$$\begin{array}{c} x \\ y \\ x^2 \\ xy \\ y^2 \\ 0 \\ 0 \\ 0 \end{array}$$

Thus, tapes 1 and 2 will be read 125 times to calculate 1000 records which will be written on tape 3.

2. Suppose that in paragraph 2 of Section 12.3 there are just eight results (one block) per case, and a variable number of cases with a tape mark at the end. The eight numbers are

$$\begin{array}{c} a_i \\ b_i \\ c_i \\ R_i \\ S_i \\ T_i \\ x_i \\ y_i \end{array}$$

where the subscript indicates the case numbers. Write the program to bring in this information from tape 1 and calculate

$$\sum_{i=1}^{n} (R_i + S_i + T_i), \qquad \sum_{i=1}^{n} x_i y_i, \qquad \frac{1}{n}\sum_{i=1}^{n} a_i, \qquad \frac{1}{n}\sum_{i=1}^{n} b_i, \qquad \frac{1}{n}\sum_{i=1}^{n} c_i$$

where n is the number of cases, and must be calculated.

3. Write a self-loading tape program which then loads tape blocks which are exactly the same as the card format of Section 11.5. This would be approximately the situation if auxiliary tape equipment were available.

4. Write the program of subroutine 1 of Section 12.2.

5. Write the program of subroutine 3 of Section 12.2.

6. Write the program of subroutine 4 of Section 12.2.

13 PROGRAM CHECKOUT

13.0 Introduction

Once a program has been written, it must be verified or *checked out* to determine if it actually does the job it is designed for. The steps of analysis and programming can lead to many *logical* errors, i.e., errors in conception or in flow through the problem. The testing of a loop may be set up improperly or the alternatives to be taken in certain situations may not have been thoroughly thought out. Possibly a mathematical procedure will not work in a certain case.

A large source of errors is the actual coding. This part of the task involves such a great mass of detail that simple mistakes can easily be made. An operation code may be copied incorrectly or remembered wrongly. Errors can be made which are simply blunders: the index control may be omitted, or a two written where there should be a three, or a tape address indicated which does not exist. There are so many possibilities of making mistakes that a perfect program is practically never written.

All these errors must be corrected before right answers can be obtained. There are several general approaches which may be taken, and several ways of making the machine help track down the errors. We shall discuss these in some detail, since checkout is a rather sizable part of the total cost of preparing a problem for computer solution.

13.1 Approaches to Checkout

The first attack on the checkout problem may be made before coding is begun. In order to fully ascertain the accuracy of the answers, it is necessary to have a hand-calculated check case with which to compare the answers which will later be calculated by the machine. This means that stored program machines are never used for a true one-shot problem. There must always be an element of iteration to make it pay. The hand calculation may be done at any point during programming. Frequently, however, computers are operated by computing experts who prepare the problems as a service for engineers or scientists. In these cases it is highly desirable that the "customer" prepare the check case, largely because logical errors and misunderstandings between the programmer and customer may be pointed up by such a procedure. If the customer is to prepare the test solution, it is best for him to start well in advance of actual checkout, since for any sizable problem it will take several days or weeks to hand-calculate the test.

In view of the time required, it is reasonable to ask why we bother with a check case. Is it not possible to check the computer's answers some easier way?

There are three answers to this. The first is the point mentioned above, which revolves around the serious communication problem in a service-bureau or "closed shop" type of computer operation. It is surprising how frequently misunderstandings can arise about details. These all have to be ironed out sooner or later, and a detailed check case is a good way to discover them fairly early.

The second reason revolves around a disastrous type of error which can be very difficult to catch, namely, the small error that results in a fairly reasonable answer. Suppose that a constant is entered incorrectly as 1.01 instead of 1.001. This is only about a 1% error, which from a standpoint of reasonableness of final answers may not be detected by anyone. Often the problem originator knows within 5% or 10% what the answers ought to be, from general knowledge of the physical situation—but not within 1%. Yet the data and numerical methods may be good to 0.1% and be used with this accuracy assumption. This sort of error is next to impossible to catch without a detailed, accurate test case. It is worth pointing out also that errors in input may result in much larger or much smaller errors in the final answers. If the number above appeared in a formula such as

$$F_n = \frac{14{,}576}{C_f - 1.001}$$

the error in the final answer would be quite large if C_f were close to 1.001, but undetectable (and probably harmless) if C_f were 10 or so.

