TEACHING DEBUGGING

by

Robert F. Mathis
Department of Computer and Information Science
The Ohio State University

ABSTRACT

A course in debugging techniques is motivated and described. A course outline, reading list, and projects list are included. Certain debugging aids are described. Debugging techniques for elementary algorithms are illustrated. Particular attention is paid to ways to teach debugging and algorithm structure.

Program testing and debugging occupies more than 50% of professional programmers' time. Almost every survey or estimate points out this area of program development as time consumming, difficult, poorly planned, and often slighted. Program debugging also consumes a large part of the time in an introductory programming course. Out of fifty or more computer runs during a term, the average student usually has only four or five runs which are even close enough to correct to hand in. But there seems to be no formal instruction on debugging or guidelines in the text or even any good folklore on how to do it. A student either learns to debug his own programs somehow or he finds somebody who can help him. Considering the advances which have been made in teaching programming, debugging instruction is still in the dark ages.

Currently debugging is a real art form and it is done in an almost magical way. Dump reading is almost always described in some variation of, "I just keep comparing fields until one catches my eye and then I have a place to start." Students and teachers are encouraged to minimize errors by having good work habits, being methodical and neat, and checking for clerical details.[1] This is certainly good advice, but it doesn't really help when it comes to fixing a problem. Structured programming and programmer/team management are both ideas which developed in an attempt to write programs with fewer errors to start with. These are important concepts and should be discussed; but once again, they are of minimal help when it comes to correcting an error. Some of the most useful (from a teaching standpoint) recent developments in compilers and programming systems have been the addition of better compile and run time diagnostics. These have made program debugging significantly easier. In the assembly language area, there are some processors which make teaching and using assembly language easier (e.g. Waterloo's ASSEMBLER G, Penn State's ASSIST, and Ohio State's Baum-Silverman interpreter). The common and most visible advantage of these processors is a simplified post mortem dump. The increasing use of these processors reinforces my belief that the most difficult part of assembly language programming is not the use of the language itself but the almost exclusive reliance on dumps for debugging. Even students in our advanced systems courses have difficulty when confronted with a dump. I have therefore included in the course some formal instruction in using dumps and other debugging aids. One of the assignments I use in our beginning systems course is to write a dumping routine. This gives the student a better understanding of the system and also makes him think about what information he would like to have upon termination of a program.

To see if something could be done in teaching debugging, I began making specific debugging related assignments in programming courses, having students work on individual special projects related to debugging, and finally teaching a course about debugging in general. This paper describes some of the aspects of that course.

The course is divided into three main parts - one, discussion of existing debugging tools and techniques; two, literature on proposed tools, test methods, and program verification; and three, student projects.

Debugging Course Outline

1. Program development phases
2. Writing better programs
   - A. Modularization
   - B. Structured programming
   - C. Standards
   - D. Management and technical problems
   - E. Documentation
3. Types of bugs/mistakes
4. Debugging in general
5. Debugging and other issues
   - A. General trade-offs
   - B. Hardware problems
   - C. Program correctness
   - D. Performance evaluation
   - E. Auditing
   - F. Security

6. Algorithm design for debugging
7. Programming for debugging
8. Programmer built-in aids
   A. Intermediate output
   B. Module interrelation
   C. Assertion checking
   D. Computation checks
   E. System features for facilitating
9. System supplied aids-text processing
   A. Compiler diagnostics
   B. Cross reference list
   C. Standards checkers
   D. Auto flowcharters
   E. Auto documentation
10. System supplied aids-during execution
   A. Preprocessors
   B. Execution monitors
   C. Co-resident
   E. Interactive
   F. Traces
   G. History keeping and processing
   H. Dumps
   I. Abend trapping
   J. Test executives
   K. Module testers
   L. Test generators
11. Propossed aids
   A. Hardware/software changes
   B. Shadow task
   C. Abend analyzers
   D. Dumps
   E. Abend traps programming language
   F. Program Control Block
   G. Error Analysis Control Block
   H. Dump analysis
   I. TSO dump reader
   J. Virtual memory abend processing
   K. Event monitoring use
   L. Programming and debugging system
12. Post execution debugging
   A. Practical methods
   B. Theoretical investigation
13. Aids for error types
   A. Keypunching
   B. Data Structures
   C. Numeric calculation
   D. Control flow
   E. Loop control
   F. Decision/branch tables
   G. Simulation
   H. Input validation
   I. Storage modification
14. Specific debugging aids
   A. PDP-10 DDT
   B. OS/MVT SYSUDUMP
   C. TSO TEST
   D. Fortran Interactive Debug
   E. Cobol Interactive Debug
   F. Pl/I Checkout and Optimizer
   G. WATFOR-WATFIV-WATBOL-PL/C
15. Case Studies
16. Projects
17. Debugging strategies
18. Review and summary

The published literature on program debugging is very small. There are three books which are relevant[3,4,5], and two published bibliographies[6,7].

