Notes on the Limits of Computation

Leo Reyzin

The Halting Problem

Given a model of computation (e.g., Python or Java or Turing Machines or Intel CPUs), we can ask if anything we may want to compute is "computable."

Since a program is just a sequence of bits, we can ask whether there exists a program to answer a certain question. If it does, we will call such a question "computable."

What do we mean by saying that some question is computable? We mean "is there a program that computes the answer given the input data." For instance, for addition, we know that there is a program that computes $x + y$ given x and y. So sum of two numbers is computable.

Sad Fact: There are things you care about but can't compute.

We will now show examples of things we can't compute. Suppose we want to make sure some program (call it P) is "doing its job" correctly. Can we write another program (call it H) that tests the program P and tells us whether P works correctly or not? That would seem to be a very useful thing to have: H could tell you whether P is a virus, or whether P will crash your computer, etc. Note that, because programs are bits (just like any other data), giving a program P as input to a program H is not a problem.

We will think simpler, however. A question such as "Is P doing its job correctly?" is hard to formalize. Instead, let's ask a more basic question: "Will P ever stop?" Note that stopping, for programs, is generally a good thing: you want your web browser to eventually compute the proper layout, display the web page, and stop, waiting for whatever you want it to do next. Similarly, you want your antivirus to eventually figure out whether your latest download contains a virus, report the results to you, and stop (again, waiting for its next task, perhaps).

Moreover, many interesting questions can be converted to the question of whether a given program P will stop. For example, Goldbach's conjecture says that for every even integer greater than 2 there exist two primes that add up to that integer. We don't know whether it is true. We can easily write a program P that goes over all even integers starting at 4 and, for each such integer, tests if it's a sum of two primes. If it ever finds an even integer greater than 2 that is not a sum of two primes, it will stop. Thus, the question of whether this program ever stops is equivalent to the question of whether Goldbach's conjecture is false.

More generally, we can write a program that tests every possible proof of a given theorem and stops if it ever finds a correct proof (remember that correctness of logic proofs can be verified by programs). The question of whether this program ever stops is equivalent to the question of whether the theorem has a proof.

The above discussion is meant to convince that the question of whether a given program will stop is an important question.

A Warm-Up Example

First, as a warm-up example, consider the following program—call it "HOTPO"—for P. This program takes a single input: a positive integer x.

- If x is even, replace it with $x/2$.
- Else (that is, if x is odd), replace with $3x + 1$.
- Stop if $x = 1$. Otherwise, go back to the beginning.

"HOTPO" stands for "half or triple plus one." Although people have studied this procedure for quite some time (since at least 1937), as of this writing (2018) there is no method to tell, given x, whether it will stop or not[1]. In other words, no one has been able to find a program H that given x as input will tell whether $HOTPO$ will stop on x.

Thus, whether HOTPO stops or not on a given x is an example of a problem that we currently cannot compute. But that does not mean that the problem is necessarily not computable: that is, it does not mean that H does not exist. Maybe such an H does not exist, but maybe we just haven't found it yet.

The Halting Problem is Not Computable

Surprisingly, there is a problem that is *provably* not computable. Namely, instead of asking whether HOTPO stops or not, let's ask whether an arbitrary program P stops or not. This problem is known as the *halting* problem (we will use the terms *halt* and *stop* interchangeably). In other words, the task of H is the following: given a program P and an input x, determine whether P will stop when running on x. Note that it is a harder task that just determining whether HOTPO stops on x: now, our single H must work for any P that we give it, not just HOTPO. We will now prove that such an H cannot possibly exist.

Theorem. *The halting problem is not computable. That is, there is no program H that for any pair of inputs P and x will always correctly tell if the program P will stop if it starts running with input x.*

This theorem was proved in 1936 by Alan Turing, with important work leading up to it by Emil Post and Alonzo Church.

Proof. Suppose, for purposes a contradiction, such a program H exists. i.e. there is a program H that takes inputs P and x and outputs yes or no depending on whether P will stop on x (it is important that we are assuming that H is always correct).

To derive a contradiction, we will essentially ask what H will do when given itself as an input program. However, first we will modify H a little to get a very similar program G (for one thing, H expects to start with two inputs P and x rather than a single input), and then from G we will get a contradiction.

We design G as follows: it takes a single input Q and feeds it to both inputs of H (i.e., it sets $P = Q$ and $x = Q$ and runs H on P and x). In addition, at the end of the computation, G does the following:

[1]Collatz conjecture states that it will stop for every positive integer x. So far, no one has been able to prove or disprove it.

- If H outputs "yes," then G loops forever
- If H outputs "no," then stop.

This is a silly program that I probably wouldn't use outside of a proof like this, but I could. The point is that if H existed, then G would also exist, and the existence G can be used to drive a contradiction.

Observe that G stops if and only if the program Q does not stop when run with Q as input.

Now for the ask what happens if we run G with input $Q = G$. In other words, ask if G stops when given G as input. Now let's think about what this means:

- If G stops, then it must be that H (which is part of the program of G) determined that G will not stop when run with G as input. But G just stopped, hence that determination was incorrect, but we assumed that H was always correct. A contradiction.
- Similarly, if G does not stop, then it must be the case that H determined that G will stop when run with G as input. But G does not stop, hence that determination was incorrect. Again, a contradiction.