This emphasizes that as far as possible the test case should pick the situations most likely to point up errors. If possible C_f should be chosen close to 1. A particularly important consideration is not to pick values which could cover up other errors. For instance, take

the formula

$$y = (a - 1)e^{(x+2)/2} + be^{-x^2}$$

If a value of 1.0 is used for a, the final answer is of course independent of the first exponential. The answer could come out correctly even with major errors in the exponential term. Similarly, a b value of 0 should not be used. Again, if x is chosen larger than 2 or so, the second exponential term will be nearly 0 and will have little effect on the result. A major error in the value of b might then go undetected.

The third reason for a detailed check case revolves around the importance of many computer applications. Some large problems may use literally hundreds of hours of computer time, which in itself is expensive. More important, the results may be the design of equipment or tests involving in some cases millions of dollars. This perhaps emphasizes that it is very worthwhile to expend considerable effort to be positive the answers are right.

The next general approach will become clearer when we discuss various machine methods: writing the code so it will be easy to check. This implies keeping intermediate storage in tidy blocks, if, as usual, memory print is to be the primary checkout tool. It also sometimes implies writing a less fancy code which will be easier to follow. This is a bit hard to illustrate; we may simply say that occasionally it is possible to use tricks in coding which save a few steps (and give a glow of inner satisfaction) but cause grief when the problem is put on the machine, particularly if anyone else has to assist in the checkout.

The third general approach is to make a detailed check of the code as written, before trying it on the machine. This is aimed at all types of errors: conceptual, logical, and stupid. Perhaps its primary mission, however, is to catch the stupid mistakes like wrong operation codes and addresses.

There are various ways of going about a detailed code check. The obvious way is for the person who wrote the code to go back over it, preferably after a few days' delay between writing and checking. The coder may simply read the instructions carefully, checking addresses, constants, loop testing, making sure operations are correct, ascertaining that the tape was rewound before reading after writing, etc. Some coders like to draw a second flow chart, working backward from the code, and compare it with the original flow chart.

A well-recommended technique is to have someone else do the checking, with or without drawing a new flow chart. The reason for this is the well-known fact that we tend to become somewhat entranced with our own mistakes. Reading stale code is uninteresting at best, and errors do not always exactly stand out from the page. This is true no matter who reads the code, but it seems to be a lot harder to check one's own work. The code is familiar, and there seems to be a strong tendency to skim over the instructions without really being critical of each little mark on the page. This corresponds exactly to the difficulty of proofreading one's own writing. A second person does not know what to expect, and is not partially blinded by what he knows *ought* to be written in a given instruction. He is more in a position of trying to read the code and understand *from it* what was done; this requires a much more careful reading than for the original writer to force himself to look it over, trying to find blunders.

It sometimes works out very satisfactorily for two people to work on a problem. If one is much more experienced, he will do most of the original work with the second doing mostly checking (which, incidentally, is a fine training method for the second person). If the two are equally qualified, both can write code and check each other's work.

This may seem like a large effort for the gain. As may become more obvious from the discussion in the next section, finding errors once a problem is on the machine is very expensive. It has been estimated that on a large machine of the IBM 704 or Univac Scientific class, each coding error costs between twenty and fifty dollars to find, including machine time and programmer time. The plan of using another person to check over codes before attempting the problem on the computer answers this high cost in two ways. First, the prechecking cost required to find an error, on the average, is not as great as the total cost of finding it later. This is because in most cases the computer costs much more per minute than a programmer does. The second economic consideration is that the checker is frequently not as highly paid as the original writer, nor as highly skilled.

