AN EXPERIMENTAL COMPUTER SCIENCE PROBLEM SEMINAR

by

Ronald Alter and Thaddeus B. Curtz

University of Kentucky
Department of Computer Science
Lexington, Kentucky 40506

I. INTRODUCTION AND SUMMARY

In conjunction with the development of a new Master's degree program, yet to receive final approval, the University of Kentucky Computer Science Department has devised and implemented a problem seminar. Students who enter graduate training in the department are required to present four of six specified items of academic background. It has been our experience that they present these credentials but have little working knowledge of their relation to Computer Science.

The seminar is therefore designed to encourage the student to use and apply certain techniques he is expected already to know. On the other hand, we make no great point of this with the students - the problems are presented to them (usually) without comment on the expected method of solution; although the background, importance and relevance of the problem is discussed.

We have found the results obtained by offering this seminar so encouraging we now propose to incorporate it as a requirement in the Computer Science graduate curriculum and it is also actively under consideration for inclusion as part of an interdisciplinary program on applied science now being developed. This creates a serious problem in staffing; obviously a course of this kind requires a high faculty-student ratio. At present, the seminar is a two hour credit course, however, we believe the students would be better served by a three hour credit course.

In what follows we discuss the objectives of our seminar, how it is organized to attain these objectives, and finally our plans for the future development of the course. We conclude by speculating briefly how we might best employ a third classroom hour.

II Background Information

The Computer Science program at the University of Kentucky has existed since 1965. Due to a variety of circumstances, principally the tragic death of the first chairman of the department, Silvio Navarro, in an airplane crash in 1967 - the program suffered a major setback. The University renewed its commitment to Computer Science with an active recruiting program in 1969 and at this point is committed in principle to the implementation of a Master's degree program

in the near future. We are at present offering a small selection of graduate courses and are provisionally accepting graduate student enrollment. Much of the undergraduate program closely resembles ACM curriculum '68 as is only to be expected in view of Navarro's close connection with the development of that curriculum.

On the other hand, we are in the process of developing a conviction that the single course most useful to the majority of our students is one not mentioned in curriculum '68 and not specifically incorporated in many of the curricula from other institutions. This is rather surprising in view of the importance attached to the rather traditional problem seminar in the mathematics curriculum. The computer science seminar which is the subject of this paper is described in some detail in the next section; for the moment let us merely say that in a setting as closely resembling typical industrial practice as we can create we pose to the student a variety of problems for computer solution.

We have experimented with this seminar on the advanced undergraduate level and more recently as a graduate course. Originally we introduced it as a means of acclimating our undergraduate students to "non-spoon-fed" problem solving using a computer. Most of our Bachelor's degree candidates seek employment as programmers and they clearly benefit greatly from the kind of experience which being led to develop their own resources offers. Of course, upper level undergraduates should program well, nevertheless it has been our experience that they sometimes do not do so. A second major benefit of the seminar is improved programming capability of the students.

On the other hand it is at the graduate level that the authors have been most impressed by and interested in the results obtained by offering the course. We believe the benefits derived by graduate students from a seminar of this kind are substantial enough to justify the faculty-student ratio implicit in offering this course - a ratio of more than 1:6 appears to us to be self-defeating.

In addition to the independence and the gain in programming skill cited above as the principal objectives in offering a course of this kind to undergraduates, graduate students derive another major benefit and several less important ones. First, as will be seen below, we attempt to design the problems posed for graduate students to meet a set of carefully defined specifications the most important one of which is the exercise of certain skills learned but rarely applied as undergraduates.

Other benefits include the usual advantages of any seminar and more particularly the following:

1) The seminar provides an opportunity to focus on the importance of documentation. We make an effort to inculcate the principles both of commenting programs and properly describing them. This forms an important part of the student's grade and we have gone so far as assigning no grade to an undocumented program.

2) We make a conscious effort to develop a mode of thinking aimed at inducing a preference for a computational solution. In the past ten years the relative cost of an analytic solution and a computational one has changed by several orders of magnitude. While the analytic effort has remained fixed, cycle times have gone from milliseconds to nanoseconds. This change will no doubt continue and we believe it to be so major as to be a change in kind rather than in degree. Thus we feel students should be taught to prefer (all other things being equal) a computed solution.

3) Some of the problems are difficult enough so that a student may hope that he can develop a thesis topic as an extension of a problem encountered during the seminar.

