On Proving a Program Shortest*

Arindama Singh

We revisit a problem faced by all programmers. Can one write a program that determines whether any given program is the shortest program? How does one prove that a given program is the shortest? After answering these questions, we discuss very briefly the Kolmogorov complexity of a string of zero and one, which leads to a barrier on any axiomatic system, known as Chaitin's barrier.

Arindama Singh works as a professor in mathematics at IIT Madras. His teaching and research interests include theory of computation, logic, and linear Algebra. He has authored three books in these areas.

Keywords
Four types of programs, simulating programs, input bit-string, input-program, length of a program, complexity, Chaitin's barrier.

Introduction

Consider writing a program that computes the function

$$f : \{1, \ldots, 20\} \to \{1, \ldots, 6\},$$

given by the following table:

n	:	1	2	3	4	5	6	7	8	9	10
$f(n)$	:	1	5	2	5	2	6	2	6	3	0
n	:	11	12	13	14	15	16	17	18	19	20
$f(n)$	:	3	0	4	0	4	1	4	1	5	2

We can write a program by giving this table as an input so that $f(n)$ can be computed for a given n by just reading the table appropriately. However, there is an alternative. It is easy to verify that the function f can be specified by

$$f(n) = [(3.7 \times n) - 2] \mod 7 \quad \text{for } 1 \le n \le 20.$$

It is the remainder obtained by dividing the integral part of $(3.7 \times n) - 2$ by 7. Is there a shorter way of specifying the same function?

In fact, we are interested in more general questions.

*Vol.25, No.9, DOI: https://doi.org/10.1007/s12045-020-1043-6

Strong Question: How do we prove that a given program is a shortest program among all those doing the same job?

Weak Question: Can we write a program which will determine whether a given program is a shortest program among all the programs doing the same job?

1. Preparation

To answer these questions, we need a little preparation as to fixing some notions, etc. We fix a computer and a general-purpose language for this computer, be it C, Java, Python, or Lisp. Though we give input to the computer and get some output from the computer by executing a program, we say that the program has taken an input and gave us that output. Based on this, we will deal with four types of programs:

1. Programs that take inputs and give outputs upon halting.
2. Programs that take no inputs but give outputs upon halting.
3. Programs that take inputs, do not halt and give outputs time to time.
4. Programs that do not take inputs, do not halt, and give outputs time to time.

There are programs that can take other programs as inputs and run an input-program.

In fact, inputs and programs are all stored in the computer as bit-strings, that is, finite sequences of 0 or 1. And all our programs can take bit-strings as inputs. Thus all programs can take other programs as inputs. That is, there are programs that can take other programs as inputs and run an input-program. Such programs may be called as *simulating programs*. If the input-program needs another input, then our simulating program can take inputs in the form of a pair of bit-strings, where the first bit-string is another input-program, and the second bit-string is an input to the input-program. Then the simulating program runs the input-program on the input bit-string. It outputs the output of the input-program when run on the input-string. We would encapsulate this scenario as follows:

There exist programs that run other programs.

Notice that bit-strings can be ordered; we fix the following ordering:

0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 000, 101, 110, 111, ...

Some of these bit-strings are programs, and others are not. According to our fixed language, it is syntactically determined as to which bit-strings are programs, and which are not. Thus, the syntax of the language can be written as a program for determining and enumerating all those bit-strings which are programs. Therefore,

There exists a program that enumerates all programs.

Each program has some length. The *length of a program* is the length of the bit-string that represents the program. Since bit-strings can be ordered in a particular way, the same ordering is used to enumerate all programs. Further, given any natural number n, from the enumeration of all programs, we may discard all those bit-strings which are of a length less than or equal to n. That is,

There exists a program that enumerates all programs of length bigger than a given natural number n.

Notice that such an enumerating program takes a single input n; it does not halt, and it gives us outputs as programs in the particular ordering, time to time. As we see, the outputs of such an enumerating program come in an orderly fashion. The order is given as follows:

An enumerating program takes a single input n, it does not halt, and it gives us outputs as programs in the particular ordering, time to time.

Programs of smaller lengths are output first; and among programs of the same length those come first in the above ordering of the bit-strings, come first.

Such an enumerating program can be easily modified to wait for another bit as an input after printing each output. Hence,

> There exists a program which takes input as a natural number n; outputs a single program of length larger than n; waits for another time-to-time input which may be either 0 or 1. If this input is 0, the program halts; and if this input is 1, then the program prints the next program of length larger than n. It continues this way unless the time-to-time input is given as 0.

We see that programs are bit-strings. A job is specified by a function, which maps inputs to outputs. For each job, which is a computable function, there exist possibly many shortest programs. And, there are infinitely many jobs. Thus,

> There exist infinitely many shortest programs.

