The Questions That Computers Can Never Answer

AATISH BHATIA SCIENCE, 02/05/2014, HTTP://WWW.WIRED.COM/2014/02/HALTING-PROBLEM/

COMPUTERS CAN DRIVE cars, land a rover on Mars, and beat humans at *Jeopardy*. But do you ever wonder if there's anything that a computer can never do? Computers are, of course, limited by their hardware. My smartphone can't double as an electric razor (yet). But that's a physical limitation, one that we could overcome if we really wanted to. So let me be a little more precise in what I mean. What I'm asking is, **are there any questions that a computer can never answer?**

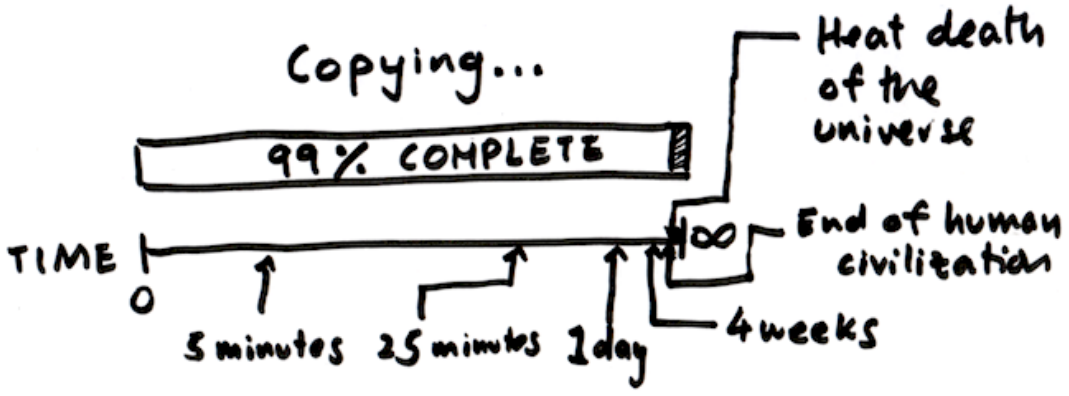
Now of course, there are plenty of questions that are *really hard* for computers to answer. Here's an example. In school, we learn how to factor numbers. So, for example, 30 = 2 × 3 × 5, or 42 = 2 × 3 × 7. School kids learn to factor numbers by following a straightforward, algorithmic procedure. Yet, up until 2007, there was a $100,000 bounty on factoring this number:

> 13506641086599522334960321627880596993888147560566702752448514385152651060
> 48595338339402871505719094417982072821644715513736804197039641917430464965
> 89274256239341020864383202110372958725762358509643110564073501508187510676
> 59462920556368552947521350085287941637732853390610975054433499981115005697
> 7236890927563

And as of 2014, no one has publicly claimed the solution to this puzzle. It's not that we don't know *how* to solve it, it's just that it would take way too long. Our computers are too slow. (In fact, the encryption that makes the internet possible relies on these huge numbers being impossibly difficult to factor.)

So lets rephrase our question so that it isn't limited by current technology. **Are there any questions that, no matter how powerful your computer, and no matter how long you waited, your computer would never be able to answer?**

Surprisingly, the answer is yes. The Halting Problem asks whether a computer program will stop after some time, or whether it will keep running forever. This is a very practical concern, because an infinite loop is a common type of bug that can subtly creep in to one's code. In 1936, the brilliant mathematician and codebreaker Alan Turing proved that it's *impossible* for a computer to inspect any code that you give it, and correctly tell you



whether the code will halt or run forever. In other words, Turing showed that a computer can never solve the Halting Problem.

You've probably experienced this situation: you're copying some files, and the progress bar gets stuck (typically at 99%). At what point do you give up on waiting for it to move? How would you know whether it's going to stay stuck forever, or whether, in a few hundred years, it'll eventually copy your file? To use an analogy by Scott Aaronson, "*If you bet a friend that your watch will never stop ticking, when could you declare victory?*"

As you get sick of waiting for the copy bar to move, you begin to wonder, wouldn't it be great if someone wrote a debugging program that could weed out all annoying bugs like this? Whoever wrote that program could sell it to Microsoft for a ton of money. But before you get to work on writing it yourself, you should heed Turing's advice – a computer can never reliably inspect someone's code and tell you whether it will halt or run forever.

Think about how bold a claim this is. Turing isn't talking about what we can do today, instead he's raised a fundamental limitation on what computers can *possibly* do. Be it now, or in the year 2450, there isn't, and never will be, any computer program that can solve the Halting Problem.

In his proof, Turing first had to mathematically define what we mean by a computer and a program. With this groundwork covered, he could deliver the final blow using the time honored tactic of proof by contradiction. As a warm up to understanding Turing's proof, let's think about a toy problem called the Liar paradox. Imagine someone tells you, "this sentence is false." If that sentence is true, then going by what they said, it must also be false. Similarly, if the sentence is false, then it accurately describes itself, so it must also be true. But it can't be both true and false – so we have a contradiction. This idea of using self-reference to create a contradiction is at the heart of Turing's proof.

Here's how computer scientist Scott Aaronson introduces it:

> *[Turing's] proof is a beautiful example of self-reference. It formalizes an old argument about why you can never have perfect introspection: because if you could, then you could determine what you were going to do ten seconds from now, and then do something else. Turing imagined that there was a special machine that could solve the Halting Problem. Then he showed how we could have this machine analyze itself, in such a way that it has to halt if it runs forever, and run forever if it halts. Like a hound that finally catches its tail and devours itself, the mythical machine vanishes in a fury of contradiction.*

Like the serpent that tries to eat its tail, Turing conjured up a self-referential paradox. The program will halt when the loop snooper said it wouldn't, and it runs forever when the loop snooper said it would halt! To resolve this contradiction, we're forced to conclude that this loop snooping program can't exist.

And this idea has far-reaching consequences. There are uncountably many questions for which computers can't reliably give you the right answer. Many of these impossible questions are really just the loop snooper in disguise. Among the things that a computer can never do perfectly is identifying whether a program is a virus, or whether it contains vulnerable code that can be exploited. So much for our hopes of having the perfect anti-virus software or unbreakable software. It's also impossible for a computer to always tell you whether two different programs do the same thing, an unfortunate fact for the poor souls who have to grade computer science homework.

By slaying the mythical loop snooper, Turing taught us that there are fundamental limits to what computers can do. We all have our limits, and in a way it's comforting to know that the artificial brains that we create will always have theirs too.