A natural question is "How do we accomplish all this?" No doubt we are hoping for a lot - perhaps for too much - from a single course. Yet we find the results encouraging. The organization we use to accomplish these results is described in the next section.

III DESCRIPTION OF THE PRESENT SEMINAR

As a basic and minimal requirement for admission to graduate training we have specified a knowledge of mathematics through the calculus and skill in the use of a problem oriented language. Beyond these requirements, to enter the graduate program without deficiencies a student is required to present qualification in the form of successful completion of a course in four items from among the following six (no significance should be attached to order):

A) Numerical analysis
B) Computer organization and programming
C) Linear algebra
D) Probability and statistics
E) Logic or logical design of computers
F) Discrete mathematics or data structures

This is neither a severe, nor an unusual requirement. In our experience so far, however, students who present these qualifications (and even those whose grade point average in these courses normally leaves little to be desired) are far removed from being able to apply these diverse disciplines to Computer Science.

The problem seminar has as its primary objective the confronting of students with problems susceptible to meaningful computational solution in a "free format". We do our best to convey to the students the impression that we are interested in a solution and that the method is secondary. However we have a list of personal criteria for the inclusion of a problem and desire for a solution as such does not appear on this list.

Ingenuity is required to formulate problems with the properties that they are:

1) Accessible - i.e posing a reasonable level of difficulty - neither trivial nor too hard. As will be seen we prefer problems which can be solved in a week or so of effort.

2) Computational in nature.

3) Relevant to one of the courses whose content we believe forms a basic element in the background of knowledge any computer scientist should have.

4) Interesting.

All these criteria should be satisfied; nevertheless, frequently one or perhaps more of the criteria are abandoned. When this happens, it is (3) that usually goes first. We believe that a good problem satisfying the other criteria deserves inclusion - especially if it is in the important area of nonumerical applications of computers.

As the seminar is presently organized we meet twice a week; the first session is devoted to presentation of new problems and faculty discussion of problem status, the second to student presentation of results and status reports. About 3 problems per week are posed; possible grades range from zero through three points. A minimal performance standard is defined for the students; in a 14 week semester we require the accumulation of 18 points. The seminar is taught by the 2 authors; typically, next semester we will have 8-10 students. Obviously this is an expensive way of teaching, but we find the progress made by typical students very encouraging.

Not only do we find improvement in student self-discipline and in programming skill during the semester. We are very pleased by the insight students develop as to the comparative merits of various solutions. The students develop a healthy scepticism as to the claims of the designers of special purpose software, in fact, lively discussion as to the merits of this or that processor are by no means unusual. They develop a harsh pragmatic view of the merits of computing software. Interestingly they tend to measure merit by execution speed rather than programming ease. There is a breed of student who will go to nearly any length to demonstrate the advantages of, say, PL1 over FORTRAN, or SNOBOL over either. This can be particularly interesting when, in fact, the putative advantages do not exist.

As an occasional sop to the weaker students, we find it necessary to include in the list of problems a certain number of straight forward and easy ones. Generally however, we prefer to err on the side of being too demanding. We are often surprised by how well students can do with a difficult problem; it is our hope that, as time goes by, we can expect a fair number of problems to develop into theses.

As an example of a typical problem, meeting all of the criteria listed above and interesting because of the number of different ways of attacking it developed by the students we offer the following: (old mathematical hands will recognize the Polya-Szego, "Aufgaben und Lehrsatz" flavor of this particular example).

The number of ways of making change

Description: Given a system of coinage with a stated set of denominations, how do you calculate the number of different ways to change a coin?

Suppose we look first at a simple example. "How many different ways can you change 15 cents into pennies, nickels, and dimes?" A little thought (or some scrambling in the pocket if you happen to have the right 19 coins), shows that in this particular case the answer is 6 ways.

$$15 = 15 \times 1 + 0 \times 5 + 0 \times 10$$
$$15 = 10 \times 1 + 1 \times 5 + 0 \times 10$$
$$15 = 5 \times 1 + 2 \times 5 + 0 \times 10$$
$$15 = 5 \times 1 + 0 \times 5 + 1 \times 10$$
$$15 = 0 \times 1 + 1 \times 5 + 1 \times 10$$
$$15 = 0 \times 1 + 3 \times 5 + 0 \times 10$$

If you try to generalize this counting scheme you will soon recognize that the question can be posed as, "How many solutions in nonnegative integers does the diophantine equation:

$$n = x + 5y + 10z,$$

have? More generally still we can ask the same question for

$$n = d_1x_1 + d_2x_2 + \cdots + d_mx_m,$$

if the d_j are the denominations available to change a coin of denomination n. We have generalized the problem in two different ways. On the one hand, we are treating an arbitrary number of denominations. On the other, we have allowed the coin to be changed to have an arbitrary value. Looked at from this point of view, one can say we are prepared to try to answer the question, "How many different ways can one combine a given set of denominations to arrive at a specified sum?"

