C20-1642-0

IBM Data Processing Techniques

1130 FORTRAN Programming Techniques

This manual contains an elementary analysis of 1130 FORTRAN object programs and discusses some specific FORTRAN programming techniques for writing more efficient programs. The information contained in this manual is primarily intended for the beginning programmer as an aid to achieving improved 1130 system performance.

Programming

CONTENTS

Copies of this and other IBM publications can be obtained through IBM branch offices. Address comments concerning the contents of this publication to IBM, Technical Publications Department, 112 East Post Road, White Plains, N.Y. 10601

INTRODUCTION

Today's programmer has changed radically from the programmer of 1954-1959. At that time, most programs were coded as concisely as possible. It was considered a good programming habit to code, recode, and even recode again to achieve the smallest program or the fastest program or some compromise between the two.

With the advent of FORTRAN, this ceased to be the case. Programming became so much easier that programmers were frequently running production problems after one or two compilations. This situation was certainly superior to that of the past, because machines were utilized more, programming time was decreased, and the production of usable results became the paramount issue.

This ease of programming had another side, however. It was so trivial to produce a small program that the principles of good coding habits began to be lost. When larger programs, or aggregates of smaller subprograms, would not reside in core because of their size, the solution used more often was more equipment rather than program revisions.

Today, it is uncommon to review the logic of a FORTRAN program after it is completed. The program is written once and only once. After that, it is patched, repaired, added to, and frequently inflated to the point where it escalates out of available storage.

Certainly, few of us wish to return to the day when two hours of hard work in problem redesign would yield a saving of two machine words out of a total of 100. Yet the problem analysis should continue to exist after programming is completed. If not, an installation will be plagued more and more by programs that fail to fit in an object machine.

In the case of the 1130, the beginning programmer should learn a variety of techniques in order to best utilize available core. These techniques, once well learned and practiced, will pay considerable dividends in machine usage.

AN ELEMENTARY ANALYSIS OF FORTRAN OBJECT PROGRAMS

What does a FORTRAN compiler do to a FORTRAN statement?
Why does a FORTRAN program overflow the available core storage?
Can anything be done to avoid this as much as is possible?

In the broadest sense, all of these questions, and more, must concern a FORTRAN programmer at one time or another. He must be aware that an innocent statement may completely exhaust his available storage, while an impressive and lengthy EQUIVALENCE statement need not overly concern him.

Many beginning users find FORTRAN object programs puzzling, and almost fail to understand how a compiler can do what it does. Only after months of experience do they begin to see the relationship between the source program and the object program (via the compiler). It has a sobering effect upon a programmer to learn that a DO loop does what it is designed to do in a very simple fashion. The computed GO TO is another mysterious source language statement which, when its veils are removed, becomes quite obvious.

Probably the greatest portion of the average FORTRAN source program is devoted to arithmetic statements. But most of an object program does not necessarily contain the information generated by the source language arithmetic statements. Consider source language arithmetic statements.

As the compiler translates each arithmetic statement, it produces other statements as a result of its analysis of the source language statement. The technique for this analysis is not under discussion here. Suffice it to say that such a capability exists. The instructions generated by the compiler (which become the object program, and which, upon execution, perform the work designated by the source language statement) are obvious in their intent when one views them. As an example the statement

```
X = A + B
```

generates object code which, in symbolic form, looks like this:

```
LOAD  A
ADD   B
STORE X
```

This means that (1) the number located in the cell whose symbolic name is A is to be placed in an accumulator (by the LOAD A instruction), (2) the number in the cell whose symbolic name is B is to be added to the current contents of the accumulator (by the ADD B instruction), and (3) the result is to be stored in the cell whose symbolic name is X (by the STORE X instruction).

