Optimization Techniques in FORTRAN

JOEL L. SEARS

Joel L. Sears

OPTIMIZATION TECHNIQUES IN **FORTRAN**

a petrocelli
book
new york / princeton

Printed in the United States
1 2 3 4 5 6 7 8 9 10

Library of Congress Cataloging in Publication Data

Sears, Joel L
Optimization techniques in FORTRAN.

Includes index.
1. FORTRAN (Computer program language) 2. Mathematical optimization. I. Title.
QA76.73.F25S38 001.6'424 79-9710
ISBN 0-89433-034-9

INTRODUCTION

Optimization has emerged as a science unto itself. In the past, computer scientists and practitioners were primarily concerned with the optimization of the computer resource itself. This meant that major efforts were aimed at reducing CPU time, minimizing storage requirements, and enhancing the overall "efficiency" of systems through a combination of hardware and software refinements.

The obvious reason for this orientation was *cost*. With computer charges in excess of $1,000 per hour, it made a great deal of sense to commit significant amounts to research and development in the field. After all, seven to ten dollars an hour for a highly trained specialist's time was a bargain when contrasted with the costs and potential savings twenty years ago.

Without saying that all this attention to the computer is no longer necessary, we must recognize that optimization has taken on a far broader meaning—particularly in the *economic* sense of the word. It is one thing to speed up a system and another to reduce its cost.

Recognizing that the tariffs for performing routine tasks with a computer have been reduced a thousandfold in some cases doesn't imply that hourly rates have been reduced from hundreds and thousands of dollars per hour to pennies. But it does indicate that, with the incredible speed and capacity of modern computers, each calculation, com-

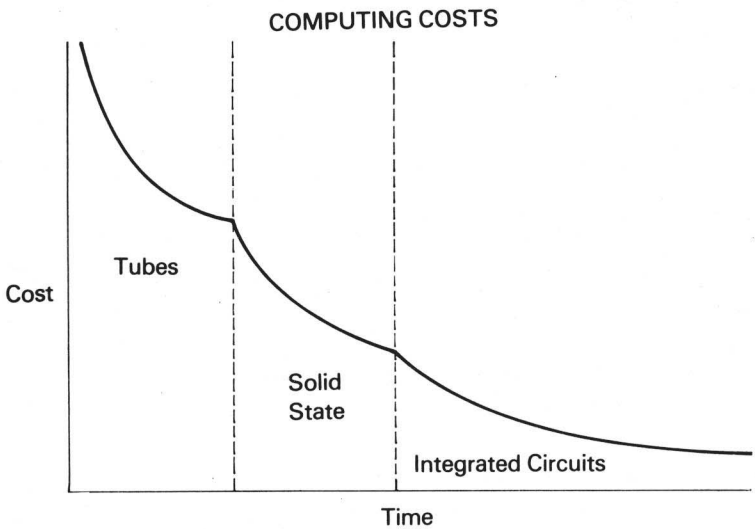


Figure A: History of computing costs

parison, or transfer can be performed in a mere fraction of the time required by machines common in the early 1960s.

This technological improvement, coupled with manufacturing advances as well as increased marketing pressures applied by hundreds of mainframe and peripheral vendors, has literally knocked the bottom out of price consciousness as we once knew it. The curve in Figure A relates computational costs and time since the dawn of the computer age.

At the same time, enemy inflation *has* been working noticeably at the human end of the cost spectrum. So much so that during the past 25 years, personnel budgets have nearly quadrupled as a result of higher wages, growing staffs, and the expense of underwriting the enlightenment of the community as a whole. Therefore, as raw computing costs have decreased significantly, so have payrolls, R&D budgets, and education costs increased as portrayed in Figure B.

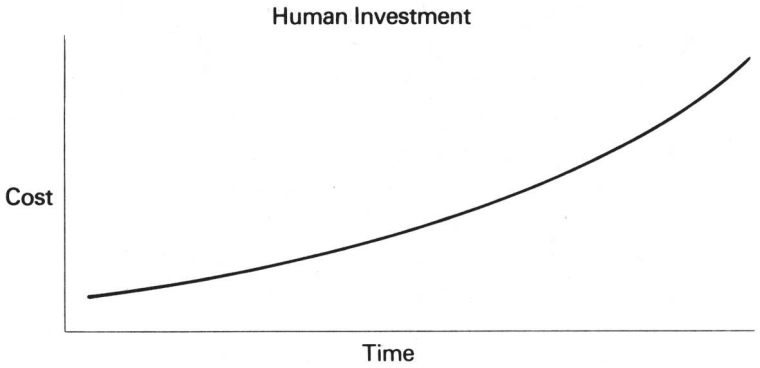
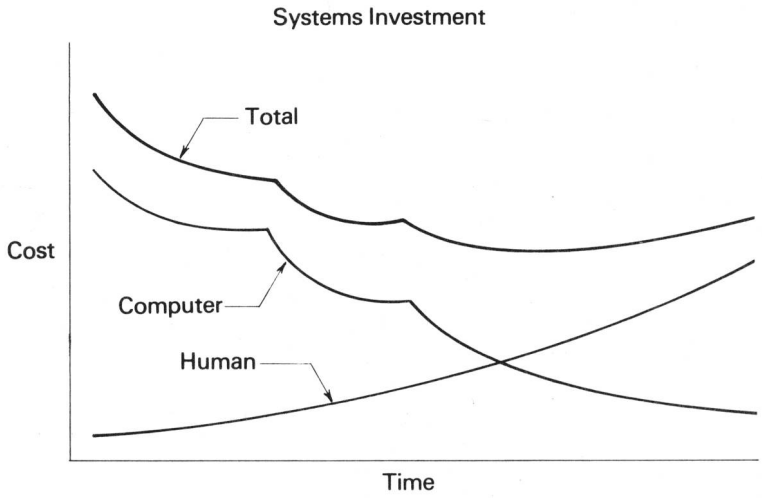