An important part of the course, and of the teaching of programming and debugging in general, are the student projects. Some of the projects have involved mainly the study of various existing programming systems and their debugging facilities but the more important ones have dealt directly with the building of debugging aids and tools. Some of the projects are

1. custom dumping routines
2. data structure outputting
3. dump analyzers
4. execution monitors
5. interpreters
6. programming aids
7. program tracers
8. automatic generation of test data
9. systematic error causers
10. testing the equivalence of programs
11. incremental compiling
12. incremental modification of programs

Many of these projects are also usable in more general programming courses.

The course has been mainly oriented toward techniques which are useful in debugging programs which have not been written in any special way. There are also techniques for writing programs in a way which will make them easier to debug. These techniques should be discussed in all programming courses.

Programming standards and methodologies, modular programming, and structured programming are all techniques which help in writing programs with fewer bugs to begin with. These techniques should be used and should be taught in elementary programming courses. As an outgrowth of the course on debugging, certain techniques became apparent which make a program easier to debug. These center around internal checking and intermediate output. The simple algorithms used in programming courses should not only show program development but also proper practices and debugging techniques.

In an elementary course we usually begin with a simple algorithm and develop it completely. First we describe the process in words and then flowchart it in one form or another. Then we teach enough of a programming language to implement this algorithm. In particular the first example in the book by Forsythe, Keenan, Organick, and Stenberg, Computer Science: A First Course[2], is the computation of the first term in a Fibonacci sequence which exceeds one thousand (their Figure 1-10). Figure 1 shows the flowchart for this algorithm from their book. Figure 2 shows a simple FORTRAN (WATFIV) implementation of this algorithm. Usually this is as far as we go. We may use the algorithm to lead into more complicated algorithms or to illustrate other FORTRAN concepts, but usually we discuss one algorithm after another - the computational and data processing concepts involved and their implementation. The computer programs which implement these algorithms must be assumed to be perfect since there is never any discussion of possible errors in implementation.

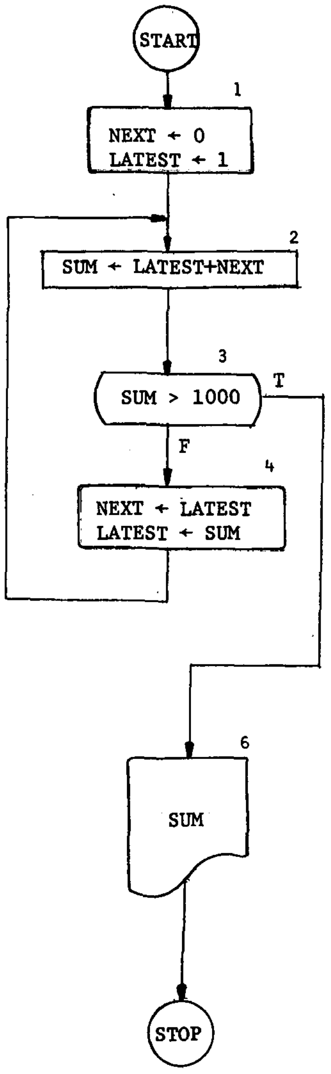

Figure #1

```
1  NEXT=0
   LATEST=1
2  SUM=LATEST+NEXT
3  IF(SUM-1000)4,4,6
4  NEXT=LATEST
   LATEST=SUM
   GOTO 2
6  PRINT,SUM
   STOP
   END
```