This is a very old problem. The earliest references to it we are aware of date from 1669. (G.W. Leibniz asked J. Bernoulli in a letter if he had investigated the number of ways a given number can be separated into two, three or many parts, and remarked that the problem was difficult but important.) Really it would be very disappointing if there were not a nice analytic "solution". By a "solution" we do not mean a closed form expression whose value is the desired count, rather in this case, we mean a formulation which makes the necessary counting a relatively simple matter. We will discuss a classical way of computing the answer to our question but let us first try to

formulate the problem for a digital computer without further analysis. This problem will yield to sheer tenacity. On first being confronted with it, many students write a program which simply exhausts all of a large set of possible combinations and keeps count of the number of successes. This leads to the statement of several problems:

1) Write a program to compute the direct count for a reasonable range of values of n, m, and the d_j.

2) If you wrote a computer program to look at every possibility, how many possibilities will be examined for the values n = 1000, $d_1 = 1$, $d_2 = 5$, $d_3 = 10$, $d_4 = 25$, $d_5 = 50$, $d_6 = 100$, $d_7 = 300$? (The reason for the inclusion of d_7 is that anyone who uses this method to count the number of ways of changing ten dollars is the sort of person who is likely to accept a three dollar bill.)

3) How can you better perform this calculation? Make the observation that the problem can be solved recursively. That is, one way of looking at it is to consider the number of ways of making change on lower levels. It is important to notice that from the programmer's point of view, what this amounts to is writing a subroutine that has the capability of calling itself, an action which is forbidden in Fortran; but for which most modern higher level languages make provision. An Algol solution to the program follows.

```
'BEGIN'
'INTEGER' 'PROCEDURE' CHANGE (AMOUNT, DENOMS, U).,
'COMMENT' PROCEDURE CHANGE RECURSIVELY COUNTS THE NUMBER OF
   WAYS TO MAKE CHANGE OF AMOUNT INTO DENOMINATIONS DENOMS(/1..U/).
   DENOMS MUST BE IN ASCENDING ORDER.  .,
'VALUE' AMOUNT, U.,
'INTEGER' AMOUNT,U.,
'INTEGER' 'ARRAY' DENOMS.,
'BEGIN'
'INTEGER' I, CHNG, NUM, AMPRI., ·
'COMMENT' CHNG IS A TEMPORARY VARIABLE WHICH COUNTS THE NUMBER
   OF WAYS TO MAKE CHANGE AT THE END OF THE PROCEDURE. NUM IS
   MORE OR LESS AMOUNT '/' DENOMS(/U/). AMPRI IS USED TO DECIDE
   WHETHER DENOMS(/U/) DIVIDES AMOUNT .  .,
CHNG . = 0.,
'COMMENT' IF AMOUNT IS LESS THAN THE SMALLEST DENOMINATION THERE
   IS NO WAY TO MAKE CHANGE.  .,
'IF' AMOUNT 'LESS' DENOMS(/1/) 'THEN' 'GOTO' QUIT.,
   NUM . = AMOUNT '/' DENOMS(/U/).,
AMPRI . = NUM x DENOMS(/U/).,
'COMMENT' IF THERE IS ONLY 1 DENOMINATION LEFT AND IT DOES NOT
   DIVIDE AMOUNT, THERE IS NO WAY TO MAKE CHANGE. IF IT DIVIDES
   THERE IS ONE WAY TO MAKE CHANGE.  .,
'IF' U 'EQUAL' 1 'THEN'
   'BEGIN'
      'IF' AMPRI 'EQUAL' AMOUNT 'THEN' CHNG . = 1.,
      'GOTO' QUIT.,
   'END' OF THEN CLAUSE.,
'COMMENT' IF NONE OF THE TERMINATING CONDITIONS MENTIONED IN THE
      LAST TWO COMMENTS HOLDS, THE NUMBER OF WAYS TO MAKE CHANGE
      IS THE SUM WHILE 0 'NOTGREATER' I 'NOTGREATER' U OF CHANGE
      (AMOUNT - I x DENOMS(/U/), DENOMS, U-1). WHEN DENOMS(/U/)
      DIVIDES AMOUNT, THIS CORRESPONDS TO ALL CHANGE COINS BEING
      THE SAME (LARGEST) DENOMINATION. THIS SITUATION IS COUNTED
      AS A SEPARATE CASE RATHER THAN RECURSIVELY.  .,