Figure B: Rising investment in people

This then introduces the modern orientation which is necessarily more balanced than its predecessor. Optimization as practiced in a business environment takes a turn less transfixed on computer costs toward the *overall* investment

Figure C: Total investment in systems



in systems. By adding the two previous curves, we can appreciate the rationale for a balanced approach (Figure C).

The balanced philosophy provides the foundation for the optimization techniques presented in this book. As a user rather than student of computers, my objective has been to control and minimize the total cost of a particular project through a combination of computer and human efficiencies. The methods advocated herein are intended to help through their direct application and to stimulate the development of extensions and original methods by each reader. You may even find areas which contradict classical efficiency techniques. Bear in mind, however, that our problem is now somewhat broader and calls for a more diversified solution.

Aside from optimization techniques *per se* is the attention paid to FORTRAN in a business context. Admittedly, there is a natural tendency to stress COBOL or PL1 in commercial applications. However, at least three factors underpin the rationale for the ever-increasing use of FORTRAN in business (or nonscientific) applications.

First, many of us have relied on FORTRAN as a "quick and dirty" means of solving problems. We could easily automate a process by brute force with little regard for the sort of developmental overhead associated with COBOL, for example (i.e., lengthy environment and data divisions).

Second, FORTRAN is commonly assimilated by students whose majors are formally separated from the computer sciences. Budding accountants, statisticians, biologists, physicists, and psychologists alike opt for FORTRAN to solve the quantitative problems associated with their specialties. The movement of these *users* into the business world in steadily increasing numbers brings all the tools and knowledge acquired in training to the problem-solving concerns of business in general.

Third, FORTRAN is one of the few "traditional" languages available in both interactive and batch environments. Such

commonality encourages program development and testing in the interactive mode for direct adaptation to production in the batch world.

The response of computer manufacturers to widespread commercial use is encouraging, to say the least. Powerful extensions which support nonnumeric processes, enhanced input/output efficiencies, and integration with data base management systems are now available to nearly every major computer system user. The strongest evidence of FORTRAN as a general language can be found in the 1978 ANSI Standard, an ambitious recognition of the extensions which have evolved during the 11 years since the last standard was published.

As a practitioner, I have drawn from ten years of experience backed by formal training in the computer sciences. Recognition belongs to the following for their inspiration and indirect contributions: IBM, Honeywell, CDC, Burroughs, Sperry-Univac, DEC, and Peterson, Howell & Heather, Inc.

Special thanks go to Dee Madden for her secretarial assistance and to Al Fentress and Mac McDaniel for encouraging me to share these techniques with you.

CONTENTS

1

CHARACTER VALUES

The variables and constants generally associated with FORTRAN programs are numeric. After all, the language evolved originally to handle scientific problems. The treatment of character data (popularly called "Hollerith" or "literal" data) was limited at best. Thus, we wrote many programs which converted alphabetic values to some numerical equivalent to allow for comparisons, sorting, etc.

In short order, practitioners and software experts alike recognized the need for additional character manipulation capabilities to satisfy the requirements of "commercial" applications. Naturally, a whole host of character and even bit manipulation features have appeared recently.

But, examining what really occurs during the compilation and execution of this character-oriented logic, we see that powerful capabilities exist even in versions of the language without modern character extensions.

First, since the *loading* of values to memory does not necessarily depend upon the declared variable *type*, it is permissible to "mix" values and variables in what appear to be incompatible ways. For example, this is a legal way to load a character value:

```
INTEGER LABEL/'ABCD'/
```

Since the INTEGER declaration defines the mathematical context of LABEL, it is now reasonable to compare or even sort variables and arrays so loaded.

To load these variables during execution (by reading their values from a tape file, for instance), merely use the appropriate A format and treat the variable as you please.

In addition, mixed alphanumeric values such as 'A123' can be handled in similar fashion. Traditionally, such values posed many interesting problems for a FORTRAN programmer. Because short alphanumeric values appear frequently in business applications as inventory codes, customer numbers, and others, they were usually decomposed, converted for analysis, and, finally, recomposed for output. Such values could have been treated directly by the method discussed earlier. In other words, 'A123' can be handled intact. There is no need to create two values for analysis purposes.

