These are notes on Gödel's Theorem and Turing's proof of the undecidability of the halting problem, taken from a longer naration.

1. GÖDEL'S PROOF

Subject: Splean

From: bald@math.unitrieste.it

Date: July 19, 1999. 2:15 (GST)

It is a hot summer night in the Adriatic, way past midnight, my theorem is going nowhere, too much caffeine in my system, and still no word from you. A perfect time, I think, for filling in a couple of the many important points and details that my lazy colleague `Turing` left out of his lecture on Gödel's theorem.

To understand this momentous result, you must first understand the man's premises. Gödel considered mathematical statements such as "$1+1=2$" and "$1+1=3$" —some of which are true –we call them theorems— and some are, evidently, false. But the more interesting statements involve variables, for example, "$x+1=y$", they are true or false depending on the values of the variables.

Certain statements involving variables happen to be true for all values, they are algebraic identities, like "$x+1=1+x$", "$x \neq x+1$", and "$(x+1) \uparrow 2 = x \uparrow 22 \cdot c + 1$" (by $\uparrow$ I denote exponentiation). That such a statement is true for all values of x can be considered itself a higher-order statement; for the last example, this higher-order statement is written "$\forall x \quad (x+1) \uparrow 2 = x \uparrow 22 \cdot x + 1$", where the $\forall x$ part is read "for all x", and means exactly that: that the rest of the statement holds for any choice of integer value for x. Such a statement is either true or false, just like the simple statements "$1+1=3$" that involve no variables. For example, $\forall x \quad x = x \uparrow 2$ is false, as the part after $\forall x$ holds for a couple but not all values of x.

(We can now answer Suleiman's question: Indeed, the truth of statements in Euclidean geometry along the lines of the ones we examined above, using the same kinds of generalistic quantifiers "for all lines" and "for all points," can be decided by an algorithm. But the tiling problem explained by Ian belongs to a higher sort of logic, with quantifiers that say "for all tilings," or equivalently "for all sets of points and lines." There is no algorithm for deciding such sentences.)

So, the truth of some statements depends on the value of a variable, while others claim (perhaps falsely) that the statement is true for all values. Gödel was interested in a curious kind of hybrid statement, such as $\forall x \quad y \neq x + x$. The truth of this statement depends on the value of y —this particular one is true if y is odd, false otherwise. That is, if $A(x,y)$ is any arithmetic statement involving variables x and y (and possibly other variables inside A, each of them, call it z, neutralized by the corresponding $\forall z$), then Gödel would construct from it the hybrid statement

$$H(y) = \forall x \quad A(x,y).$$

Notice the y in parentheses, it codifies the fact that the truth of H depends on the value of y.

A key ingredient in Gödel's idea was that all statements can be really thought of as numbers — numbers that encode arithmetic statements are known as Gödel numbers. This is no big deal, just think that we have an infinite list of all possible legal statements, shortest first, and each sentence is identified by its rank in this list. Gödel was interested in such an enumeration of his hybrid statements,

$$H_0(y) = \forall x \ \ A_0(x,y), \quad H_1(y) = \forall x \ \ A_1(x,y), \quad H_2(y) = \forall x \ \ A_2(x,y), \quad \ldots$$

where the $A_i(x,y)$'s are all possible arithmetic statements involving x and y.

Recall now Gödel's purpose in all this. He wanted to show that, in any mathematical system that expresses the integers and operations on them, not all theorems can have proofs. But what is a proof? The important point here is that a proof is necessarily a string of characters. Therefore, lo and behold, proofs in the assumed system (the mathematical system that is to be proved incomplete) can also be enumerated:

$$P_0, P_1, P_2, \ldots$$

So, we have an enumeration of all hybrid statements, and an enumeration of all proofs. That much is easy. Now comes the crucial idea. If you substitute a number for the variable y of a hybrid statement, then you get something that is either a theorem (and may have a proof) or a negation of a theorem (in which case it does not). If I give you, say, $H_{87}(87)$ —notice the beginning of a diagonalization: This is the 87th hybrid statement with the value of variable y set to the statement's own rank in the enumeration, 87— and a proof, say P_{290}, then it should be easy to tell if P_{290} happens indeed to be a proof of $H_{87}(87)$. But Gödel's genius was in refining this: Not only can you tell this for any hybrid statement with its variable set to its rank, $H_n(n)$, and any proof P_m, but there is an *arithmetic statement* involving the variables m and n that achieves this. In other words, you can express the fact that P_m is (or, is not) a proof of $H_n(n)$ as a long list of equations and inequalities involving m, n, multiplied and added and exponentiated together in complicated ways (and possibly other variables eclipsed by $\forall$'s). It can be done, I am omitting lots of details here, it is very complicated and rather tedious in the end —the truly clever part is not to achieve it, but to reale that it is useful.