Tests of accuracy can often be programmed, to be carried out along with the solution. One technique is to compute each important quantity twice, using different memory locations and different sequences of instructions, and compare the answers. If done consistently, this of course requires twice the computer time to get a solution. It is usually not done on machines which have extensive self-checking, such as Univac, or on machines which are felt for other reasons to be sufficiently reliable. Neither is it often done if there are other ways of accomplishing the same result with less effort. For

instance, after solving a system of linear equations it is not very time-consuming to substitute the answers back into the original equations to test whether they actually satisfy the system. It would be more precise to say, satisfy the system *within satisfactory limits*. Because of round-off and other errors inherent in digital solutions, the unknowns will almost never satisfy the system exactly, and some allowance must be made for this in programming the back substitution. The same comment applies to other examples of checking which are not so lengthy as computing the answers twice. After computing a long chain of tabulated values of some function, it may be possible to apply an asymptotic formula to check the last value. If the sine and cosine of an angle are computed in the course of a problem, the identity $\sin^2 x + \cos^2 x = 1$ can be programmed. The list could be extended.

The last general approach to the checkout problem which we will discuss is that of testing all possibilities in a program. Some applications involve many branches and forks all the way through; *any* problem has some alternatives built into it. An engineering calculation may specify that a certain parameter has a limit of one; if it is calculated as greater, *one* should be substituted. An accuracy check might be built into a problem to find the sum of the sine squared and cosine squared; if the sum is more than 10^{-7} away from one, the program should stop. Examples could be multiplied. The point is, of course, that a program is not necessarily free of all errors simply because it gets correct answers to one set of input. As far as feasible, all possibilities in the program must be checked. Sometimes the nature of the problem is such that this is almost impossible, particularly if contingencies can occur in pairs or triples.

This again points up a kind of error that may go undetected for months. A parameter is supposed to have a programmed limit of one, but the code is wrong so that nothing happens if it is actually greater than one. For the first 6 months the program is run, this situation never arises; then a particular set of input results in the parameter going to 1.04. It may take days to track the trouble down.

Checkout is perhaps a fourth or a third of the total cost of problem preparation prior to first production running. It is costly and time-consuming enough to be worthy of more careful planning than it often receives.

13.2 Machine Checkout Methods

There are several ways to make the computer itself assist in the checkout procedures. Two of these are simple adaptations of machine features; the others make use of a special-purpose program which must be in memory along with the program being checked.

The most elementary machine checkout technique is the use of the single step key. It will perhaps be remembered that with the automatic-manual switch set to manual, pressing the single step key causes one instruction to be executed. This will be in normal sequence or a jump, depending on the instruction in the current instruction register. It is possible also to make a manual jump to any location via the enter instruction key. Single-stepping consists of getting the first instruction of interest into the current instruction register, then repeatedly pressing the single step key and watching what happens. In many cases it is necessary to copy down certain information from the registers, such as the actual addresses and contents of index registers. In some cases the arithmetic may be verified against a hand-calculated case.

This is unfortunately an extremely expensive method and is seldom used except on very small sections of programs in order to find errors which resist more conventional attacks. Besides the high cost of machine time, the method has the disadvantage that there is no permanent record of what is happening. Frequently in such cases, the programmer takes his scratch paper back to his desk to analyze, only to discover that there was one critical piece of information which was not copied down.

This is obviated by a tracing program, the next level of sophistication in checkout. As the name implies, this is a program which must be in memory along with the program being checked. Its purpose is to record on tape or printer or punch all the information which a person using single step might record. A fairly typical tracing program for TYDAC might punch out on a card the following information:

Location of instruction.
Operation-address-index control, i.e., current instruction register.
Contents of accumulator.
Contents of MQ.
Contents of memory location specified by address part of instruction.
Contents of index registers.

All of the arithmetic information is punched or printed as it appears after the execution of the instruction specified by the location counter and the current instruction register. All of the information can be obtained in much less time than it takes to punch it, so that tracing can proceed at full punching or printing speed. However, this is still

vastly slower than the high-speed operation of the arithmetic section. The IBM 650, for instance, can punch only 100 cards per minute, whereas arithmetic can be carried out at an average of perhaps 25,000 operations per minute. Another way of illustrating this is to point out that the single loop used as the first example in Chapter 6, which simply added fifty numbers, would take almost 10 minutes to trace at 100 cards per minute.