There exist infinitely many shortest programs. There exist only finitely many shortest programs for doing the same job.

We should not misunderstand this; there exist only finitely many shortest programs for doing the same job.

2. The Weak Question

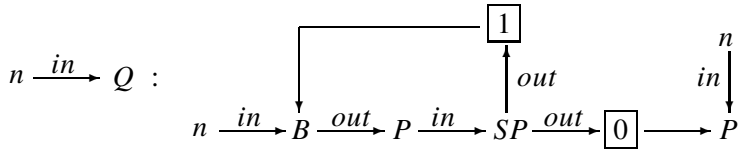
We try to answer the weak question first. Assume that there exists a program, namely, SP, which takes a program P as an input and outputs 0 if P is the shortest program among all those which do the same job as P; else, it outputs 1.

Let B be the program that takes input as a natural number n, enumerating all the programs of length larger than n, in a time-to-time fashion. That is, it prints the first program of length larger than n, and then waits for a bit input. Once this input is given as 1, it goes to print the next program of length larger than n. If the input is 0, then B halts.

We construct a program Q which takes a natural number n as its input, and does the following:

Q first simulates B with the same n as an input to B. Then Q passes the output of B to SP. Next, Q passes the output of SP to B, and so on until B halts. Once B halts with a program P as its output, Q runs P with the same input n.

The following schematic diagram shows the structure of the program Q:



Here,

$\square \longrightarrow y$ means 'run the program y',
$x \xrightarrow{in} y$ means 'run the program y with input as x',
$y \xrightarrow{out} x$ means 'output of the program y is x'.

The details of the execution of Q are as follows. Q runs B with input to B as the natural number n. Recall that B is supposed to print all the programs of length larger than n one by one following the ordering of the bit-strings. So, B prints the first program P of length larger than n, and waits for an input which may be either 0 or 1.

If SP finds that P is not the shortest program, it outputs 1. Then Q passes this output to B. B prints the next program of length larger than n.

If SP finds that P is the shortest program, it outputs 0. Then Q passes this output to B. B then halts. Once B halts, we see that SP has found the shortest program P. Now, Q runs P with the input n, producing the same output as P does.

Once B halts, we see that SP has found the shortest program P. Now, Q runs P with the input n, producing the same output as P does.

Since Q simulates B and SP, we summarize the execution of Q as follows:

Q is a program that takes a natural number n as its

input; it eventually finds the shortest program P of length larger than n, and then it runs P producing the same output as P.

Suppose the length of the program B is b, and the length of the program Q is q. Suppose the syntax for giving input to the program takes c bits in our language. Then take $k = 2^{2^{b+q+c}}$. Run Q with k as the input.

Since there exist infinitely many shortest programs, B eventually finds one of length larger than k. So, suppose B finds the program P, which subsequently, is determined by SP as the shortest program. Now, we know that P is the shortest program, whose length is larger than k. This means that for doing the same job as P does, there exists no shorter program.

We have a program in our language, namely, Q along with the input k which does the same job as P.

Notice that Q does the same job as P with input as k. We see that length of k in bits is 2^{b+q+c}. Thus, we have a program in our language, namely, Q along with the input k, which does the same job as P. Now, the length of our program is at most Q, plus the length of k, plus c, which is $2^{b+q+c} + q + c$ which is much smaller than k.

This contradiction proves the following result.

Theorem 1. *There exists no program to determine whether or not a given program is the shortest.*

3. The Strong Question

Suppose we have written a program for doing a particular job. We have an intuitive feeling that this might be the shortest program for doing that job. Since we know the length of our program as a bit-string, our strong question may be formulated as follows:

> How do we prove that for a given computer, with a given language, for doing a given job, the shortest program will have at least n bits?

Observe that given any axiomatic system, its axioms and rules of inference can be encoded in a program so that the program generates all theorems one by one. So, suppose T is the theory of all programs. Let P be a program that generates all theorems in the theory T. Let S be the statement:

> For the computer C, with the language L, the shortest program for doing a job J will have n bits and no less.

Suppose that S has a proof. Then the program P will print this sentence S sooner or later. If S does not have a proof, then P will never print it. However, we may not be able to know that P would never print S. But the programs of length less than n bits are finite in number. Thus, they can all be enumerated and checked whether any of those is meant to do the same job or not. So, let Q be this program that generates and checks all the programs of length less than n bits.

Now, combine the programs P and Q to write a new program which runs P and then Q, next P followed by Q, and so on. This new program now determines whether or not there exists a shortest program for doing the same job. This contradicts Theorem 1. Therefore, our assumption that S has a proof is wrong. We obtain the following result.