Thus, there is an arithmetic statement $A(x,y)$ whose truth depends on the variables x and y, and it is true only if it so happens that P_x is *not* a proof of the statement $H_y(y)$. But this arithmetic statement $A(x,y)$ gives rise to a hybrid statement $H(y) = \forall x \ \ A(x,y)$ —stating, if you think about it for a minute, that $H_y(y)$ has no proof. But this hybrid statement must be somewhere in our enumeration of the hybrid statements, perhaps it is the k-th such statement, $H_k(y)$, where k is some fixed number.

The diagonalization is now complete: Statement $H_k(k)$, since it is obtained from $H_k(y)$ (whose truth only depends on y) by substituting the number k for y, is unambiguously either true or false. And, as we argued above, statement $H_k(k)$ states that statement $H_k(k)$ has no proof. $H_k(k)$ is precisely the statement called K by `Turing` —the statement that declares itself unprovable. As Turing argues then very convincingly, K must be true but unprovable, thus establishing Gödel's incompleteness theorem.

I hope this helps. If not, there is a nice book that contains a more detailed explanation: *Gödel's Proof* by Ernest Nagel. Another delightful book recounts the development of the field of logic, leading to the results by Gödel and Turing: *The Universal Computer: The Road from Leibniz to Turing* by Martin Davis. *Computers Humbled* by David Harel covers computability but also complexity and NP-completeness (the subjects of a subsequent related lecture by `Turing`). But the mathematically sophisticated reader has more choices: *Introduction to the Theory of Computation* by Michael Sipser, *Computational Complexity* by Christos Papadimitriou, and (in relation to complexity) *Computers and Intractability: A Guide to the Theory of NP-completeness* by Michael Garey and David Johnson. Finally, it is worth noting that the P vs. NP problem explained by `Turing` in the chapter *Complexity* is first in the list of the "Millennial" prize mathematical problems of the Clay Institute:

`http://www.claymath.org/prize_problems/index.htm`

Dott. Baldovino Croce, Trieste

COMPUTABLITY Same here, my friend, same here.

OK then, computability. You see, it's like this: Before the computer, we thought we were invincible. *"There are no unsolvable problems"* —that's Hilbert again. If the problem is hard, you use more sophisticated techniques, develop better math, work longer hours, talk to a smarter colleague, perhaps wait for the next generation. But it *will* be solved. Now we know better. But then — that's what people thought. And there was a reason for this optimism: We had not seen really hard problems. There were there, for sure, but we had no eyes for them. And then the computer came. A beast constructed expressly for further facilitating and speeding up our inescapable conquest of all problems. And —surprise!— it was itself the epitomy of complexity. And the code that we had to write to make it useful, the code was even more complex. And when we saw the computer, when we saw its code —and Turing saw it first— we were looking at complexity incarnate. And then suddenly we saw complexity everywhere. It materialized, it crystalized around us —even though it had always been there. We have yet to recover from the shock.

Take code. It's everywhere, in our computers, on the Net. It's in the little disks you get in junk mail, in the back covers of books, in little applets you download from the Net sites you visit. It's easy to forget, somebody must write this code. When you work with code, you see many times more code than you write. Code written by others, often years ago, often by people you will never meet, you will have no chance to chat with them over coffee, a printout spread in front of you. You are an archeologist, Alexandros, you must know the feeling: What the hell was *this* for? What were those people *thinking?*

Picture this. A mysterious, chaotic sequence of statements is spread in front of you. Is it good code or bad code? Slow code or fast code? Does it have bugs? Is it a virus, will it take over your files, sniff your password, deplete your bank account? Will it ever send a mail message from your account? Will it crash on January 1? Will it ever print out something? And if so, will it ever stop printing? Is it correct code, will it do what it is supposed to do —process orders, for example, update sales figures, and print mailing labels? Does it have redundant parts, pieces of code that will never be executed, can be erased with impunity? You spend your day trying to figure these things out. You can run tests, of course, but for how long? How many experiments will you run, how many test inputs will you try?