'IF' AMPRI 'EQUAL' AMOUNT 'THEN'
```

```
                'BEGIN'
                    CHNG . = 1.,
                    NUM . = NUM-1.,
                'END' OF THEN CLAUSE.,
                'FOR' I . = 0 'STEP' 1 'UNTIL' NUM 'DO'
                    CHNG . = CHNG + CHANGE (AMOUNT - I x DENOMS(/U/), DENOMS, U-1).,
QUIT..      CHANGE . = CHNG.,
                'END' OF CHANGE.,
                    'INTEGER' AMOUNT, DIM.,
                    'COMMENT' AMOUNT IS AMOUNT TO BE CHANGED, DIM IS NUMBER OF
                        CHANGE DENOMINATIONS.  DIM MUST BE NONNEGATIVE.  .,
READ..              ININTEGER (0, AMOUNT).,
                    'COMMENT'  NONPOSITIVE AMOUNT IS END OF DATA SIGNAL.  .,
                    'IF' AMOUNT ' NOTGREATER' 0 'THEN' 'GOTO' QUIT.,
                    ININTEGER (0, DIM).,
                    'IF' DIM 'LESS' 1 'THEN' 'GOTO' ERROR.,
                    'BEGIN'
                        'INTEGER' J.,
                        'INTEGER' 'ARRAY' DENOMS(/1..DIM/).,
                        'COMMENT' DENOMS, THE ARRAY OF CHANGE DENOMINATIONS,
                            MUST BE NONNEGATIVE AND IN ASCENDING ORDER.  .,
                        INTARRAY(0, DENOMS).,
                    'IF' DENOMS(/1/) 'NOTGREATER' 0 'THEN' 'GOTO' ERROR.,
                    'FOR' J . = 1 'STEP' 1 'UNTIL'
                        DIM-1 'DO'
                        'IF' DENOMS(/J/) 'NOTLESS' DENOMS(/J + 1/) 'THEN' 'GOTO'
                            ERROR.,
                    OUTSTRING (1, '(' THE NUMBER OF WAYS TO MAKE CHANGE')').,
                    OUTINTEGER (1, AMOUNT).,
                    OUTSTRING(1, '(' INTO DENOMINATIONS')').,
                    OUTARRAY(1, DENOMS).,
                    OUTSTRING(1,'(' IS')').,
                    OUTINTEGER(1, CHANGE(AMOUNT, DENOMS, DIM).,
                    SYSACT(1, 14, 1).,
                        'GOTO' READ.,
                    'END' OF INNER BLOCK.,
ERROR..             OUTSTRING(1. '(' BAD DIM OR DENOMS.  SEE COMMENTS. ')').,
                    SYSACT(1, 14, 1).,
                    'GOTO' READ.,
QUIT..          'END' OF PGM.,
```

Program by Lee Crawfort

Finally let us take a brief look at the classical method of solving the problem. We would like to have all of the different ways that a positive integer n can be written as a sum of positive integers chosen from a given set S. This is a problem in the theory of partitions. The partition function p(n) is defined as the number of ways that the positive integer n can be written as a sum of positive integers. Two partitions are considered to be the same if they differ only in the order of their summands. Other partition functions can be defined for which the summands satisfy certain restrictions. If $p_m(n)$ is the number of partitions of n into summands less than or equal to m and we make the convention that

$$p_m(0) = p(0) = 1,$$

the following theorem can be proven.

Theorem The number of partitions of n into m summands is the same as the number of partitions of n having largest summand m. The number of partitions of n into at most m summands is $p_m(n)$.

Many results concerning the partition function depend on the theory of analytic functions, however for our purposes there is an elementary technique that is quite suitable, namely the theory of generating functions. In a nutshell, the basic idea underlying the use of generating functions is that given a sequence of numbers that it is desirable to study, the manipulation of the formal power series whose coefficients are the given sequence, is very informative.

