WHAT SHOULD WE COMPUTE?

F. J. Gruenberger

September 1963

P-2786

WHAT SHOULD WE COMPUTE?

F. J. Gruenberger*

The RAND Corporation, Santa Monica, California

If computers are to be used efficiently (that is, in a manner that is healthy for both the user and the vendor), then we should define what constitutes a good computer problem. It seems patently obvious that if the wrong problems are put on the machines, inefficiency--or bankruptcy--will result. Fortunately, many problems are good computing problems intrinsically and intuitively, so we stay out of trouble by good luck. As with many things, the extremes are obvious to any dolt; the purpose of analysis is to allow us to discriminate between the borderline cases. For example, payroll calculations, or the solution of large systems of simultaneous equations, are fine computer problems. On the other hand, a simple tally of the results of a questionnaire is not a computer problem, nor is the calculation of 17^5.

In the cold practical world, the first criterion of a good computer problem is utility. There must be some value (to somebody, usually the man who foots the bill for the

---

*Any views expressed in this paper are those of the author. They should not be interpreted as reflecting the views of The RAND Corporation or the official opinion or policy of any of its governmental or private research sponsors. Papers are reproduced as a courtesy to members of its staff.

This Paper was prepared for presentation at the Summer Institute for Secondary School Mathematics Teachers, University of Oklahoma, July 11, 1963, and will be submitted for publication in Datamation magazine.

machine) to the results we are trying to get. So our first rule is:

1. Usefulness.

If we are only learning the art of computing, of course, then Rule 1 can be waived. If some useful technique can be driven home, then something as inane and useless as counting to a million might be a good problem.

From here on, we can develop criteria that apply to any use of a computer, whether practical or in a classroom.

2. Definition.

The problem must be defined; that is, we must know precisely what the problem is. It's an "obvious" criterion, but many a man-hour has been wasted on a plunge into programming an undefined or ill-defined problem. Consider the warehouse manager who innocently asks to have his inventory problems "put on the computer." Or the student who asks, "Can we 'do' roulette on the machine?"

The old hand learns to keep asking, "What is the problem?" In a specific situation, he may ask it to the point of open rebellion on the part of the customer. He appears, to the customer, sort of stupid; after all, it's perfectly clear to the customer what his problem is. Or is it? Our short history is already loaded with sad tales of people who never did get their problem properly defined.

So definition is necessary. It is hardly sufficient. Few problems are better defined, for example, than Fermat's Last Theorem.

If we know precisely what the problem is (and we should continually emphasize "precisely"), then we need a method. To be high-sounding about it:

3. Algorithm.

In other words, we must know some way to solve the problem, with or without a computer. Note that we say some way. There are usually many ways. Here may be the first distinction between a computer solution to a problem and any other solution. We are privileged to keep in mind that a computer is fast; we may capitalize on this fact to seek, as a first cut, a brute-force (or crowbar) method. As a criterion, it matters little. What is vital to understand is that we must have an algorithm that works, at least in theory.

Here, of course, is where the theorem of Fermat fails as a computer problem. We don't have a method, with or without a computer. Often a problem can be solved by exhaustion, by which we attempt to try all possibilities. This is frequently acceptable as a method to use on the computer.

OK; we know what our problem is and a way to solve it. We should now check for

4. Machine match.

This sounds like the most trivial criterion of all; namely, that our problem should fit the machine we propose to solve it on. It must fit in two ways: the instructions plus data must fit the storage capacity of the computer, and the running time to solution must fit what time is available.

Brute-force methods may kill us here. Even in a classroom atmosphere (perhaps especially in that atmosphere), computer time is not unlimited, and is almost never free.

In a well-run computer course, the student does many exercises. He should also do at least one problem. The distinction is this: an exercise relates to a specific technique, and the approach is usually spelled out. A problem, on the other hand, will involve a broad goal, using many techniques, and with very little spelled out. Part of the student's task is to choose a suitable problem; that is, one suited to his own capabilities and to the machine time available. Hopefully, it should also be a good computer problem. The instructor must prevent him from going to either extreme (i.e., too trivial a problem, or one he cannot hope to complete) and should also guide him to a problem that is, according to the criteria we are discussing, a problem worth putting on a computer. All such situations can be covered by furnishing a stock collection of problems. This will satisfy 90% of any class. But the best situation of all is the one where the bright student selects a problem all his own; perhaps one never tackled before. The checklist we are developing might be of some help.

5. Repetition.

We make the statement categorically: a good computing problem has a large element of repetition. This rule has many exceptions, of course. The use of a computer for true one-shot calculations is fairly common, but that doesn't mean that one-shot problems (and particularly straight-line formula evaluations) are good computer problems. One thing

a computer does well is repeat a sequence of instructions (with or without modification). A good problem, then, capitalizes on this capability.

