On Computable Numbers

Hector J. Levesque
Dept. of Computer Science
University of Toronto
hector@cs.toronto.edu

September 27, 2022

I spent a lot of my career thinking about operations over symbolic representations of propositions: which ones were meaningful, which ones not; which ones were readily computable, which ones not. This led me to some thoughts on operations over representations of numbers, and the idea of computing with numbers.

Let's start at the beginning with Turing Machines (TMs). The thing is this: You can't put a number on the input tape of a TM; you can only put a sequence of characters taken from some predetermined alphabet. Typically, the characters are digits, and the sequence of characters is understood to be a *numeral* standing for a number. With an alphabet of two characters, you get binary numerals; with ten, you get decimal numerals, and so on.

So TMs do not really compute functions overs numbers; they only compute functions over strings of characters. We are free to read these strings as numerals, of course, but we might also interpret them as messages or digital images or whatever. When we say that a function over numbers is not computable (or not computable efficiently), this is just a sloppy way of saying that there is no TM that would compute some related function over strings of characters. It's actually the function over strings that is not computable (or not in polynomial time, say).

When we first learn about TMs, the distinction between numbers and numerals of various sorts is made clear to us. We are told about converting back and forth between binary and decimal numerals, and other representations. We often say "binary numbers" but this is sloppy too; it's the numerals not the numbers that are binary. (Incidentally, the adjective "numerous" is not right either; it's the number here that matters not the numeral.) We might look at a variety of numerals, but eventually settle on what might be called the *standard binary numerals*: a possibly empty sequence of 0 and 1 characters, with the most significant digits appearing first, and where the lead character must be a 1.

But everybody is supposed to see that this choice is not crucial. To take a simple example, we might have *reverse numerals*, like the standard ones, but with the least significant digits first. Interestingly, there are machines where the difference between these two representations is significant. There is a finite automaton (in the usual formulation) that can add two numbers represented as reverse numerals, but there is no finite automaton that can add two numbers represented as standard numerals. (Addition, it would appear, wants to start with the least significant digits.) So can a finite automaton add two numbers? For some representations of numbers it can; for others, it can't.

These sorts of considerations raise the prospect of other representations of numbers and what can or cannot be done with them. Consider this representation, which we might call *factor numerals*. It will use the characters 0,1,and \$. Any non-zero number n will be represented by a sequence $x_1\$x_2\$\ldots\$x_k$, where $k \geq 0$, and where the x_i are the standard binary numerals for the prime factors of n, sorted in increasing order. So, for example, 1 is represented by the empty sequence, and 20 is represented by the sequence 10\$10\$101 (that is, prime factors 2, 2, and 5). The interesting thing about this representation is that multiplication is now dead easy: just interleave the two sequences of factors. Again, it is a mistake to say that a TM cannot multiply two numbers in linear time (or test for primality, for that matter). For some representations, it can; for others, it can't.

The simplest case of something being hard for a TM is when a function is not computable. There are different ways to make this precise. Imagine an enumeration of all the TMs: TM_1,TM_2, etc. (One way to do this is to encode each TM as a number in some standard way, and then order these encodings.) This then leads to a famous result due to Turing: there is function F over binary numerals that is not computable. The F is defined as follows: given a numeral x, $F(x)$ is 1 if x represents a number that is the index of a TM that halts on all inputs, and 0 otherwise. Even though F is a well-defined function over strings of characters, there is no TM that calculates it.

It is tempting to then conclude something like this: there is no TM that can decide the halting problem. Tempting, but not quite right. Consider this representation of numbers, which we might call *magic numerals*. It will only use the characters 0 and 1. A number n will be represented by a string xc where x is a string and c is a single digit: the x will be standard binary numeral for n, and the c will either be 1 or 0 according to whether n is the index of a TM that halts on all inputs. With this representation of numbers, it is easy to decide if a number is the index of a TM that halts on all inputs. So is the halting problem undecidable? For some representations of numbers, it is; for others, it isn't.