A second example of generated object code is as follows:

$$X = A + B * C/D$$

The symbolic object code might appear as:

Instruction	Operand
LOAD	B
MULTIPLY	C
DIVIDE	D
ADD	A
STORE	X

If a computer is designed without the functions of loading, multiplying, dividing, etc., it is necessary to produce slightly different object code. The first example, X = A + B, might appear as:

Instruction	Operand 1	Operand 2
LINK	LOAD	A
LINK	ADD	B
LINK	STORE	X

This means that the object program will simulate the conditions of loading, adding, and storing through subroutines contained in the object program and that it will arrive at these subroutines through a linking technique peculiar to the implementation of that particular compiler. Thus the first of the generated instructions states that a link to a routine which simulates loading is desired with the argument specified by the variable whose symbolic name is A. The second instruction states that a link to a routine which simulates adding is desired with the argument specified by the variable whose symbolic name is B. The third instruction states that a link to a routine which simulates storing is desired with a variable whose symbolic name is X.

This latter type of coding is termed out-of-line object code, because the actual work is performed outside the flow of the instructions generated by the source program (via the compiler). The former type is in-line, because the work desired by the source program is done directly by the instructions generated by the source program (via the compiler).

The compiler author's choice of in-line or out-of-line coding is based primarily on the repertoire of instructions in the machine under consideration. However, the size of the subroutines required to perform the work and the construction of the compiler itself play an important role in this decision. In either event, object code, whether in-line or out-of-line, does perform the arithmetic required by the source program, either directly (in in-line coding) or indirectly (in out-of-line coding).

The object code produced by the compiler may be symbolic in nature, in which case a subsequent assembly is required, or it may be the actual object code itself, in which case no subsequent assembly is required.

In the 1130 most of the object code is out-of-line, because of the need for subroutines to do the actual arithmetic.

In the light of the previous comments on compilation, consider the statement:

X = J

If a FORTRAN program containing nothing but the above statement were written, the core used could be approximated by the following:

A. Three Subroutines Required

1. Load the Accumulator (35 words)
2. Store the Accumulator (35 words)
3. Convert from Integer to Real (50 words)

B. Two Data Cells Required

1. X as a REAL variable (2 words — or 3 in extended precision)
2. J as an INTEGER variable (1 word)

C. Three Generated Instruction Links Required

1.	LINK	LOAD (J)	(2 words)
2.	LINK	FLOAT	(1 word)
3.	LINK	STORE (X)	(2 words)

D. Three Transfer Vectors Required (3 words each)

This is a compiler peculiarity.

Thus, the innocent statement

X = J

uses 11 words (12 with extended precision), plus the core required for three subroutines (120 words), for a total of 131 (or 132) words. This is 3.1% of the total core on a 4096-word 1130.

A FORTRAN program containing the statements

X = J

Y = K

would not generate a requirement for two separate sets of subroutines, to LOAD, FLOAT and STORE, because one set of three may be reused as many times as required. However, the data cell requirements will be increased by the REAL variable Y and the INTEGER variable K. The generated instructions requirement will be increased by three additional links, or five words. The transfer vector requirement does not change.

Thus, to go from a program with no statements to a program consisting of the statement

X = J

calls for 131 (132) words, but a program of

```
X = J
Y = K
```

requires only 8 (9) words of additional storage and the core used goes only from 3.1% to 3.2%.

A third FORTRAN program of

```
X = J
Y = K
Z = K
```

would require 7 (8) additional words, because a data cell is not set aside for the second occurrence of the INTEGER variable K.

Another program of

```
X = J
Y = K
Z = A
```

would, in addition to the Z space, require a data cell of 2 (3) words for the REAL variable A.

Thus, the increase from

```
X = J
Y = K
```

to

```
X = J
Y = K
Z = K
```

is 7 (8) words, while the increase required by

```
X = J
Y = K
Z = A
```

is 8 (10) words.

This is because 2 (3) calls are set aside for both Z and A in the data cell area giving 4 (6) words. Also, 2 links are generated, at 2 words each, instead of 3, because no floating of the REAL variable A is necessary. It is already in that form internally.