Figure #2

Consider the problems of translating the flowchart in Figure 1 to a computer program. Figure 2 shows a simple implementation. What statements would we add to check this implementation? The authors themselves give one hint. They discuss this algorithm in detail by tracing its execution step by step through the flowchart boxes. In the computer implementation we can achieve a similar result by placing the statement "PRINT, NEXT, LATEST, SUM"after the FORTRAN statements which correspond to each box in the flow chart. To make the output a little clearer, we could add a sequence number (NSEQ) and a statement number label. We also need to add special output statements for the results of the test. The expanded version of this program is illustrated in Figure 3. The added statements appear as comments except for the two statements to which the IF test might transfer.

```
C     SUM=0
C     NSEQ=1
 10   NEXT=0
      LATEST=1
C     PRINT,NSEQ,'1',NEXT,LATEST,SUM
C     NSEQ=NSEQ+1
 20   SUM=LATEST+NEXT
C     PRINT,NSEQ,'2',NEXT,LATEST,SUM
C     NSEQ=NSEQ+1
 30   IF(SUM-1000)41,41,61
 41   PRINT,NSEQ,'3','TEST FALSE'
C     NSEQ=NSEQ+1
 40   NEXT=LATEST
      LATEST=SUM
C     PRINT,NSEQ,'4',NEXT,LATEST,SUM
C     NSEQ=NSEQ+1
      GOTO 20
 61   PRINT,NSEQ,'3','TEST TRUE'
      PRINT,SUM
      STOP
      END
```

Figure #3

In this example, the debugging statements are self-checking in the sense that they are output statements with specific expected results and they involve no computations which might affect the primary statements of the algorithm. They would probably be excessive if they were all used at once. Using them all would help in understanding the algorithm and it is better to let the computer play computer than tracing the whole program by hand. Inclusion of all these intermediate output statements as comments also shows where in the program diagnostic output might be useful. Quite often an algorithm is designed and improperly implemented and there is no information on where to request diagnostic output.

A common problem with debugging aids like traces is that they generate too much useless output. The only reason to completely trace all variables and statements, as in the preceeding example, is to gain some understanding of the inner workings of the algorithm. Usually there is some smaller aspect of the program which needs monitoring - the result of certain tests or the modification of certain variables. The problem is to determine just what information would be useful in debugging a program. Consider the following example of a sort routine. (The two sorting algorithms discussed in the following are from figures 3-29 and 4-48 of the Forsythe, et.al. text.) The algorithm in Figure 4 makes very little change in the list for each "pass". It would be foolish to print the entire list each time there was an interchange. Similarly, the new values assigned to variables would be uninformative. The most useful thing here is to use this algorithm as an exercise in intermediate output and debugging aids design. My own suggestion would be some kind of output following box 5 to indicate the subscript of items being interchanged and occassionally print the part of the list which has changed. The frequency of printing parts of the list is one of the problem areas in designing temporary output statements to debug this algorithm.

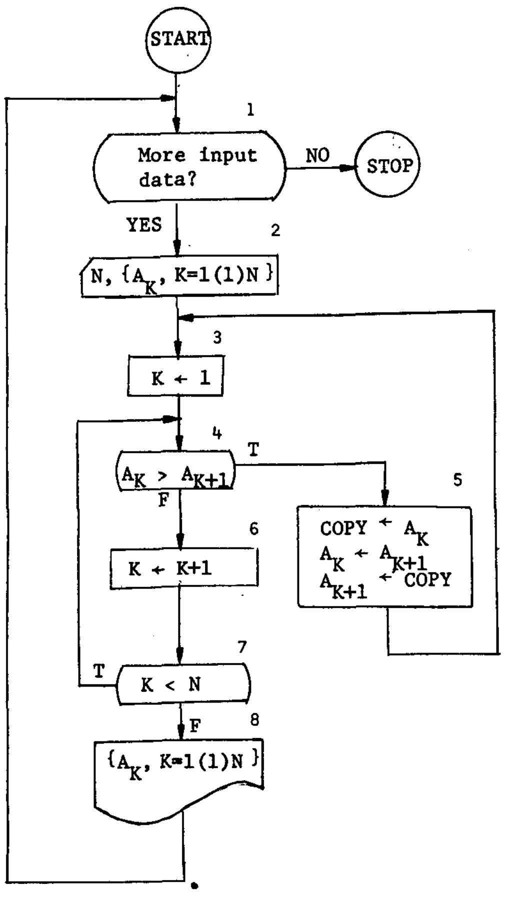

Figure #4

Since the primary activity of a sorting routine is to change a data structure, the place to start is with a routine to print that data structure and to indicate changes in it. For a vector, or linear array, of numbers this is relatively easy. The list can be printed by a single statement

```
PRINT,('A(',K,')=',A(K),K=1,N)
```

Two versions of the list can be easily compared and the differences printed:

```
      DO 100 I=1,N
      IF(AOLD(I).EQ.A(I))GOTO 100
      PRINT,I,AOLD(I),A(I)
      AOLD(I)=A(I)
  100 CONTINUE
```

Here AOLD is a copy of the privious version of the list being sorted. AOLD would need to be initialized at the beginning of the routine. It is automatically updated as differences are printed. This kind of output routine can be well used in another sorting algorithm like the one in Figure 5, the shuttle-interchange sort. This algorithm makes considerable changes during each pass but they are all similar. One item is moved up and the intervening elements are each moved down one place.