But these magic numerals are ridiculous! It's cheating to use a representation of numbers that includes the answer we are after as an extra bit. (Incidentally, this

representation of numbers is not my idea. I was first told about it by my friend the philosopher Brian Cantwell Smith who, as far as I know, has not written about it.) But the real question though is this: Can we be clear and say why we think reverse binary numerals are fine as representations of numbers, but magic numerals are not? What are the rules of the game here?

We might be tempted to say this: You can use any representation of numbers as long as it can be calculated from a standard one, like binary numerals, using a TM. This would rule out magic numerals, and allow all the other representations we have seen. But here's the objection: Who decided that binary numerals were going to be the gold standard? Why don't we start with magic numerals instead, since we can calculate all the other numerals from them too (and do more)?

The problem goes back to numbers. We feel that we know how to write a number as a binary numeral, even if it takes some work, but not as a magic numeral. We want a representation function r that maps numbers to strings that is, in some sense, computable. The difficulty here is that we can't really talk about computing r since r takes *numbers* as arguments, and this was our original problem. A function r might be computable for some representations of numbers but not for others.

So is there not a way to talk about representation functions for numbers without talking about numerals of any kind?

I think there is. Here is one way. Suppose we are going to use characters drawn from a finite alphabet A, and we have a function r from $\mathbb{N}$ to A^*. We want to know if r is a proper representation function. It needs to be injective (or 1-to-1) of course, so that every number gets its own representation. To rule out bizarre things like magic numerals, let z be the function from A^* to A^* defined in terms of r by $z(x) = r(|x|)$. Let us now define a representation function r to be *proper* iff it is injective and the corresponding function z over strings is computable.

It is not hard to see that this requirement does the job. Suppose we have a machine TM_z that calculates z. Here is how we can use it to figure out how to represent a number according to r. For any number n, write down n characters on the input tape of TM_z. (It doesn't matter which characters from A we use.) Then run TM_z. The output will be the required representation of n, which can be then be used as the input to a TM.

With this idea, we can now say what it means to compute with numbers. A function F over numbers is defined to be *computable* iff there is a proper representation function r and a TM m such that such that for any number n, m halts on input $r(n)$ with output $F(n)$. (If the output is supposed to represent a number, then some additional talk is needed.)

So there you have it: computation over numbers, not just over strings. And is the halting problem then undecidable, as expected? It is, since for no *proper* representation function is there a TM that does the job.

The observant reader will notice that all I have really done is moved from using binary numerals as the gold standard, to what amounts to a version of unary numerals. Instead of requiring r to be computable when its argument is written in binary, I am requiring it to be computable when its argument is written in a form of unary notation using A. So why not stick with the better known binary version?

The reason is this: The use of numbers I am proposing is really more basic, more in keeping with what Turing tried to do when he proposed the operations for TMs. It goes back to the idea of counting. Before there were numerals, there was the more elementary idea of using a representative for each thing being counted. You have a bag of stones with as many stones in it as there are sheep to be counted. Or you have a piece of paper with that many marks on it. Or you have an input tape with that many characters on it. Historically, this notion predates the idea of numerals. You don't need to talk about an ordered sequence of characters if they are all treated the same, like a bag of stones. The idea of numerals, as strings of distinguishable characters, is a more advanced notion which came much later, and led eventually to arithmetic, computation, and all the rest of it. It is only in retrospect, once we have seen the notion of numerals, that we might go back and recast counting as something involving unary numerals.

As a final comment, note that we would not want to skip the whole idea of a representation function r and simply insist that a TM for numbers must take as its input a number written in unary notation. There are good reasons for wanting things like binary numerals as input, the main one being that they have lengths that are logarithmic in the number being represented. If we insisted on computing starting with unary numerals as input, then just reading the number would already take time that is exponential in the length of the corresponding binary numeral, and many distinctions in efficiency we care about would be lost.