More formally, let $\text{Halt}(P, x)$ be a predicate that is true if and only if P halts on input x. Then, by definition of H, for all P and x, $H(P, x) = \text{Halt}(P, x)$. And by definition of G, for all Q, $\text{Halt}(G, Q) = \neg H(Q, Q)$. Combining the two equations, we get that for all Q, $\text{Halt}(G, Q) = \neg\text{Halt}(Q, Q)$. And now, plugging in $Q = G$ (formally, this is just universal instantiation of Q with G), we get $\text{Halt}(G, G) = \neg\text{Halt}(G, G)$, which is a contradiction.

Thus, starting from the assumption that H exists and is always correct leads us to a contradiction, which means that H cannot exist! □

A Less CS-ish Analogue

The style of this proof is similar to a paradox that was known well before (commonly called Russell's Barber paradox after the logician Bertrand Russell, though he doesn't seem to have been its first inventor).

A Paradox Imagine a small town where men don't wear beards. In this town, the only person who shaves beards is the barber. And the barber follows a simple rule: the barber shaves all the men (but only those men) who do not shave themselves. The barber himself is a man and needs his beard shaven. But here's a question: who shaves the barber? Two possibilities:

1. The barber. But then he shaves himself, but he only shaves those who don't shave themselves. That's a contradiction.
2. Not the barber. Then he doesn't shave himself. But the barber shaves everyone who doesn't shave himself, so the barber must shave the barber. A contradiction.

Such a barber is like a program G: his very existence leads to a contradiction, once you start thinking about how the barber deals with the barber.

Other Impossible Problems and Consequences

You may wonder why the halting problem is significant. After all, maybe we can compute other useful properties of programs if not whether a program stops. As it turns out, the halting problem leads to many impossibility results in computation and mathematics. Here we list just a few, very informally.

- *Rice's theorem*: For any "nontrivial" question about the behavior of programs (we will not have time to detail what "nontrivial" means), there is no program H that answers it for every program. Thus, there is no universal antivirus, universal detector of crashing programs, etc.

- *Gödel's incompleteness theorem*: there are true mathematical statements for which there is no proof that they are true (even assuming all the reasonable axioms we are used to). This is a particularly unsettling fact, because for a while, well into the 20th century, people believed that every true mathematical statement has a proof—it's just that the proof has to be found.

  Indeed, although the original proof of this fact, by Gödel, was quite complicated, it becomes relatively easy now that we know the answer to the halting problem. Suppose, for purposes of contradiction, that every true statement had a proof. In such a case, for every program P and input x, there would be a proof that $P(x)$ halts or a proof that $P(x)$ doesn't halt (because at least one of these two statements must be true). Of course, proofs can be written down as strings. Moreover, veirfying proofs is something computers can do, if the proofs have sufficient detail. Therefore, consider the following program H: it takes P and x as input, and then goes through all possible binary strings s in order of length, and verifies if s is a proof for "$P(x)$ halts" or a proof for "$P(x)$ does not halt." Such an H would solve the halting problem if a proof for every true statement existed. But we know that there is no H that can solve the halting problem. Therefore, we arrived at a contradiction, and thus there must be true statements whose proofs do not exist.

- *Kolmogorov's theorem*: It is impossible to know, in general, the shortest length a given seuence of bits can be compressed to. We have lots of compression algorithms that do things like replacing one million zeros "000…0" with the command "one million zeros," which is much shorter. It would be nice to know how good our compression can get, but we can't, in general.

- *Hilbert's 10th problem*: Hilbert's 10th problem (posed in 1900 by David Hilbert among 23 problem he thought would significant for 20th century mathematics) asked to find a method of determining, by looking at a polynomial equation with integer coefficients, whether it has positive integer solutions (such equations are known as diophantine equation). For example, the Pythagorean theorem involves the following equation: $a^2 + b^2 = c^2$. We know it has integer solutions, called "Pythagorean triples" (such as $a = 3, b = 4, c = 5$). On the other hand, the similar-looking problem $a^3 + b^3 = c^3$ has no integer solutions other than $a = b = c = 0$. Note that Hilbert posed his problem before a formal model of computation existed, and before any problems were shown to be

uncomputable. In fact, it took until 1970—long after the Halting problem was shown to be impossble—with mathematicians (namely, Martin Davis, Yuri Matiyasevich, Hilary Putnam and Julia Robinson) building on the Halting Problem, to show that there is no such method.

- *Finding the lowest airline fare.* Carl de Marcken looked at writing a program to find low airfares sometime around 2001 and realized that the set of rules airlines are allowed to put on fares was complex enough that it is possible to encode any diophantine equation using the rules. A possible routing that satisfies all the rules will exist if and only if the equation is solvable. Thus, it is impossible to write a program that finds if a valid routing between two cities exists for every conceivable database of fares. It so happens that real-life databases of fares are simpler, and so the problem is solvable for those databases—but any program that solves it cannot (provably!) work for an arbitrary database of fares.

One may ask whether so many uncomputable things are just an artifact of our model of computation. Maybe it's all the fault of bits—perhaps if we computed using mechanical gears, or fingers and toes, or DNA and proteins, we'd get different results. So far, no method of computation has been found that is not equivalent to our model. The idea (established empirically—not by proof, but by observing the real world) that all computation is equivalent to whatever you can compute using bits and gates is known as the Church-Turing thesis. If the thesis is true (and most people think it is), then uncomputable thing are really uncomputable, not matter what kind of computer you devise.