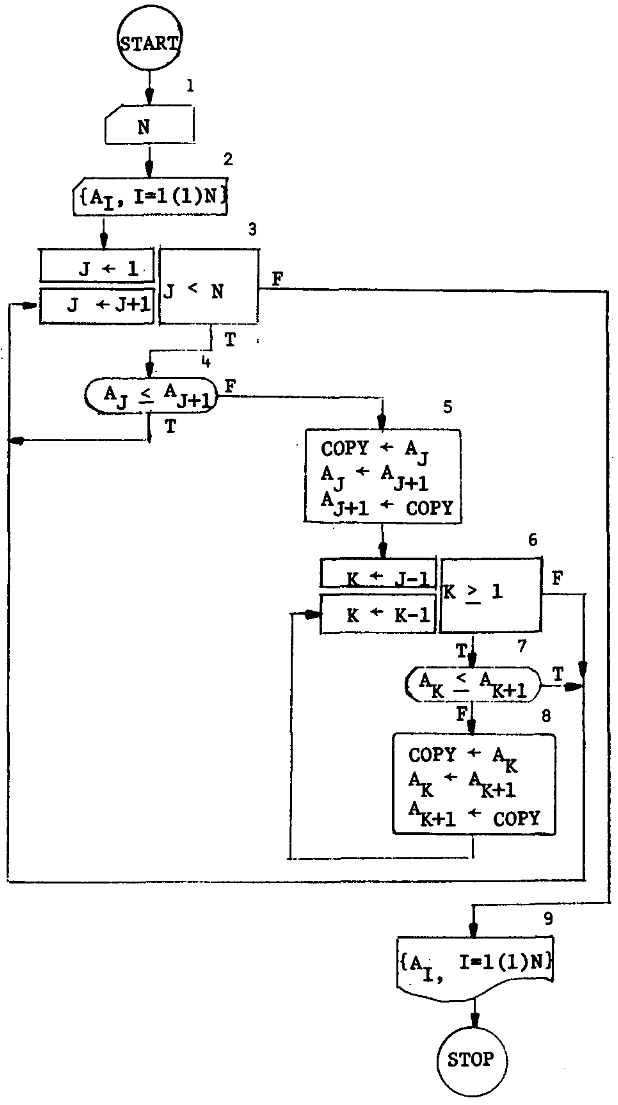

Figure #5

Rather than printing all the changed values, we mostly want to know--one, the initial and final subscripts of the item which moved up in the list; two, that nothing was changed above or below these points; and three, that the items in between each moved down one place. In all cases the design of intermediate output and debugging aids leads to a closer investigation and better understanding of the working of the algorithm.

I hope that these examples have shown that we can teach some debugging concepts and a better understanding of the algorithms by designing intermediate output routines for some of the common elementary algorithms we use in our courses.

This debugging course is an attempt to relate various concepts and techniques of programming so that the students can more easily produce correct working programs. The contents and approach of the course are still under development. Additional reports of results, a more complete bibliography, and reports from some of the student projects will be available in the future.

Footnotes

1. Forsythe, A.I., Keenan, T.A. Organick, E.I., and Stenberg, W., Computer Science: Teacher's Commentary, John Wiley and Sons, New York, 1969.

2. Forsythe, A.I., Keenan, T.A., Organick, E.I., and Stenberg, W., Computer Science: A First Course, John Wiley and Sons, New York, 1969

3. Rustin, Randall, Editor, Debugging Techniques in Large Systems, Prentice-Hall, Englewood Cliffs, N.J., 1971.

4. Brown, A.R. and Sampson, W.A., Program Debugging, American Elsevier, New York, 1973.

5. Hetzel, William C., Editor, Program Test Methods, Prentice-Hall, Englewood Cliffs, N.J., 1973.

6. Kocher, W., A Survey of Current Debugging Concepts, NASA Report, August, 1969.

7. Kosy, D., Annotated Bibliography of Debugging Testing and Validation Techniques for Computer Programs, Rand Corporation, Santa Monita, California, WN-7271-PR, January, 1971.