We need not be concerned as yet with the details of how the tracing program operates; it is an interpretive method, as discussed in Chapter 15. It is controlled by jump instructions just as the program is; in other words, it gives a record or *trace* of what went on at each point through the program. It is subject to the rather obvious limitation that no sequences of instructions may be traced which depend on timing of mechanical parts. For instance, a Select instruction must be followed within a certain time limit by a read or write, so there would be no time to trace these.

Since tracing is so slow, it is often desirable to make it a selective process, where only some of the instructions are traced. It is not too difficult to design the trace program so that it automatically skips over input-output instructions where there would be timing problems. The program can be designed to accept information as to where to start tracing, such as the location of the first instruction to be traced. The information can be more complete, consisting of a table of regions of instructions to be traced. In some machines, instructions may have either a plus or a minus sign. This choice can often be used to control tracing; instructions might be written with plus signs if they are to be traced, minus if not. This demands foresight in the code writing, since what will be selectively traced must be established in advance. A usual procedure would be to plan to trace the instructions which compute significant intermediate answers. It is possible to change the signs of the affected instructions during checkout, but this is not too convenient.

The procedure allowed by some trace programs, of specifying regions to be traced, gives more flexibility. The regions to be traced are usually signaled by punching initial and final addresses on control cards. These can quite simply be inserted in the deck, or removed, as the checkout progresses and different sections of the program become of primary interest.

Tracing is not only expensive but it very often does not give all the information needed. A trace may show that a certain loop contains wrong addresses. The programmer sitting at his desk with no more information than the trace may wish to know what happened in an early section of the program which set up the initialization. Since it is almost never possible to trace an entire program—which might literally require hours of computer time—he probably has no printed record at all of what happened in that initialization. With the clue of where the error is, he may be able to go back to the instructions in question and figure out the trouble without the printed record, but often he cannot. What is really needed is a listing of what the instructions in question looked like after the trouble arose.

A listing of a consecutive section of memory, either numbers or instructions, is called a *memory print,* or often, a *memory dump.* It has the characteristic of some tracing programs of being highly selective: as little or as much of memory may be dumped as may be pertinent. At any one point it gives a great deal more information than tracing does. A significant point here is that the memory dump does give information *at one point.* In this sense it is fundamentally different from tracing. We may say that tracing gives a dynamic record of what happened at each instruction as the program was executed. It is a sort of *vertical* record: a little information at many points. The dump is a sort of *horizontal* record: it gives a complete cross section consisting of much information at a few points. This often requires a partial memory dump at several points through the program.

The break point switch is frequently a help in such situations. To review, this is a rotary switch on the console which may be set to one of ten positions, 0 through 9. A test instruction is used to "interrogate" this switch. If the switch is set to 3, say, and the instruction

Break jump $a, 3$

is given, the machine will stop. When the start button is pressed, the next instruction is taken from a. If the switch is set to any other position, the next instruction in normal sequence is taken, and there is no stop.

This instruction may be used to control a memory dump by inserting Break jump instructions at points in the program where information is needed. The switch can be set to different positions as checkout proceeds.

To understand this more fully, we must discuss how memory dump programs operate. One fairly common technique requires the initial and final addresses of a region to be entered manually into the MQ. Under this plan, when a Break jump stops the machine, the operator specifies the region(s) to be printed or punched and makes a manual jump to the start of the dump program. The dump program stops after completion, at which point the operator makes a manual jump back to the instruction following the Break jump. Depending on the circumstances, he may change the setting of the break point switch before starting the program again, to prepare to get another dump when and if the program arrives at another critical juncture. In this case the important feature of the Break jump is not the jump, but the stop.

Another way of using the instruction requires more foresight as to what information may be needed during checkout, but saves considerable console fiddling. It is possible to print or punch sections of memory by using a program called from memory. The plan is to have in memory several calling sequences to dump sections of storage; the addresses in the Break jump are the addresses of the appropriate calling sequence. At the end of each group of calling sequences is an Unconditional jump back to the instruction following the appropriate Break jump. All the operator has to do now is push the start button when the stops occur, and possibly change the setting of the break point switch before doing so.

