Automated problem solving for the behavioral sciences*

B. JAMES STARR†
Howard University, Washington, D.C. 20001

In view of the obvious advantages of computers for the behavioral sciences, the question is raised as to how to make more effective use of computing capabilities. One idea is to require that fledgling behavioral scientists receive brief training in flowcharting and algorithm generation rather than a full course in computing skills. Such training would be an important adjunct for those who would go on for further instruction, especially in view of current deficiencies in many computer courses. One recurrent deficiency lies in the lack of specific guidelines for conceptualizing problems for computer implementation. Such guidelines are developed here, along with other suggestions designed to attenuate the difficulty in conceptualization.

The advent of behavioral science applications of computers has offered the capability for greatly facilitating the work of the applied researcher. Electronic data processing allows a fairly straightforward assessment of a complex of variables. A morass of data can be rendered meaningful quickly and efficiently. Computer simulation of human processes offers interesting research possibilities.

Although the behavioral sciences have witnessed increased training in computer skills in recent years, the majority of researchers still see their activities as divorced from computing. Data analysis via a few popular "canned" programs accounts for most computer usage. There are difficulties in attempted communication with key data processing personnel. This lack of understanding engenders confusion and blocks the development of mutually beneficial relationships between researchers and programmers.

Clearly, an intimate knowledge of computer functioning is desirable for many researchers. Still, a full semester's course on computer applications may not be practicable for most researchers. Furthermore, in the author's experience, such courses and the textbooks used in them are seriously deficient in aiding the student in the conceptualization stage of automated problem solving. The textbooks (e.g., Lehman & Bailey, 1968) generally provide little in the way of guidelines for such conceptualization. Brief general training in formulating computer problems (via an explicit protocol) and communicating the results, e.g., by flowcharting (see Chapin, 1970), should enhance the researcher's likelihood of efficiently achieving the desired computer output. The suggestion here is that researchers should learn to develop algorithms (i.e., modes of problem solution). Procedural guidelines and flowcharting principles can be integrated as one segment of some noncomputer course, e.g., research methods. The object of this approach is to develop skill in communicating with computer center personnel. Moreover, a tangible benefit accrues to those seeking further computer training: They will have already learned how to resolve some of the difficulties involved in conceptualizing problems for automated solution.

The purpose of this paper is to set forth some guidelines for use in the formulation stages of automated problem solving. It is hoped that these guidelines will be applied to the wide variety of problems in behavioral research which could benefit from the speed and accuracy of computer processing. A task recently completed by the author (Starr, 1969b, 1970) underscored the value of the suggested scheme. He was asked to program a problem about which he knew very little. The work was vastly simplified by consulting with someone who understood the problem and was skilled in breaking it down.

WRITING ALGORITHMS

An algorithm is an unambiguous set of instructions on how to go about solving a problem (Morsund, 1969). Thus, many types of instructional media qualify as algorithms: recipes, game instructions, assembly instructions for a new barbecue grill, bookshelf, or other unassembled object. The key word in the definition of algorithm is "unambiguous." If we were dealing with recipes, for example, the instruction, "Bake a cake," would not qualify as an algorithm. It leaves too many questions unanswered. What ingredients should be used? In what proportions should they be used? Obviously, the following instructions are not much better: "The ingredients are flour, shortening, sugar, eggs, and milk; mix them together in a pan; put the pan in the oven."

The above example suggests certain general ideas which could prove helpful to an individual developing an algorithm.

(1) *Algorithms are language-bound in the sense that we must take account of the limitations of the language to be used in implementing them.* An algorithm could be written in any language set—English, Greek, ALGOL, or idiosyncratic symbolese. The important thing here is that the person who develops the algorithm should have an accurate perception of the implementer and his language set. In preparing a recipe, it is important to know how cooks generally work and how they communicate regarding their work (i.e., how recipes are generally written). Therefore, those who would write algorithms for computer implementation should have some knowledge of how computers work[5] and how programmers communicate (e.g., flowcharting practices). Thus, one must realize that the computer truncates (destroys) all but a set number of significant digits. When obtaining means on survey data, for example, small numbers at the end of the lengthy summation will likely be lost.

(2) *All instructions should be so clearly stated that the implementer need know nothing beyond the implementation language.* A common failure in algorithms developed by novices centers on their assumption of the implementer's boundless knowledge. For example, the fledgling programmer may write "calculate the mean" when he means "sum N scores" and "divide the sum by N." Computers (and some humans!) do not know what the more global instruction means until it has been defined in terms of computer "do-able" processes. The final instructions should be sufficiently elemental to be compatible with the step-by-step nature of computer functioning.

