COMPUTER METHODS IN OPERATIONS RESEARCH

ARNE THESEN

COMPUTER METHODS IN OPERATIONS RESEARCH

ARNE THESEN

DEPARTMENT OF INDUSTRIAL ENGINEERING
UNIVERSITY OF WISCONSIN
MADISON, WISCONSIN

ACADEMIC PRESS New York San Francisco London 1978

A Subsidiary of Harcourt Brace Jovanovich, Publishers

ACADEMIC PRESS, INC.
111 Fifth Avenue, New York, New York 10003

United Kingdom Edition published by
ACADEMIC PRESS, INC. (LONDON) LTD.
24/28 Oval Road, London NW1

Library of Congress Cataloging in Publication Data

Thesen, Arne.
Computer methods in operations research.

(Operations research and industrial engineering series)
Includes bibliographies.
1. Operations research–Data processing.
I. Title.
T57.6.T47 001.4'24'02854 77-74063
ISBN 0–12–686150–1

PRINTED IN THE UNITED STATES OF AMERICA

PREFACE

This text is designed to fill the growing gap between the computational requirements emerging in modern industrial engineering and operations research (IE/OR) applications and the algorithmic and computational methods commonly available to the IE/OR student and practitioner.

As an illustration of this gap, consider the conventional labeling or "scratch pad" approaches to shortest or longest (i.e., critical path method) path network problems discussed in introductory OR texts. These techniques work well for hand calculation where a visual representation of the network is available. They also work in computer programs designed to handle small problems where manual procedures can be mimicked in an exact manner. However, substantial additional considerations enter when programs are to be written that handle larger than illustrative problems.

These considerations include issues such as: (1) What errors are likely to occur in the input phase? How can they be avoided, detected, or corrected? (2) What are the different ways available to represent the network in the computer? (3) What is the most efficient algorithm for the problem at hand? (4) What errors are likely to occur during the analysis phase? How can they be avoided or detected and possibly corrected? (5) How should the performance of the resulting program be evaluated? While these issues are

rightfully ignored both in introductory level computer programming courses and in introductory level OR courses, they are important issues of substantial economic significance that the practicing systems analyst cannot afford to ignore.

The academic level of a course based on this book would be suitable for a senior or first-year graduate student in engineering or business. Prerequisites for this course would include a course in FORTRAN programming and a course in deterministic OR models (including linear programming (LP) and network analysis). The course itself could be useful as a corequisite for more advanced courses in graph theory or network analysis as well as a useful prerequisite for a thorough course in simulation techniques.

In Chapter I we provide a review of some of the basic principles that make a software development effort successful. Throughout this chapter the need to keep things simple and understandable is stressed. The computer is a servant, not a master. The development of a code that is not used because the potential user does not believe or understand it is a wasted effort. Considerable attention is also devoted in Chapter I to the subject of software evaluation.

Chapters II and III cover the basic principles of list processing, searching, and sorting. Such subjects are normally included in second or third level computer science courses. However, these courses are usually too specialized and have too many prerequisites to be available to the anticipated audience of this textbook.

In Chapter IV the concept of networks is introduced and several matrix and list oriented methods for representing networks in the computer are discussed. Techniques for spanning all nodes and/or links in a network are then developed.

The critical path method (CPM) is discussed in Chapter V. We show how to develop efficient CPM algorithms. We also investigate the problem of designing CPM packages as computer-based management tools. Guidelines are developed for handling the detection of input errors and for the design of reports. The needs of the user (rather than the computer or the analyst) are the major concern here.

Chapter VI presents more complex programs and algorithms to handle scheduling of activities under precedence and resource restrictions. To illustrate these approaches, the resource-constrained scheduling problem is formulated both in an exact (using integer programming) and in a heuristic manner. The difficulties of implementing these approaches in a computer are discussed. Finally, a complete commercial scheduling package for managerial use is reviewed in detail. This chapter also serves as a vehicle for an in-depth discussion of many aspects of program evaluation.