Theorem 2. *There is no way to prove that a program is the shortest for doing an arbitrarily given job.*

It is quite possible that for a particular job, the shortest program is indeed the shortest. For example, "print 0" may be the shortest program in some language to print the symbol 0. Theorem 2 asserts that there is no uniform method of proof that might show that any given program is really the shortest.

Theorem 2 asserts that there is no uniform method of proof that might show that any given program is really shortest.

4. Chaitin's Theorem

In fact, Chaitin proved a generalization of these results by defining the complexity of a bit-string. To get an idea of his result,

consider the following bit-strings having some finite lengths (not shown here):

1101110111011101...; 10110011100011110000...;

10110000001101111100...

The first string has a pattern: 1101 is repeated. The second string has a pattern:

10 1100 111000 11110000...

The third string does not show any obvious pattern. We do not know whether it follows any pattern or not. Now, to produce the first string as an output of a program is easy. Just print 1101 repeatedly. The second string can also be printed though in a bit complicated manner. But in the absence of any pattern, the third one can be printed only by specifying "print that-string-itself". Observe that finding the shortest program to generate a bit-string has obvious applications to zipping a file.

We may define the (Kolmogorov) *complexity* of a bit-string as the length of the shortest program that prints the string.

We may define the (Kolmogorov) *complexity* of a bit-string as the length of the shortest program that prints the string.

Then the complexity of the first string is $4 + c_1 + \log_2(n)$, where c_1 is the length of the added instructions needed to give the print command, and n is the length of the required bit-string, etc. Notice that along with the pattern encapsulated in the program, the length n should be given as an input. Here, c_1 does not depend on the string.

The second string has the complexity $c_2 + \log_2(n)$, where c_2 is the length of the instructions for describing the pattern and the print command, and n is the length of the required string. Here also, c_2 does not depend on the string.

For the third string, the complexity is $c_3 + n$, where again c_3 is the length of the print command that does not depend on the string, and n is the length of the string. In this case, we need to give the string as a built-in input in the program.

Thus, the complexity of each string is at most $c + length(string)$ for a constant c. There are 2^n strings of length n, and at most 2^{n-k} programs (also bit-strings) of length less than $n - k$. Thus, the number of strings of length n and complexity at most $n - k$ decreases exponentially as k increases. Therefore, there are very few strings of length n whose complexity is much less than n.

That is, the majority of strings of length n have complexity approximately n.

Since we have an ordering of the strings, consider the first string that can be proved to have complexity greater than one billion. Now, this description itself can be used as a program to generate such a string. But the length of such a program is clearly less than one billion. Using our earlier discussed method of generating proofs, we see that:

> There exists a program of length $\log_2(n) + c$ bits that computes the first string that can be proved to be of complexity greater than n.

Here, the $\log_2(n)$ term comes from the requirement that n should be given as an input to our program in the built-in form. This contradicts a large n, since such a string has complexity $\log_2(n) + c$, whereas, it is supposed to be greater than n. Therefore, such a string does not exist. Thus we obtain Chaitin's theorem:

Theorem 3. *For all sufficiently large values of n, it cannot be proved that a particular string is of complexity larger than n.*

In fact, Chaitin goes further to prove the following theorem:

Theorem 4. *Suppose an axiomatic system can be encoded in LISP in N bits. Then there exists a natural number $L \leq N + 2359$ such that no proof in the axiomatic system proves that the complexity of any specific bit-string is of length more than L.*

The natural number L is called *Chaitin's barrier* for any axiomatic system.

The natural number L is called *Chaitin's barrier* for any axiomatic system. Observe that the constant 2359 is specific to the lan-

guage LISP. If another general purpose language is used in place of LISP, this constant will possibly change.

We see that most strings of large length are of complexity almost the length of such a string, but we cannot exhibit one such string.

There are many applications of Chaitin's theorem. For this, theory of programs, and the notion of proofs in an axiomatic system, see the references.

Acknowledgement

The author cheerfully thanks the referee for very constructive suggestions that improved the presentation of the paper.

Suggested Reading

[1] G J Chaitin, *The Unknowable*, Springer-Verlag, Singapore, 1999.

[2] G J Chaitin, *The Limits of Mathematics*, arXiv:chao-dyn/9407003v1/7 July 1994.

[3] J P Lewis, Large limits to software estimation, *ACM Software Engineering Notes*, Vol.26, No.4, pp.54–59, July 2001.

[4] J McCarthy, A basis for mathematical theory of computation, *Studies in Logic and the Foundations of Mathematics,* Vol.35, pp.33–70., 1963.

[5] A Singh, *Elements of Computation Theory*, Springer-Verlag, London, 2009.

[6] A Singh, *Logics for Computer Science, 2nd Ed.*, PHI, New Delhi, 2018.

Address for Correspondence
Arindama Singh
Professor of Mathematics
IIT Madras
Email: asingh@iitm.ac.in