(3) *The individual who develops the algorithm should know how to solve the problem at hand.* This is obvious. Few people would develop a recipe if they had not cooked the dish themselves (or at least seen it done).

(4) *The problem solver should attempt to work out a general solution for the exercise.* While it is not necessary to adhere to this guideline, it

*This is an expanded version of a paper presented at the XVIIth International Congress of Applied Psychology, Liège, Belgium, July 25-30, 1971.

†The author is indebted to William H. Churchill and Wendell Joice for their helpful suggestions in response to earlier drafts of this manuscript.

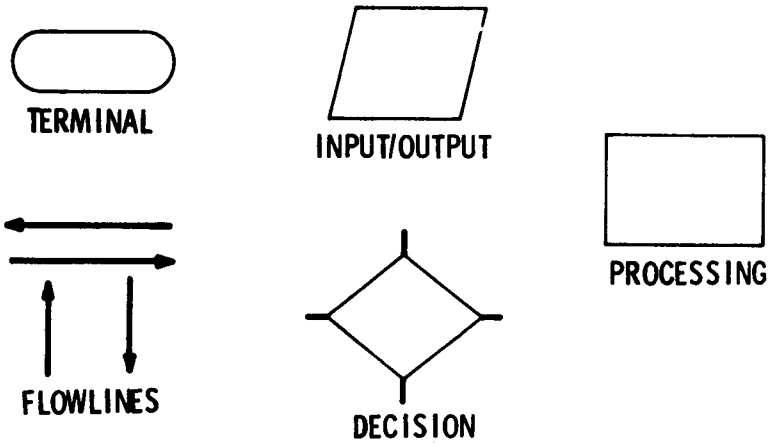

Fig. 1. Some of the major outlines suggested for use by the American National Standards Institute for creating flow diagrams.

is helpful to do so. Because algorithms require time and effort for their development, an efficient algorithm would provide the solution to a class of problems (as opposed to a single problem). Thus, a program which computes autonomic lability scores from physiological data (see Starr, 1969a) could be structured to handle *any* type of physiological data for experimental groups of virtually any size. Moreover, when a problem is conceived in its full generality abstracted from the details of a particular case, its solution sometimes turns out to be surprisingly simple.

SPECIFIC GUIDELINES FOR ALGORITHM GENERATION

With the preliminary notions outlined above in mind, the development of a specific protocol for evolving an unambiguous instruction set is now possible. Because the most common mode of instruction in algorithms involves simulation of the instructor, an explicit formulation has the advantage of being less dependent on both his communication skills and on the perceptivity of the students.

(1) *A good beginning for problem solution may be made by working out a concrete example and carefully noting all steps involved in arriving at the correct outcome.* Essentially, this is a self-simulation technique whose success depends on an orderly and *elemental* set of procedures. (In order to be appropriately elemental in his thinking, the individual should have some knowledge of computer operation, e.g., how comparisons are made, etc.) Here the important information (see Lehman & Bailey, 1968) regarding what items must be present for processing (i.e., input), the form in which these items must be entered (i.e., how they are to be read in), and the nature of the outcome (i.e., desired output) are obtained. Clearly, this type of thought process is applicable to the generation of many varieties of algorithms.

(2) *All input, processing, and output variables and constants should be tabled; any changes in the values of the variables from the start of the problem to its finish should be noted.* The tabling procedure would provide information on initialization (the starting values for processing), loop parameters (values indicating the length and stepwise nature of the processing), and, generally, the logic of the step-by-step process. A fairly clear picture of actual changes in variables inside the computer should be reflected in the table.

(3) *The procedures, variables, and constants from the initial steps should be translated into the language of the recipient.* This step would involve the production of an accurate flow diagram (or relatively language-free graphic representation) or written computer code. Such a program segment should adhere to the strictures imposed by both the communication media (e.g., FORTRAN) and the nature of computer functioning.

(4) *An independent example should be worked out using only the procedures embodied in the results of Step 3.* This is primarily intended as a preliminary "debugging" (i.e., error eliminating) aid. Obviously, an example sufficiently different from the original one will also provide a test of the generality of the algorithm for the particular class of problem involved.