The design of algorithms for the solution of large linear programming problems is discussed in Chapter VII. A simple review is given of the formu-

lation and solution of LP problems, using the revised simplex method. This discussion is followed by a review of how various algorithmic aspects, such as the handling of sparse matrices, pricing, and the updating of the inverse, are implemented so that the resulting usage of computer resources is minimized. This chapter also contains a brief overview of available mathematical programming systems, and the mathematical programming systems data input format is reviewed in detail. Finally, the chapter contains a discussion of the concepts of generalized upper bounds as implemented in the MPSIII system designed by Management Science Systems, Inc. (Permission to reprint their discussion on this subject is greatly appreciated.)

The application of list processing concepts to the development of branch and bound algorithms for solution of combinatorial optimization problems is discussed in Chapter VIII. Design considerations for branch and bound algorithms are given. An illustration is presented to demonstrate how sorting and list processing can be used as the two basic techniques to solve multidimensional knapsack problems efficiently.

The design of pseudorandom number generators is discussed in Chapter IX. We first show how uniformly distributed pseudorandom numbers can be generated, using multiplicative congruential methods. Values of appropriate constants for several different computers are given. This discussion is followed by a review of four different tests for checking the acceptability of pseudorandom number generators. Finally, the generation of random deviates from the exponential, Erlang, normal, chi-square, and Poisson distributions is discussed.

Chapters X and XI are concerned with discrete event simulation studies. The discussion of fundamental modeling and programming aspects in Chapter X is centered around a simple data structure and a few rudimentary subroutines designed to carry out the essential steps in any discrete event simulation program. Chapter XI is concerned with simulation modeling, using formal simulation languages. Discussions are presented here on two widely different languages. GPSS (the main simulation language in use in the United States today) and WIDES (a much simpler language especially designed for inexperienced users).

ACKNOWLEDGMENTS

I wish to acknowledge the labors and influences of a few friends and colleagues without whom this text would not have materialized. Dr. E. L. Murphree of Sage, a consulting cooperative in Champaign, Illinois, introduced me to many of the subjects in the text, and, more notably, his contagious curiosity and ability to kindle creative thinking have had a monumental impact on my professional growth. The present form of this manuscript is the result of many hundreds of hours of stimulating interaction with students using several earlier drafts of this text. In particular, I wish to acknowledge the efforts of S. Miller, E. Hobbs, F. Cheung, and L. Strauss for weeding out many annoying errors in earlier drafts. The assistance of Ms. Lynda Parker in editing an early draft of the text is also appreciated. Finally, Mrs. Doreen Marquart is to be thanked for a patient and highly competent job in typing all drafts as well as the final manuscript.

CONTENTS

CHAPTER I

CONSIDERATIONS IN PROGRAM DESIGN AND EVALUATION

A. INTRODUCTION

Although the code in a FORTRAN program must follow very strict syntactic rules, the act of designing such a program is a very creative process subject to few if any formal rules. This has resulted in the development of highly individualistic styles of program development and design. Some of these styles are effective; others are not. Some do not work at all.

Our intent in this chapter is to identify the key attributes of "good" programs and "good" programming practices. In addition, we will suggest several programming procedures and guidelines that may reduce the time and effort required to write a program and may also improve the quality of the end product. However, we are dealing with empirical guidelines; what works for us, may or may not work for you.

The process of writing complex programs requires many considerations and decisions. Of primary concern is the reduction of the time and effort to be expended in the programming task so as to release time for the design and development of the algorithms which your program represents.

B. PROGRAM DEVELOPMENT PROCESS

As shown in Fig. I-1, the program development process starts with the user recognizing a problem and ends with his interpreting a computer printout that may provide relevant information and/or answers. The success or failure of a particular programming effort is primarily determined by the degree to which this printout assists the user in resolving his problem.