When you work with code, these are your bread-and-butter problems. And here is my point: *They are all unsolvable.* There is no systematic way for answering them. You have to be constantly on your toes, one IQ point smarter than the code on your screen. There is no silver bullet. I can prove it for you.

Let's take perhaps the simplest problem of all: *Will this code ever stop?* The halting problem. It can't be solved. Suppose I give you a piece of code, Alexandros, a couple of hundred lines long. How would you figure out if it ever stops? I even give you its input. Will it stop? You will probably eye the code for a few minutes. If it has no `return` instruction, no `stop` instruction, then it's a dead giveaway, it will never stop. But suppose that it has a few, buried among the others, the if-then-else's, the repeat-until's, then what? Will the execution ever reach those points? How do you ever figure this out? You will probably run the code with the given input, to see if it will eventually stop. If it does stop, you are home free —you have your answer. But if not, how long will you wait? "Maybe if I wait a little longer, just a little, it will stop." How many times should you indulge? How do you decide before forever? How do you systematically decide if a given code will ever stop, when started with a given input?

Well, you can't. And here is proof: Suppose you could. Suppose you have written your silver bullet, the almighty code halts(code, input) which, given some code with its input, it computes away for a while, and then announces its conclusion: "yes" means that the code will eventually halt on the input, "no" that it won't. So, just suppose that you have that. You are now in the mercy of Cantor and his evil diagonals:

```
algorithm turing(code)
if halts(code, code) then
  repeat { x ← 1 } forever
else stop
```

"Wow, this is the most." Aloé is in love —at the same time, she can't wait to tell Timothy.

See? This code does something very simple: For any given piece of code, it asks: "Will this code eventually stop if supplied *with itself* as an input?" If so, then turing(code) happily jumps into an infinite loop. Otherwise, it rushes to stop.

And now comes the unanswerable question, the absurd situation that will expose the absurdity of halts(code, input): "What will the program turing do when given *itself* as an input?" Does turing(turing) stop eventually, or does it compute forever? Can you figure it out, Alexandros?

"I think I got it," Alexandros is beaming. "If turing(turing) ever stops, then the line halts(turing, turing) will return "yes," and so turing(turing) will never stop, it will get into the repeat-forever loop." Pause. "But if turing(turing) does not stop, then halts(turing, turing) will return "no," and it will stop immediately. So, it stops if and only if it does not."

Exactly. And this is a contradiction, of course. Code either stops or doesn't. So, we must have erred in assuming that the halts(code, input) program exists —this was the only slippery part in this construction, everything else is clean solid coding. So: There can be no code that solves the halting problem. *The halting problem is unsolvable.*

But so are all the other questions about code that I mentioned. Take for example the question "Will this code ever print anything?" Well, suppose that the only *print* statements of your program are just before your *stop* statements. Then it will print something if and only if it will stop. So, the "printing problem" is as unsolvable as the halting problem. And so on, and so on, for all of them. You can't analyze code systematically. Code is hard, its secrets are unfathomable. Code analysis can only be done by tedious, thankless toil, by discovering *ad hoc* tricks that will work for this program but will be worthless on the next.

OK, starting from the halting problem you can argue that almost any question you can ask about code is unsolvable. But there are unsolvable problems everywhere in science and math. Even in geometry.

So. Computational problems —problems for which you would have liked to write code— are subdivided into two big categories: Those problems that are solvable by algorithms; and those that are unsolvable. We have known this for a long time, we have learned to live with unsolvability. The unsolvable problems seeped in our culture, we instinctively steer clear of them. Trouble is, there are too many other problems that fall somewhere in between. They are solvable all right, but the only code we have for them runs for way too long. Exponentially long. For such problems the diagnosis has to be more subtle. Practically unsolvable. **NP**-*complete.* But this is a whole new story.