6. Payoff.

The sixth criterion refers to overall efficiency. It does not intersect with item (1), Usefulness. The question is this: is a solution by computer more efficient, in some sense, than a solution by any other means? For a good computer problem, there should be a large payoff.

Let's consider items (2) through (6) in relation to an actual problem. Consider the series of numbers shown in Fig. 1. In the May 1950 issue of The American Mathematical Monthly there appeared a proof of the following theorem: For any integer, r, there exists a power of 2 each of whose last r digits is either 1 or 2. The proof given is analytic, of course, and merely establishes that such powers exist. The mathematician is satisfied at having proved the existence. The student of computing has found a good problem.

From Fig. 1, we can fill in the first four lines of the table of Fig. 2. For r = 1, the first power of 2 satisfies (that is, its last one digit is either 1 or 2). For r = 2, the first occurrence is the ninth power. For r = 3 and r = 4, the 89th power satisfies, and so on. The computer problem is to extend the table to, say, r = 20. From Fig. 1 we could fill in the table only through r = 4.

Except for usefulness (and let's face it; the market for the table of Fig. 2 is sluggish, or even nonexistent), this is very nearly a perfect computer problem.

x	2^x
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
.	.
.	.
.	.
.	.
53	9007199254740992
.	.
.	.
.	.
.	.
89	618970019642690137449562112

Fig. 1--Part of a Table of Powers of 2

r	x
1	1
2	9
3	89
4	89
5	589
6	3089

Fig. 2--Some Results

The problem is well defined; it can be understood by a sixth grader.

A method of solution leaps to mind; namely, extend Fig. 1 and search each entry for r ones or twos at the low-order end. Developing the powers of 2 is easy; each new power is found from its predecessor by doubling. The searching process may take a small loop. There will have to be some housekeeping and perhaps a simple restart procedure. All told, it might take 50 or 60 instructions on a machine like the IBM 1620.

The problem fits the machine; in fact, it will fit <u>any</u> computer built. We defined the limit of the problem to be when r = 20. This means that we do not need to generate the complete powers of 2, but only the low-order 20 digits of each power. With a need for storage of only 20 decimal digits of data, and less than 100 instructions, any computer can work on this problem. We might note in passing that the problem is intrinsically decimal; if it is to be done on a binary machine, the machine would have to be programmed

to operate decimally. This would add a small amount of data and a few instructions.

There is a large element of repetition; we will use our few instructions many millions of times, as the problem is so far defined.

Finally, the payoff is tremendous. One might proceed, perhaps, to r = 8 with a desk calculator, but not much farther (the entry for r = 11 is 8128089). On the 1620, the solution as outlined above runs at about 400,000 stages per hour. To run to the power where r = 11 would consume some 20 hours of machine time. This is intolerable. It is time to apply a further principle of good computing:

7. Brains over brawn.

Offhand, this does not seem to parallel items (1) through (6), especially as a criterion. Nevertheless, it is a vital element in many computer attacks to a problem. It is axiomatic to experienced programmers that when a problem has gone through all its stages (analysis, flow-charting, coding, debugging, testing, and production), <u>then</u> you know for the first time how the problem should have been solved. Nearly always, one wishes he could tear the whole thing up and start over. Usually this is not practical, and the inefficient and unsatisfying solution is used for production runs.

The situation we have met in our sample problem is not unusual. We have gone on the machine with a workable solution to a simple problem, but it is obvious that this solution will never fulfill our goal to find the value of x for r = 20. We must find a way to speed up the solution.

We can't speed up the machine, but we can apply item 7.

Refer again to Fig. 1. The units digit of the powers of 2 follow a cycle that repeats as follows: 2, 4, 8, 6, 2, 4, 8, 6, ..., endlessly. If the numbers we seek must end in 1 or 2, then only every fourth number is even eligible. If we search every power, then three out of four searches are automatically wasted. Not only do we not need to search, we need not even generate the intervening powers. We could advance in one step from, say, the fifth power to the ninth power with one multiplication (by $2^4 = 16$) rather than with four doublings. Just this simple change speeds up the solution by a factor of four.

We can go further. If we examine the sequence formed by the last <u>two</u> digits (jumping by steps of four in the original series), we find again a repeating cycle; namely, 32, <u>12</u>, 92, 72, 52, 32, and so on. Again, only one term of this series satisfies the conditions of the problem; namely, the term 12. The original series, then, repeats in its last two digits with a cycle of 20, and we can jump 20 steps at a time by multiplying by $2^{20} = 1048576$. Well. Our problem could now run to r = 20 (on the 1620) in about twelve million hours.

Without dwelling on the details of discovery, let us state two theorems that can now be applied.