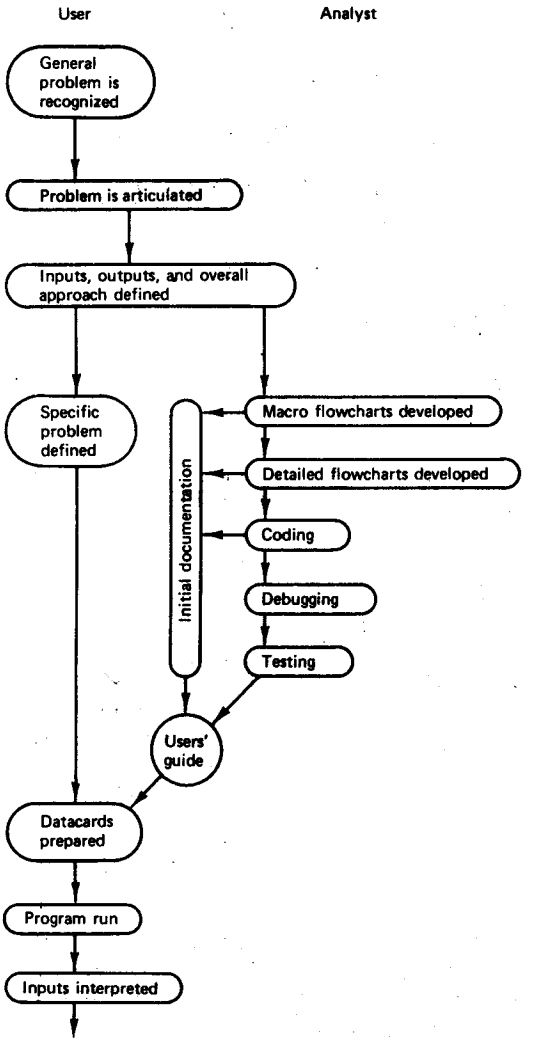

FIG. I-1 Program development process.

All program development activities should be geared toward the goal of maximum user impact. Activities that have little or no bearing on this impact (such as the development of a superefficient code that reduces the number of statements or execution time by 10%) should be omitted or abandoned. Activities that do have great user impact, such as documentation and output design, should be given in-depth attention.

In order to achieve maximum user impact, the program should possess the following attributes and features:

(1) adequate documentation,
(2) user oriented input requirements,
(3) outputs designed to fit user needs,
(4) validity checking of user inputs,
(5) meaningful error messages if necessary,
(6) reasonable turn around time.

These features must be incorporated as integral program elements from the start of the program design process and cannot be added as an afterthought.

The following guidelines have proved effective in developing reasonably efficient, error-free code in a relatively short time:

(1) Spend considerable time planning the programming effort.
(2) Use a standardized program organization.
(3) Write simple code.
(4) Write self-documented code.
(5) Be general whenever possible.
(6) Do not punch a single card until the program design is completed.

To breach this last guideline can at times be most tempting. However, to do so will almost certainly result in failure and loss of precious time. Detailed discussion of all six guidelines will occur in the following sections.

C. PROGRAM ORGANIZATION

Simplicity is the key to good program organization. The goal is to achieve a modularized program wherein two things occur: (1) the different logical functions are separated in different sections of the program, and (2) these separate sections are integrated in as simple a manner as possible.

A good test for adequacy of organization is to show the program to a competent programmer. If after some study he is able to determine what the program is supposed to do, then the program structure is probably sufficient.

One way to achieve a well-structured modular program design is to use a hierarchical approach in drawing the program flowchart. An initial one page

flowchart is drawn showing the major steps in the program. On successive flowcharts, the functions in individual flowchart "boxes" are "blown up" in more and more detail. Whenever possible, do not extend a flowchart to a continuation page. Keeping the flowchart on one page increases its readability and allows the programmer to focus on program details relevant to a given module without being distracted by other programming concerns. In Fig. I-2 we show a hierarchical flowchart of a program designed to multiply two matrices together. This flowchart also incorporates the following two key flowcharting conventions:

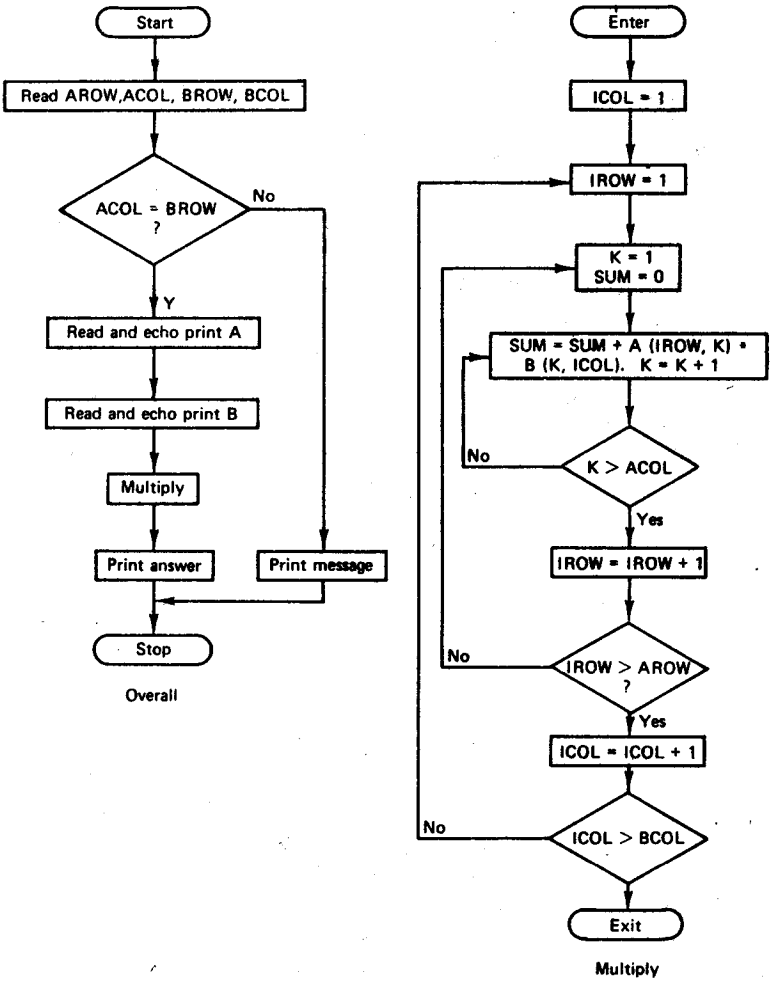

FIG. I-2 Hierarchical flowchart.

(1) Always draw flowcharts such that the longest chain of logic appears as a straight line down the middle of the page.
(2) Do not cram too much information into one flowchart.

The layout of the computer code can be determined as soon as the flowchart has been completed and all issues regarding method and program design have been resolved. We advocate the use of a consistent layout for all programs. The following sequence of major program blocks may serve as an illustration of how a large number of computer codes can be structured:

(1) comment cards describing program purpose, outputs, input card preparation,
(2) additional comment cards describing the method and key variables (if necessary),
(3) declarative statements in the following order:

IMPLICIT
REAL
INTEGER
DIMENSION
COMMON
LABELED COMMON
DATA

(4) program initialization section,
(5) data input section,
(6) method,
(7) output.

While you may choose a different layout than the one suggested here, it is important that you consistently use the same layout every time you write a program.

D. PROGRAMMING DETAILS

A standardized approach to programming results in a semiautomatic execution of trivial tasks (such as program layout, choice of variable names, assignment of statement labels). Thus, it reduces the likelihood of errors in these tasks and frees time and attention for more important tasks. A standardized approach also pays off in maintenance and debugging since the resulting standardized format renders the programs very easy to read and understand.

The following rules will assist in achieving a consistent and self-documenting style in the production of programs:

TABLE I-1
KEY FEATURES OF USA FORTRAN IV

Identifier	Abbrev.	Definition	Example
Constant	*a*	Real number	2.735, −0.153
	i	Integer	73, 8321, −43
	—	Logical variable	.TRUE., .FALSE
	d	Double precision	2.23576382173
Expression	*e*	Composite of operators and variables and/or constants of a single mode	(A * B−Q)/2.738
Label	*l*	Unsigned integer constant	1000, 100, 2730
List	—	List of variables separated by commas	A, B, C
Name	—	Identifier of functions, subroutines, and common areas	WIDES, A100
Variable	*a, i, v*	Identifier of storage areas containing constants	A, B, C, I, KONT

Executable statement	General form	Example
Assignment	variable = expression	A = 2.2 * AVG−STDV
BLOCK DATA	BLOCK DATA	BLOCK DATA
	COMMON/name/list	COMMON/LOC/A, B, C
	DATA list/constants/	DATA A, BC/2.2, 2 * 7.01
	END	END
CALL	CALL name (list)	CALL BILL(CREDIT)