const
03     size = 1000;
04     numkey = 5;
05   type
06     arraytype = array[1..size] of integer;
07   var
08     t : arraytype;
09     i, j, temp, count, index, key, low, high, middle, arrayindex: integer;
10     infile : text;
11   begin
12     showtext;
13     reset(infile, 'debug.protocol:data);
14     index := 1;
15     while _____ do
16       begin
17         readln(infile, a[index]);
18         index := _____ ;
19       end;
20     count := _____ ;
21     for i := 1 to _____ do
22       for j := 1 to _____ do
23         if _____ then
24           begin
25             temp := a[j + 1];
26             a[j + 1] := a[j];
27             a[j] := temp;
28           end;
29     for i := 1 to numkey do
30       begin
31         write('key = ');
32         readln(key);
33         low := 1;
34         high := count;
35         while _____ do
36           begin
37             middle := (low + high) _____ ;
38             if _____ then
39               high := middle
40             else
41               low := _____ ;
42           end;
43           if _____ then
44             arrayindex := low
45           else
46             arrayindex := 0;
47         writeln('key = ', key, ' value = ', arrayindex);
48       end;
49   end.
```