Reviewing total core requirements, we have

```
X = J
```

requiring 131 (132) words;

```
X = J
Y = K
```

requiring 139 (140) words;

```
X = J
Y = K
Z = K
```

requiring 146 (147) words; and

```
X = J
Y = K
Z = A
```

requiring 147 (149) words of storage.

This discussion shows that:

1. In general, FORTRAN statements generate, at compile time, other statements. This second set of statements resides in core when running an object program.

2. Data storage areas for every variable or constant are also required at object time.

3. Certain work-performing subroutines may be required, and these will also reside in core at object time.

4. Miscellaneous object core requirements may result from peculiarities of the compiler design.

5. Together, all these core requirements determine the size of an object program. If a FORTRAN program is sufficiently large, storage will be exhausted by these requirements and the program will not fit in the available core.

6. Once a FORTRAN statement is made, it is relatively inexpensive to make more of the same kind. Thus, a large program of like statement may be possible in a modest core, while a smaller program of dissimilar statements may not be possible. We may even predict, with considerable accuracy, how many statements of the form $VARIABLE_i = VARIABLE_j$ can be fit in core. This could be done by making certain acceptable assumptions:

a. All statements are of the form REAL VARIABLE = INTEGER VARIABLE and INTEGER $\text{VARIABLE}_i \neq \text{INTEGER VARIABLE}_j$ We have seen that all statements of this form require three generated instruction links at 5 words and two data cells at 3 (4) words. Thus, each statement takes an 8 (9)-word byte from the 4096 words minus the 123 words constant requirement for subroutines and transfer vectors. Thus, we have room for 496 (441) such instructions. The actual amount is slightly less than this because of some minor items not discussed here. This number would increase if the restriction $\text{INTEGER VARIABLE}_i \neq \text{INTEGER VARIABLE}_j$ were removed.

b. All statements are of the form REAL VARIABLE_i = REAL VARIABLE_j. By the same logic we see that the generated instruction links account for 4 words and the data cells for 4 (6) words. Thus, each statement takes an 8 (10)-word byte from the remaining 4096 minus 123 words. This yields 496 (397) such instructions.

It would be foolish to assume that a reasonable program could be composed of nothing but 496 instructions of the form:

A = I

B = J

etc.

However, the points to be digested should be the clarity of the program analysis and the implication of a collection of like statements — not the utility of this particular case.

OTHER FORTRAN STATEMENTS

Few FORTRAN programs are made up of just one type of statement; control statements such as DO, GO TO, IF, etc., are required. There are input/output statements such as READ, WRITE, etc. There are data arrangement specifications such as COMMON, DIMENSION, EQUIVALENCE, etc., and also data type specifications.

Some of these statements eat core in small bytes while others take larger gulps. Almost all take something. Consequently, the aggregate of the FORTRAN source statements produces an object program that either will or will not fit core. The purpose of the following discussion is to present some techniques that will cause the former condition to occur most frequently. These techniques are:

1. Program rewriting
2. Program segmentation
3. Subroutine subprogramming
4. Using all of core
5. Chaining
6. LOCAL
7. Use of DIMENSION
8. Use of EQUIVALENCE
9. Data Transfer
10. Superscription

Program Rewriting

It is almost a truism that every program can be shortened. The following examples demonstrate a particular case, but the techniques are applicable to many programs. The discussion is based on Simpson's rule for numerical integration. This very simple problem was chosen to illustrate the point of shortening source programs. It is unlikely that such a trivial case would exist in actuality.

$$\int_{x_0}^{x_n} x\,dx = \frac{h}{3}(x_0 + 4x_1 + 2x_2 + 4x_3 + 2x_4 + \ldots + 2x_{n-2} + 4x_{n-1} + x_n)$$

where $x_{i+1} = x_i + h$

Example: $\int_1^5 x\,dx$

Choosing $h = .5$

$$\int_1^5 x\,dx = \frac{.5}{3}(1 + 4(1.5) + 2(2) + 4(2.5) + 2(3) + 4(3.5) + 2(4) + 4(4.5) + 5) = 12$$

The following seven examples progressively shorten the program. Upon investigation program 5 is larger, from the standpoint of core requirements, than several of the less elegant cases even though it does not seem to be so. Program 6 is a different approach entirely and is the shortest and best of all those presented. Program 7 is an example of going too far with this technique. Though the source program is the shortest (in number of statements), the number of object instructions executed is very high for large $(x_n - x_0)$ and/or small h.