(5) *The above step should be repeated until an error-free execution of the problem is obtained.* This guideline involves modifying the procedures in Step 3 as needed to obtain a relatively "bug-free" (errorless) general solution. It also gives explicit recognition to the fact that a completed algorithm represents the culmination of a series of successive approximations to problem solution. In addition, special cases likely to cause trouble might appropriately be considered at this point. Thus, if one were generating an algorithm to compute correlation coefficients, for example, one might consider the difficulties created by a variable with a standard deviation of zero. Clearly, some contingency must be built into the program to circumvent attempts to divide by zero.

Of course, any instruction set designed for computer use is not a completed algorithm until it has actually been successfully implemented on the computer. The above guidelines are suggested to increase the probability of successful implementation.

Classically, algorithm development (programming) has been taught via simulation of a skilled individual (instructor or author). This technique is of dubious efficiency in the absence of explicit procedures such as those outlined here. In the subsequent two sections, some general principles of flowcharting are presented, along with an example comprising the guidelines stated above.

FLOWCHARTING

Some of the major outlines used in creating flow diagrams are presented in Fig. 1. These outlines are taken from the standard flowchart symbols of the American National Standards Institute (1970).[1] The purpose of the flow diagram is to portray graphically the procedures involved in processing information in a specific way. Thus, it can be an efficient mode for the researcher (qua problem solver) to communicate with the programmer.

The symbols shown in Fig. 1 are only a small subset of the standard symbols, but they will suffice for the understanding of many basic flow diagrams. The elliptical terminal outline is used at the start and the termination of program segments (i.e., algorithms). The flowlines indicate the direction of the logical flow from one outline to another. Usually the flow diagram is read from top to bottom and left to right. When this typical pattern is violated, open arrowheads (as shown in Fig. 1) must be employed to show the change appropriately. The parallelogrammic input/output (I/O) outline is used to depict any read or write operation (regardless of the medium, i.e., cards, tape, paper). The diamond-shaped decision outline is used for branching, the direction being determined by the value of the expression (or variable) enclosed. Finally, the rectangle defines a general processing (e.g., arithmetic) operation.

It is one of the most common outlines in flow diagrams.

A SPECIFIC EXAMPLE OF ALGORITHM GENERATION: THE ARITHMETIC MODE

There exists a large class of problems which are easily unraveled without the use of computers. Often the programming involved is quite complex, and there is some question about whether efforts toward computerizing the exercise are worthwhile. Plainly stated, such an effort could amount to inefficient use of the programmer's time. This is a happy confluence of facts for the type of tutorial paper presented here. Problems of the class indicated above have the advantage of being easily understood, while at the same time being sufficiently complex (in terms of programming) to enable the novice to work with some fairly sophisticated concepts.

One example of this type of problem might involve finding the modal value in a set of scores. The author has had his students attempt this as part of a first project designed to derive measures of central tendency from a set of data. The mode is defined as the score value which appears with the greatest frequency in a distribution of scores. In order to simplify the program segment and because programming the median (computed in the central tendency program) involves rank-ordered data, the numbers used here will already have been ranked. Algorithm generation is, of course, an art. The solution offered below is only one among a number of possible solutions of varying efficiency. The ranking procedure will not be discussed. Instead, the procedures offered by McCracken (1965, p. 70) and Lehman & Bailey (1968, p. 134) are recommended.

A knowledge of computer functioning involves the realization that comparisons can be made only in a pairwise fashion (i.e., the first number is compared to the second, etc.). Self-simulation (the first specific guideline) might comprise the following activity. In comparing the first number to the second, a tally mark may be made if there are equal scores. The second number is then compared to the third one and again a tally mark is made if equality is found. As tallies appear in the working of the problem, they may be recorded elsewhere as a number. For example, two tally marks indicate three equal numbers (say, the first, second, and third ranked numbers). The score and its frequency are recorded. Whenever the stepwise comparison process reveals an inequality, previous tallies are erased and started anew. The modal score and its frequency are not erased until a score with a higher (or equal) frequency of occurrence is found. If an equally frequent score is found, it is placed in a subsequent slot in the modal *score* column and the length of this column is recorded. When a score with greater frequency than the current modal score is found, everything may be erased and the new mode and its frequency are recorded. This basic procedure may be repeated until there are insufficient scores remaining to create a new mode—even if all the remaining scores are equal. This last requirement must be met because a computer would have to process the remaining data completely, where a man could determine quickly that there were no further equalities.