Character values and variables also appear in the context of "variable formats" (discussed in greater detail in another section). Such variables actually contain format information which can be altered during execution like any other variable. The format itself can be read from an input file or created and modified strictly in the coding. Here are two brief examples:

Example 1

```
*   DECLARE STORAGE FOR VARIABLE FORMAT, FMT1
      REAL FMT 1 (10)
        .
        .
        .
*   LOAD FORMAT INFORMATION INTO FMT1
      READ (1,10) FMT 1
101   FORMAT (10A6)
        .
        .
        .
      READ (2, FMT1, END = 200) RECORD
```

Example 2

```
*   SET UP VARIABLE FORMAT WITH DUMMY TAB AND REPEAT INFO:
      CHARACTER VFMT * 13 /'(A4, TYY, XXI6)'/
```

```
*  SET UP TAB AND REPEAT INFO. AS FUNCTION OF K.
      ENCODE (VFMT, 6001) 90-K*6, K
6001  FORMAT (T6, I2, T9, I2)
        .
        .
        .
      WRITE (2, VFMT) LABEL, (OUTDATA (I), I = 1, K)
```

In the first example, a generalized program has been created to allow the user to specify the actual record layout in a flexible fashion at program execution time. In the second, representing a well-extended FORTRAN compiler, the format VFMT is altered to insure that the information in OUTDATA is always *right* justified in the output record.

2

COMMENTS

Nothing obscures the meaning of a program more than the passage of time, and nothing is more exasperating than reinterpreting old, nondocumented code. To put the teeth into true optimization, then, each of us is bound to include comments in our programs while its concepts are fresh. Follow this simple rule and you can painlessly create well-documented code which will be as clear in the future as it is on the drawing board: *Say it in English* first, *then code it.*

To minimize the effort required to "say it," practice creating highly descriptive variable names, use indentation, blanks and blank lines, and try creating informal comments with executable logic. The following example serves to illustrate the difference in approach and readability. First, try to figure out the meaning of this "traditional" section of code:

```
      .
      .
      .
      DO 10 I = 1,N
10    P = P + X(I)
      P = P/N
      R = .0125
      IF(CC.EQ.10) R = .008
      .
      .
      .
```

Now try to figure out this commented version:

```
                    .
                    .
                    .
*   COMPUTE CLIENT'S AVERAGE DAILY BALANCE
**  FIRST ACCUMULATE TOTAL PURCHASES FOR THE MONTH
                    DO 10 I = 1, DAYS
                10 TOT PUR = TOT PUR + PURCH (I)
**  DIVIDE THIS TOTAL BY THE NUMBER OF DAYS IN THE MONTH TO
**  OBTAIN THE AVERAGE DAILY BALANCE
                    AVG BAL = TOT PUR/DAYS
*   BECAUSE WE GIVE LOWER INTEREST RATES TO PREFERRED
*   CLIENTS, THE CODE MUST BE CHECKED BEFORE FIXING THE
*   RATE
**  FIRST, ASSUME CLIENT IS NOT PREFERRED, THEN . . .
                    RATE = .0125
**  HOWEVER, . . .
                1F (CODE.EQ.10) RATE = .008
                    .
                    .
                    .
```

While the traditional approach is obviously easier to code, it certainly lacks the clarity and explicit rationale demanded by the modern accountants, auditors, and managers whom we serve. Furthermore, the commented version may even obviate the need for a separate document, since it fully explains the assumptions and procedures employed by the analyst. This continually encourages documentation which is both accurate and up-to-date.

3

COMMON

Now to explore one of the most powerful features of FORTRAN. Used traditionally and in combination with other FORTRAN statements, COMMON is the key that unlocks a treasure chest of programming capabilities.

By definition, COMMON establishes those locations in memory which are to be shared by many program routines. Additionally, COMMON invokes contiguous organization of the variables and arrays associated with it.

Typically, there are two types of COMMON—block and labelled. To confuse matters, "block" is often called "blank" or "unlabelled" since it is not identified by a label. The term "block" was coined to associate COMMON with the BLOCK DATA subprogram feature of FORTRAN which provides compile-time initialization of the variables in "block" COMMON.

In general, COMMON variables which are required globally—that is, in every routine of a program—are stored in block COMMON. Those which are used in a smaller subset of the total are best associated with a labelled COMMON area. The relationship of block and labelled COMMON is depicted for a typical program in Figure 3.1. For a program mapped like this one, the COMMON statements might look like this:

```
*MAIN
      COMMON SALES, EXPENS, MARKTG
      COMMON /L1/ MARGIN
```

```
      .
      .
      .
      END
      SUBROUTINE SUB1
      COMMON  SALES, EXPENS, MARKTG
      COMMON  /L1/ MARGIN
      COMMON  /L2/ TAXES
        .
        .
        .
      END
      SUBROUTINE SUB2
      COMMON  SALES, EXPENS, MARKTG
      COMMON  /L2/ TAXES
        .
        .
        .
      END
```

Figure 3.1: Relationships of block and labelled COMMON *to the routines in a program*