PROGRAM 1

```
      SUBROUTINE INTGR(XZERO, XSUBN, H, SUM)
      A=XZERO
      B=XSUBN
      HOVR3=H/3.
      I=1
      SUM=A+B
1     A=A+H
      GO TO (2,4),I
2     SUM=SUM+4.*A
      IF(A-(B-H))3,6,6
3     I=2
      GO TO 1
4     SUM=SUM+2.*A
      IF(A-(B-H))5,6,6
5     I=1
      GO TO 1
6     SUM=SUM*HOVR3
      RETURN
      END
```

Note: There is no requirement to introduce the variables A, B, and HOVR3. They serve only to lengthen the program. The testing (IF (A-(B-H))) is clumsy and badly placed in the text.

PROGRAM 2

```
      SUBROUTINE INTGR(XZERO, XSUBN, H, SUM)
      I=1
      SUM=XZERO+XSUBN
      XZERO=XZERO+H
1     GO TO (2,4), I
2     SUM=SUM+4.*XZERO
      I=2
3     XZERO=XZERO+H
      IF(XZERO-XSUBN)1,5,5
4     SUM=SUM+2.*XZERO
      I=1
      GO TO 3
5     SUM=SUM*H/3.
      RETURN
      END
```

Note: The extraneous variables have been removed and the test for completion less clumsy, but the presence of a pair of two identical statements (XZERO = XZERO + H, I = 1) implies that a further rearrangement is possible. One should always be wary of a program that repeats identical statements.

PROGRAM 3

```
      SUBROUTINE INTGR(XZERO, XSUBN, H, SUM)
      SUM=XZERO+XSUBN
1     I=1
2     XZERO=XZERO+H
      IF(XZERO-XSUBN)3, 6, 6
3     GO TO (4, 5), I
4     SUM=SUM+4.*XZERO
      I=2
      GO TO 2
5     SUM=SUM+2.*XZERO
      GO TO 1
6     SUM=SUM*H/3.
      RETURN
      END
```

Note: By rearrangement of statements, we have eliminated one appearance of XZERO = XZERO + H, and I=1, However, the similarity between SUM = SUM + 4. *XZERO and SUM = SUM + 2. *XZERO is now suspect.

PROGRAM 4

```
      SUBROUTINE INTGR(XZERO, XSUBN, H, SUM)
      SUM=XZERO+XSUBN
1     A=4.
      I=1
2     XZERO=XZERO+H
      IF(XZERO-XSUBN)3, 5, 5
3     SUM=SUM+A*XZERO
      GO TO (4, 1), I
4     I=2
      A=2.
      GO TO 2
5     SUM=SUM*H/3.
      RETURN
      END
```

Note: The two similar (but not identical) statements of program 3 have been modified so that now a single statement SUM = SUM + A *XZERO appears with A alternately 2 and 4. Now, however, note the similarity between SUM = SUM + A *XZERO and SUM = XZERO + XSUBN in that a special case is required for the first and last points of the series.

PROGRAM 5

```
      SUBROUTINE INTGR(XZERO, XSUBN, H, SUM)
      SUM=0.
1     A=1.
2     I=1
3     SUM=SUM+A*XZERO
      XZERO=XZERO+H
      IF(XZERO-XSUBN)4, 1, 7
4     GO TO (5, 6), I
5     I=2
      A=4.
      GO TO 3
6     A=2.
      GO TO 2
7     SUM=SUM*H/3.
      RETURN
      END
```

Note: Although we have succeeded in reducing the principal arithmetic to one statement (SUM = SUM + A *XZERO), the extraneous control arithmetic necessary for proper functioning makes this case less desirable than program 4.