Table 1
Analysis of Hypothetical Data for Use in the Calculation of the Mode

Verbal Labels							
Scores	Parameters for Comparing Loop		Temporary Frequency	Permanent Frequency	Mode Vector	Size of Mode Vector	Value to Stop C Loop
FORTRAN Labels							
X	L	K	ITALLY	NHI	AMD	Length	N-L
2.2	1	2	2	2	2.2	1	10
2.2	2	3	3	3	2.2	1	9
2.2	3	4	1	3	2.2	1	8
3.1	4	5	1	3	2.2	1	7
4.6	5	6	2	3	2.2	1	6
4.6	6	7	3	3	4.6	2	5
4.6	7	8	1	3	4.6	2	4
5.0	8	9	2	3	4.6	2	3
5.0	9	10	3	3	5.0	3	2
5.0	10	11	4	4	5.0	1	1
5.0							

An example of the type of tabling suggested in Guideline 2 is shown in Table 1. It depicts both verbal labels and possible FORTRAN language labels. The first column contains the actual example distribution of scores. Columns 2 and 3 contain the values needed to compare two scores at a time in an orderly fashion. The ITALLY column records the frequency of appearance of the score currently under scrutiny. The subsequent two columns contain the highest frequency found thus far and the score with which this frequency is associated (i.e., the current mode). The LENGTH column gives the length of the mode column. The last column supplies information used to stop comparisons when further search would be fruitless. The numbers appearing in the body of the table are the values the variables would take on if the problem were solved in the manner suggested.

The third guideline suggests generation of a flow diagram in keeping with the procedures developed in the earlier steps and observing good flowcharting practices. Such a flow diagram appears in Fig. 2. At the start, L is set to "1" and K to "2," indicating that the first two scores in memory should be compared. Since each score will have a frequency of at least 1, both the current tally variable (ITALLY) and the highest frequency variable (i.e., the frequency of the current modal value or NHI) are set to "1." The length of the modal vector is set to "0," and "I," which will be used in writing the results, is set to "1." The first two numbers are compared, and ITALLY is incremented by "1" if there is an equality. If there is no equality, ITALLY must be reset to "1," because it is possible that it is already greater than 1 (from an earlier comparison). An inequality, then, leads to resetting of the comparison loop counters, L and K, a check as to whether further comparisons are necessary, and, if necessary, a new comparison. Further comparisons will be unnecessary if the number of comparisons remaining is less than or equal to NHI and the current comparison involved an inequality.

Following the discovery of an equality and the increment of ITALLY, the value of the highest frequency thus far is subtracted from ITALLY. A negative result leads to an increment in the comparison loop counters and the subsequent procedures outlined above. A zero result indicates a current multimodal situation. Here the length of the mode vector would be incremented by one and the appropriate score stored before L and K were incremented. Finally, if the subtraction of NHI from ITALLY yields a positive number, a new single mode has been discovered. Therefore, the length of the modal vector would be set to "1," NHI would be made equal to ITALLY, and a new mode would be stored in the first position on the mode vector