With regard to the partition functions it can be shown in a rather elementary way that

$$\sum_{n=0}^{\infty} p_m(n)x^n = \prod_{n=1}^{m} (1-x^n)^{-1}$$

and

$$\sum_{n=0}^{\infty} p(n)x^n = \prod_{n=1}^{\infty} (1-x^n)^{-1}.$$

As an example of the significance of these formulas, suppose now we want to change our original 15 cents into pennies, nickels and dimes. The number of ways we can do this is (according to the above) the coefficient of x^{15} in the series expansion of $\frac{1}{1-x^1} \times \frac{1}{1-x^5} \times \frac{1}{1-x^{10}}$.

This can be determined from

$$\frac{1}{1-x^{10}} = 1 + x^{10} + \text{----}$$

$$\frac{1}{1-x^5} = 1 + x^5 + x^{15} + \text{----}$$

$$\frac{1}{1-x} = 1 + x + x^2 + \text{----}$$

Clearly the x^{15} term in the final product is easily seen to have coefficient 6.

For a typical example of a problem that does not have property (3), yet is relevant and important enough to include, consider the problem of rotating a shaded shape.

Shaded Shape Rotation

Discussion: Computer graphics is playing a major role in the field of computer applications today. The next problem does not use the computer to design or construct something, nevertheless, it is related to computer graphics. The idea is to take a particular figure or shape and rotate it through a fixed number of degrees. The shape can be made by printing x's where desired. It can be made darker by printing, say, y's on top of the x's. By performing various tricks like these, one can shade the shape in almost any desired way. In rotating the shape, a problem to be aware of is truncation. That is, if you rotate the shape out of range, a section will be truncated. Distortion is another problem.

Problem: Given a shaded shape, write a program which will rotate the shape x degrees. Thus if the shape is as in

Figure 1

Figure 1, your result will look like Figure 2 after being rotated through x degrees.

Figure 2

The user should be able to select whether the original shape and/or the resultant shape is to be printed. The user may also specify the approximate positions on the page for the shapes to be printed.

IV. PLANS FOR FUTURE DEVELOPMENT

Though we are pleased with the results of offering the problem seminar there are nearly as many questions remaining unresolved as toits best organization as we have already answered. Conditions in our University and in the academic world are changing radically and two of the questions we consider to be most important are directly connected with these changes. One of these is the staffing problem mentioned above - we are anxiously seeking ways to handle a larger number of students in a format similar to the present one without losing the apparent benefits of informal organization and cordial working relationship offered by the present arrangement.

Secondly, we would like to develop a technique for feeding back into our own undergraduate curriculum some information based on the students capabilities and shortcomings as displayed in the seminar. Since a major rationale for the present course is the student's need for practice in applying basic skills taught in other courses, the teaching in such courses might reasonably be influenced by observation of the results we obtain.

Finally there are a couple of less important but nevertheless interesting possibilities with which we now plan to experiment. As previously mentioned, we will give three hours of credit next semester. At present we are considering three possibilities as to best employ another hour. Rather than simply multiply by 1 1/2 what we believe is already a sufficiently demanding problem-solving work load we prefer either to require that each student learn and employ unfamiliar language to solve one or more of the problems, (SNOBOL, ALGOL, LISP, etc. are all good candidates for the purpose) or, more traditionally, to require that each student read and report on a paper selected for its relevance to one of the problems.

A third possibility, that each student select, early in the course and with the instructor's approval, a major programming project, has also been considered. This project would have to be completed during the semester in addition to the assigned problems.

Of the three possibilities, our inclination is in favor of the latter two, primarily because they seem to us to develop the student's potential for research. That is, either presenting a paper or undertaking a major programming project could be a stepping stone to a research topic for a Master's thesis.

Also, though we already place a modest emphasis on efficient programming, we are considering distributing a budget of computer dollars to each student with which he must make do for the entire semester.

Finally, we welcome comments and criticism on the seminar. In particular we would like to hear from those who may already have developed a similar course. We would also be most gratified if readers would attract to our attention any appropriate problems which could be used in the context described above; problem selection without repetition poses a serious and continuing dilemma for the authors!

ACKNOWLEDGEMENT

The authors are grateful to the students who have taken this course and to the Computer Science Department faculty who have contributed problems from time to time. In particular we would like to thank our colleagues Professors A. C. R. Newbery, H. C. Thacher and Mrs. K. Nooning and our students L. Crawfort, D. Gravitt, J. Hall and E. Ryan.