These break points, at which memory is dumped or other information is obtained, are simply critical spots in the program, at which a little information may tell a great deal about the process.

A memory dump may be used in two ways. We have discussed how to use it to get information during the execution of a program. It is also employed to get information after completion of a program, or when it unexpectedly "dies" during a checkout run. In either situation it is desirable to have a complete listing of all numbers and instructions to use as reference in tracking down troubles. It is also very desirable to have the contents of all registers at the time of the breakdown so that the immediate fault can be pinpointed.

As a final indication of the time comparison between tracing and dumping, we may observe that the entire 2000 words of TYDAC could be punched in the format discussed in Chapter 11 in 3 minutes at 100 cards per minute. In the same time, only 300 instructions could be traced, which in a program with extensive looping would be a very small part of the total number of instructions to be executed. And even in the same time, the memory dump would give much more pertinent information. We may safely say that memory dumping is a much more sophisticated approach to the checkout problem. A common tendency is for new programmers to learn tracing first, and to misuse it very badly. The experienced programmer uses tracing occasionally, for small parts of a program, but only after other methods have failed to locate the trouble.

Special-purpose diagnostic programs can be designed which combine the best features of all these techniques, and at the same time allow quite flexible control. Some of these depend on special features of machines;* one will be described which could be used on any machine, and the TYDAC in particular.

A reasonable name for the program might be "dynamic diagnostic." It has the following characteristics:

1. Small regions to be traced may be specified by an initial and final address entered on a card.

2. At any point in the program which is not limited by timing requirements, a memory dump of several sections of memory may be called for.

3. At any memory dump a simple code, punched on the card which calls for the dump, will call for punching out the contents of all registers.

4. Items 2 and 3 may be done only a limited number of times through a loop, or not until a specified number of times through, according to a code punched in the same card.

5. Provision may be made to get certain information if the program breaks down—which we might call "post-mortem" information. By entering a "post-mortem control" card, we may ask for the contents of the arithmetic and control registers, a dump of specified sections of memory, and a count of the number of times certain instructions were executed.

All of the above except the last are carried out automatically during execution of the program. There are no stops as when the branch switch is used, but the execution of the program is slowed down by any tracing or dumping and by internal red tape operations associated with counting the number of executions, etc. The fact that much of the operation of the diagnostic work is carried on simultaneously with the operation of the program being checked is the basis of the word "dynamic."

The post-mortem information is wanted only after the program breaks down, and since this cannot be predicted, a manual jump is required to start the post-mortem punching. The dynamic diagnostic program itself would occupy several hundred storage locations. This space could not be used unless there was information to go into the program which would not be needed during checkout and which could be entered later. This would not usually happen, so we must admit that we have given up a significant fraction of memory to the checkout problem.

* Such as the trap jump on the IBM 704.

To be more specific about how such a program would work, the details of input and output will be discussed.

The deck for dynamic diagnostic checkout would consist of:

1. The card loading program discussed in Chapter 11.

2. The deck being checked out, including all data.

3. The diagnostic program, which would load on normal cards. The last card would be a transition to the dynamic diagnostic program.

4. The program transition card.

5. The various control cards for dynamic traces or dumps, plus any post-mortem controls.

All these would load at full speed before any diagnostic procedure began. The program transition card and the control cards would be loaded by the diagnostic program; the program transition card would not immediately be executed, but the information on it would be stored for use after the controls were loaded. It is important to note that these control cards are taken to *be* control cards only because of their position in the deck, and because they are loaded by a special input program in the dynamic diagnostic.

The output of the program is fairly straightforward. In TYDAC it would be a deck of cards which would be listed on a tabulator. Any sections of tracing would be obvious enough. Dumps would be in standard memory output form with initial addresses and seven numbers per line. Contents of registers would be punched out in much the same format as a line of tracing. Counts of the number of times an instruction was executed would punch with the instruction location in the address part and the count in the index control part.

13.3 Conclusion

There are many programs available for actual machines to assist in checkout. Properly used, they can save much personnel time and practically eliminate aimless console fiddling. These programs will become more and more sophisticated as experience is gained in their use.