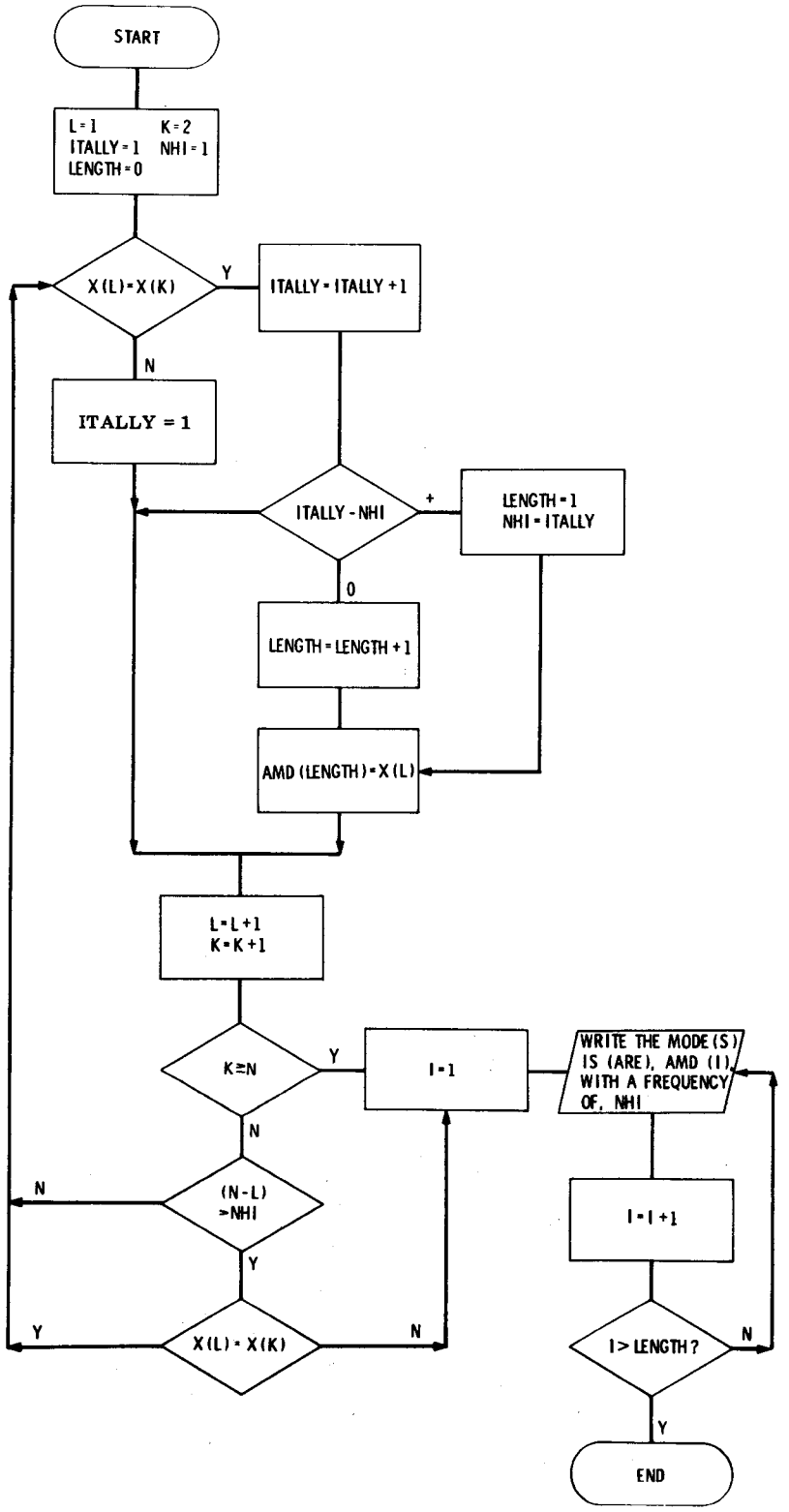

Fig. 2. Flow diagram for program segment selecting the mode or modes from a set of data. Legend: L, K = parameters involved in comparison; ITALLY = frequency of the score currently under scrutiny; NHI = highest frequency found thus far; LENGTH = parameter indicating length of the mode vector (for multimodal distributions); AMD = score *value* of the mode(s).

before L and K were incremented. Upon completion of the requisite number of comparisons, the mode(s) would be written out.

Only one step (or series of steps) would now remain. The procedures embodied in the flow diagram would be tested on an independent example. This will not be done here.

CONCLUSIONS

An algorithmic approach to many behavioral science research problems should prove fruitful. It is hoped that the specific guidelines and general notions offered in this paper can provide the structure necessary for a fairly rapid introduction to this approach. The example employed here, while trivial, should have the advantage of being easily followed by novices. Their simulation of the principles outlined in this paper may lead to the more efficient use of an important resource.

Hopefully, the method advocated here will not only provide an important "entrance" skill for those who would go on to become programmers, but it would also enhance (with relatively brief training) basic communications skills for those researchers who might make use of computing facilities. The researcher who is able to conceptualize and depict the steps toward the solution of the problem will have valuable tools for interacting with computer programmers.

REFERENCES

CHAPIN, N. Flowcharting with the ANSI standard: A tutorial. Computing Surveys, 1970, 2, 119-146.

LEHMAN, R. S., & BAILEY, D. E. *Digital computing: Fortran IV and its applications in behavioral science.* New York: Wiley, 1968.

McCRACKEN, D. D. *A guide to Fortran IV programming.* New York: Wiley, 1965.

MORSUND, D. G. *How computers do it.* Belmont, Calif: Wadsworth, 1969.

STARR, B. J. A program for the computation of autonomic lability scores. Behavioral Science, 1969a, 14, 169 (CPA 323).

STARR, B. J. S-POOL: A program for keeping track of participation in psychological experiments. Behavioral Science, 1969b, 14, 252-253 (CPA 328).

STARR, B. J. Computerizing the activities of subjects for psychological research at a large university. Behavior Research Methods & Instrumentation, 1970, 2, 29-31.

NOTE

1. See Chapin (1970) for a more complete exposition on flowcharting standards.