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.