First, the principle hinted at above is quite general. The low-order digit repeats with a cycle of four, which is obvious on inspection. The two low-order digits repeat with a cycle of 20. The three low-order digits have a cycle of 100, and each additional digit from there on increases the cycle length by a factor of five. (The proof can be found

in the June 1951 issue of The American Mathematical Monthly.)

It can also be proved that the terminal sequence of digits consisting entirely of ones and twos is unique. That is, when we find (for x = 89) that the last four digits are 2112, then we know that for a longer set of satisfactory terminal digits, the last four will be 2112. Similar statements can be made for the last r digits, no matter how great r is.

Our reasoning now goes like this. Let's go on the machine and type the low-order digits of every fourth power until we achieve success (namely, on x = 9, when the terminal digits are 12). At this point, we want to start jumping by steps five times as big; namely, by 20's. We now look for the entry that has its last three digits all ones or twos. We are, at this point, examining only those powers which all have their last two digits satisfactory. The digit just to the left (called the critical digit) will maintain its parity; that is, if it starts out odd it will stay odd, and if it starts out even it will stay even (this is part of the original proof, May 1950). We are thus assured of success, at the next higher level for r, within the next five lines.

Each time we achieve success at the next level, we can increase the jump rate by a factor of five and get our next success within, at most, five stages.

In other words, our original attack was all wrong. We need a routine that will develop and type the terminal 20 digits of any arbitrary power of 2 and go on to higher powers by an increment that we can alter by a factor of five when success is achieved.

This is readily done by calculating first a table of

powers of powers of 2, as in Fig. 3. Fifty lines of this table (the argument is formed by doubling; the function by squaring) suffice to carry our problem to its conclusion.

We can now go on the machine with starting values of, say, $x = 89$ and $\Delta x = 100$. We know from our experience with the problem that we have reached $r = 4$ and that we can afford to jump by 100's. A search-and-multiply loop on the table of Fig. 3 will produce for us the last 20 digits of any arbitrary power of 2. For example, if we want the 89th power, we need to multiply together the functional values corresponding to the arguments 64, 16, 8, and 1 (precisely the binary representation of 89, of course).

Argument	Function
1	2
2	4
4	16
8	256
16	65536
32	4294967296
64	744073709551616
128	607431768211456
.	.
.	.
.	.

Fig. 3--A Table of Powers of Powers of 2

We will arrange also to alter Δx by a factor of five whenever we please. We are now ready to really solve our problem. We will generate the 89th, 189th, 289th, 389th, and so on, powers (all of which end in the digits 2112) and wait for the appearance of a terminal sequence of more than four digits that are all ones or twos. We know that this will occur after, at most, five lines. At the time we will alter Δx by a factor of five and proceed. The process repeats from then on. In about ten minutes (on the 1620) we can develop the table shown in Fig. 4.

We have now exhausted the problem (well, almost) from the computing point of view, too. At least we now have a method that could extend Fig. 4 a considerable distance with only modest expenditure of machine time. There are still, however, one or two loose ends.

(1) We don't quite understand what goes on at the stages where r = 3 and 4, r = 6, 7, and 8, and r = 9 and 10. We get more than we should and it upsets the orderly pattern of things (but for the best).

(2) As a matter of pedagogy, a good computing problem should have one more feature:

8. Transfer.

We ask students to work on problems with the hope, however faint, that they'll learn something that can be applied to the next problem. Therefore, consider another problem. Refer again to Fig. 1 (p. 6).

The tenth power of 2 contains a significant zero, and is the first such power. The 53rd power is the first to exhibit two consecutive zeros. We are building up the

r	x
1	1
2	9
3	89
4	89
5	589
6	3089
7	3089
8	3089
9	315589
10	315589
11	8128089
12	164378089
13	945628089
14	1922190589
15	11687815589
16	109344065589
17	231414378089
18	1452117503089
19	4503875315589
20	65539031565589

Fig. 4--More Results of the Terminal 2^x Problem

table shown in Fig. 5.

As one might expect, the theorem implied here can be proved. Given any integer r, it can be shown that there exists a power of 2 containing r consecutive zeros.

This sounds like much the same problem. It fits the criteria in the same way, for example. And as before, a method leaps to mind: Generate the successive powers and search for strings of zeros. And, alas, there the resemblance ends. We have found no way yet on this problem to apply principle 7.

Moreover, the mechanics of solution is much worse. We must develop all of each successive power (rather than just the last r digits), and so the time involved in creating the numbers and searching them goes continually up. At x = 11000, the process runs about 5 powers per minute on a Model I 1620, or about 40 powers per minute on a 7094. Until principle 7 comes into play, the problem is somewhat hopeless.

r	x
1	10
2	53
3	242
4	377
5	1491
6	1492
7	6081
8	(greater than 11088)
.	.
.	.

Fig. 5--The Problem of Consecutive Zeros in 2^x