PROGRAM 6

```
      SUBROUTINE INTGR(XZERO, XSUBN, H, SUM)
      A=1.
      SUM=XZERO+XSUBN
1     XZERO=XZERO+H
      IF(XZERO-XSUBN) 2, 3, 3
2     SUM=SUM+(3. +A)* XZERO
      A=-A
      GO TO 1
3     SUM=SUM*H/3.
      RETURN
      END
```

Note: A total change of direction here has produced the best program so far.

PROGRAM 7

```
      SUBROUTINE INTGR (XZERO, XSUBN, H, SUM)
      A = 1.
      SUM = H/3. * (XZERO + XSUBN)
1     XZERO = XZERO + H
      SUM = SUM + (H/3.) * (3. + A) * XZERO
      A = -A
      IF(XZERO + H -XSUBN) 1, 2, 2
2     RETURN
      END
```

Note: From the standpoint of length, this program is less satisfactory than number 6. From the standpoint of the number of object instructions executed, it is extremely poor. Note that (H/3) is a multiplicative constant for every execution of the statement SUM = SUM + (H/3) * (3 + A) * XZERO

	1	2	3	4	5	6	7
Real variables	7	4	4	5	5	5	5
Real constants	3	3	3	3	5	2	2
Integer variables	1	1	1	1	1	0	0
Integer constants	2	2	2	2	2	0	0
Statement numbers	6	5	6	5	7	3	2
Number of floating-point operations	19	16	14	13	13	15	17
Number of fixed-point operations	3	3	2	2	2	0	0
IF's	2	1	1	1	1	1	1
GO TO's	2	1	2	1	2	1	0
Computed GO TO's	1	1	1	1	1	0	0

Comparative analysis of programs 1 - 7

Program Segmentation

Every main-line program should be written as if it were to be segmented eventually. The worst case, that of memory overflow, should be planned for in advance so that if it occurs, it will be much more easily handled.

Breakpoints in the logic of a program — that is, points where a program phase may be terminated and a subsequent phase entered — should be frequent.

Through the judicious use of the COMMON facility, data that is common from program to program may be kept. However, not all data need be kept in COMMON storage.

Programs that read data, compute and punch/print results lend themselves to segmentation most readily, but they are not the only type that may be segmented. Programs having interspersed reading, computing, and punching may be segmented into reading and computing phases and computing and punching phases.

The best segmentation is to have like phases in memory as much as possible. Thus, it is more profitable to have a read phase followed by a compute phase than to have a read-compute phase. This philosophy may be even extended further: it is more profitable to have a floating-point phase followed by a fixed-point phase than to have a floating-point/fixed-point phase.

One may even say that a floating add-subtract-store phase followed by a floating multiply-divide-store phase is more economical to core than a phase requiring add, subtract, multiply, divide, and stores. However, this may not be economical to the user. There is, of course, a point where phasing could be detrimental to the user and this point must be avoided. Still, to ignore the possibility of segmenting is to ignore a powerful tool.

Subroutine Subprogramming

Whenever possible, entire programs should be written as sequences of calls to subroutine subprograms, because each CALL is a logical breakpoint and one can load the entire machine with as many as will fit. Then the next phase repeats the process of loading core with the next group of subroutines, etc.

After a little practice, one can get a feeling of how much will fit. Three phases of a program might look like this:

Phase 1

```
COMMON A, B, C, D, E, F, G,  ANS 3
CALL IOIN (A, B, C, D, E, F, G)
CALL SUB 1 (A, D, F, ANS 1)
CALL SUB 2 (B, C, ANS 1,  ANS 2)
CALL SUB 3 (E, G, ANS 2,  ANS 3)
STOP
END
```

Phase 2

```
COMMON A, B, C, D, E, F, G,  ANS 3,  ANS 4
CALL SUB 4 (F, G,  ANS 3,  ANS 4)
STOP
END
```

Phase 3

```
COMMON A, B, C, D, E, F, G,  ANS 3,  ANS 4
CALL IOOUT (ANS 4)
STOP
END
```

This solution to overflow problems has both advantages and disadvantages.

ADVANTAGES

- If any subroutine is expanded in function, only a phase rearrangement is needed to rerun the entire problem.
- There is no main-line code to be debugged.
- Since the size of each subroutine can be determined, little effort is required to determine how many subroutines will fill core to its capacity.
- These subroutines may be used repetitively for a variety of tasks.
- Short or medium-length subroutines are more easily written than long main-line programs.

DISADVANTAGES

- The recordkeeping to find a particular subprogram and its function is no small job.
- It is frequently necessary to carry unused data from phase to phase in order to keep the data in COMMON properly ordered.
- A good deal of shop standardization becomes necessary (an advantage or disadvantage, depending on point of view).
- Not all problems lend themselves to this linear approach.

In the long run, the flexibility provided by this technique, and the resultant machine use for problem program, far outweigh the inherent difficulties.

Using All of Core

In general, no more than 90% of available storage for any phase, program or subroutine should be used for the first six months. While this may seem to contradict previous remarks, it really does not.

At one time or another, every program expands because of such things as

1. Programmer errors.
2. The need to fulfill a function larger than originally anticipated.
3. Errors in the system program itself or the associated subroutines.
4. Upgrading of the size of data arrays.

If all of core is used for a program and no room is left for these eventualities, the coder finds he has no avenue of reasonable solution.

On the other hand, preparing for normal expansion makes the overall job much easier. It is the overall use of a computer leading to the production of problem answers that should concern the user — not each phase of the problem.

Phase Chaining (Disk Systems)

The remarks made under "Program Segmentation" apply to both disk and non-disk systems. However, the user with a file can segment his program so easily as to make the technique even more desirable.

Within reasonable limits, there is no restriction on the number of phases into which a program may be broken; thus there is no restriction on the size of a program. This technique may use disk storage in substantial amounts, but that is the purpose of any external storage device: to be used to its fullest extent.

The LOCAL Facility (Disk Systems)

If one is willing to accept certain restrictions (to be discussed), the LOCAL (LOad subroutine when CALled) facility is one of the most outstanding memory savers in the programmer's repertoire.

In essence, no LOCALed subroutine subprogram resides in memory unless it is specifically called through the encountering of an object time CALL, at which time it is loaded and executed. A later CALL to another subroutine causes the previous one to be overlaid, etc. Thus, the size of the largest subprogram determines the size of the overlay area.

Thus a program having many CALLs needs room only for those subroutines that are non-LOCALs, plus one area large enough for the largest of the LOCALs.

The restrictions mentioned before might be enumerated as:

1. The largest LOCAL must be small enough to reside in core.

2. Execution of object programs requires more control cards, since the monitor must know which subroutines are LOCALized. Also, it takes time to find, relocate, and store these subroutines in working cylinders.

3. Execution is less rapid. How much so depends upon the number of LOCALs and the number of times they are called.

Despite these drawbacks, the technique is immensely useful and, in conjunction with phase chaining, allows enormous FORTRAN programs with modest hardware requirements.

Use of DIMENSION

Generalized programs are frequently prepared to accommodate the maximum case. However, since this maximum case seldom, if ever, occurs, much space is wasted. For instance:

```
DIMENSION   A (30),   B (30, 30),   C (30, 30)
```

In this case, we assume that the programmer is planning on doing manipulation of data whose maximum sizes will occupy 1830 data cells at 2 (3) words per cell. Is this maximum case the norm? Does it ever occur? How frequently does it occur? When it does occur, can a special case, written just for the occurrence, be made to handle it? Might some of the array be equivalenced:

```
DIMENSION   A (30),   B (30, 30),   C (30, 30)
EQUIVALENCE   (B, C)
```

and thus save 900 data cells? Can the problem be rearranged to allow for just such a relationship?

The DIMENSION statement is frequently the most abused (and therefore the most damaging) statement in the FORTRAN programmer's repertoire.

Use of EQUIVALENCE

It is not at all common for a label's occurrence in a program to be unnecessary. Another data cell could have been employed but the mnemonic utilized by the latter was of limited meaning to the former. Thus, the programmer decides to invent a new label. This is acceptable if he also equates the variables so that they may both occupy the same memory location.

Thus many variables can be made to occupy less memory space than the maximum by an investigation as to which are not utilized beyond a certain point in the program. This, of course, is not the only use of EQUIVALENCE. It was conceived partially as a device to equate variable names whose spellings differ because two or more programmers have worked jointly. However, its application as a core saver should not be ignored.

EQUIVALENCE becomes very valuable when large arrays are assigned the same location or when a small array is made to reside entirely or partially in a larger one. The nature of the problem must allow this type of data storage.

Data Transfer in Subroutine Subprograms

If a main-line program calls a subroutine subprogram.

```
CALL SUB (ARG 1, ARG 2, ... ARGn)
```

the link to the subroutine is followed by the array of argument addresses. In addition, the subroutine must receive these addresses and join them in some fashion to the instructions that will use them. All of this takes space.

If neither an argument list in the main-line program nor a reception/distribution-of-the-addresses routine is required in the subroutine, considerable space may be saved. Such transmission may be done implicitly through the COMMON facility:

Mainline

```
COMMON ARG 1, ARG 2, ..., ARGn
CALL SUBA
CALL SUBB
CALL SUBC
END
```

Subroutine

```
SUBROUTINE SUBA
COMMON ARG 1, ARG 2, ..., ARGn
.
.
.
RETURN
END
```

Of course, arguments that are constant do not lend themselves to such easy manipulation but can be made to do so by equating them to arguments that are in COMMON. Thus considerable savings may be accomplished.

Superscription

Raising numbers to integral powers involves a different mathematical technique than raising numbers to fractional powers.

In the case of A^I, the implication (though not necessarily the implementation) is A^I=A.A.A. ... .A, I times. In the case of A^B, the implication is $A^B=e^{B\ln A}$.

Because of the requirements for exponential and logarithmic subroutines, A^B is frequently more expensive than A^I. However, it is possible, once it has been chosen, to use A^B for all exponentiation. On the other hand, it may be possible to use only the A^I routine.

If one has a program containing X=A**I, it might be more practical (depending on the other factors) to have B=I, X=A**B. The contrary might also be true.

On the other hand, if A < O, A^B is not defined, in which case the A^I form would be obligatory. Also, if a program has X=A**B and attempts replacement by I=B, X=A**I, B must be integral or A**I≠A**B. This technique works best in statements of the form X=A**2., where X=A**2 might have been employed.

It would be self-defeating if each programmer had to reflect for some time to choose the better form. He should not make too much of this issue, because it is not a large problem but rather just another applicable technique.

CONCLUSION

The use of just a few of the previously mentioned techniques can demonstrate memory savings in amounts ranging from small to substantial. However, these devices are a means to an end and not an end in themselves. Some programmers take this matter so seriously that coding tricks become the principal installation activity. This is not what a computer is for. It is, in the final analysis, a device for obtaining practical answers to practical problems. If, in free moments, one develops an extremely elegant way to calculate primes, determine perfect numbers, etc., he should by all means test it, but should not let this become the prime occupation of the computer (unless he happens to be in a numerical analysis laboratory).

As one learns more about his machine, compiler, assembly system, etc., he will find the few hints contained in this manual to be just an introduction to the thousandfold ways of shortening problems. If used wisely, they should represent the beginning of a substantial increase in computing potential.

Printed in U.S.A. C20-1642-0

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York 10601