6.080 / 6.089 Great Ideas in Theoretical Computer Science
Spring 2008

6.080/6.089 GITCS	Feb 14, 2008
Lecture 4	
Lecturer: Scott Aaronson	*Scribe: Aseem Kishore*

1 Previously in 6.089...

Last lecture, we talked about two different models of computation, *finite automata* and *circuits*. Finite automata allowed us to recognize many properties of an arbitrarily long input, while circuits allowed us to represent any Boolean function.

However, both models had significant limitations. Circuits were limited in hardware–we have to know how big an input is before we can make a circuit for it–while finite automata were limited by their memory–which also had to be known in advance of a problem.

2 Turing Machines

How can we generalize finite automata to overcome their limitations? A first idea is to let them move backwards on the tape as well as forwards. This is a good start, but by itself it actually provides no extra power. To see why, suppose a two-way finite automaton is in some state a, going forward at some point x on the tape. At this point, if it goes backwards on the tape and then returns to x, that simply induces some function $a \Rightarrow f(a)$, where f depends on the earlier part of the tape. But this means that we can simulate the two-way machine by a one-way machine, which simply keeps track of the whole function f instead of just the state a.[1] Thus, being able to move in two directions provides no additional power on its own.

What we really need is a machine that can not only move backwards and forwards, but also *write* to the tape and *halt* at any time of its choosing. And that's what a Turing machine is. The ability to write essentially gives Turing machines an unlimited memory, since any information that can't fit in the machine's internal state can always be written to the tape. The ability to halt at discretion means that Turing machines aren't "tied to the input" the way finite automata are, but can do as much auxiliary computation as they need.

So at any given point on the tape, a Turing machine faces three questions:

1. Change state?
2. Write to the tape?
3. Move left, move right, or halt?

The machine's answer to these questions is a function of its current state and the current input on the tape.

A clear example of these features overcoming the limitations of finite automata is a Turing machine's ability to solve the palindrome problem. By using a simple back-and-forth process, a

[1] Note that the number of functions $f(a)$ mapping states to states grows exponentially with the number of states in the original machine.

Turing machine can repeatedly check that a letter at one end exists at the opposite end, and by marking letters that it has seen, the machine ensures it continuously narrows its scope. (Here we're assuming that additional symbols are available besides just 0 and 1.) This algorithm takes $O(n^2)$ time (interestingly, there is a proof that argues that this is the best a Turing machine can do).

Likewise, addition of integers is also possible, as are multiplying and some other mathematical operations. (We won't prove that!) Searching for non-regular patterns also becomes possible. But perhaps the most interesting thing a Turing machine can do is to emulate another Turing machine!

3 Universal Turing Machines

In his 1936 paper "On Computable Numbers" (in some sense, the founding document of computer science), Turing proved that we can build a Turing machine U that acts as an interpreter for other Turing machines. In other words, U's input tape can contain a description of *another* Turing machine, which is then simulated step by step. Such a machine U is called a *Universal Turing machine*. If a universal machine *didn't* exist, then in general we would need to build new hardware every time we wanted to solve a new problem: there wouldn't even be the concept of *software*. This is why Professor Aaronson refers to Turing's universality result as the "existence of the software industry lemma"!

A question was brought up in class as to how this can be, if the machine being interpreted may require more states than the interpreting Turing machine has. It turns out that universal Turing machines aren't limited by their states, because they can always keep extra state on blank sections of the tape. They can thus emulate a machine with any number of states, but themselves requiring only a few states. (In fact, there is a popular parlor-type competition to find a universal Turing machine that uses as few states and symbols as possible. Recently, one student actually came up with such a machine that uses only two states and a three-symbol alphabet. To be fair, however, the machine required the inputs in a special format, which required some pre-computation, so a question arises as to how much of the work is being done by the machine versus beforehand by the pre-computation.)

4 The Church-Turing Thesis

Related to the idea of universal machines is the so-called *Church-Turing thesis*, which claims that anything we would naturally regard as "computable" is actually computable by a Turing machine. Intuitively, given any "reasonable" model of computation you like (RAM machines, cellular automata, etc.), you can write compilers and interpreters that translate programs back and forth between that model and the Turing machine model. It's never been completely clear how to interpret this thesis: is it a claim about the laws of physics? about human reasoning powers? about the computers that we actually build? about math or philosophy?

Regardless of its status, the Church-Turing Thesis was such a powerful idea that Gödel declared, "one has for the first time succeeded in giving an absolute definition to an interesting epistemological notion."

But as we'll see, even Turing machines have their limitations.

5 Halting is a problem

Suppose we have a Turing machine that never halts. Can we make a Turing machine that can detect this? In other words, can we make an infinite loop detector? This is called the *Halting problem*.

The benefits of such a machine would be widespread. For example, we could then prove or disprove Goldbach's Conjecture, which says that all even numbers 4 or greater are the sum of two primes. We could do this by writing a machine that iterated over all even numbers to test this conjecture:

```
for i = 2 to infinity:
  if 2*i is not a sum of two primes
    then HALT
```

We would then simply plug this program into our infinite-loop-detecting Turing machine. If the machine detected a halt, we'd know the program must eventually encounter a number for which Goldbach's conjecture is false. But if it detected no halt, then we'd know the conjecture was true.

It turns out that such an infinite loop detector can't exist. This was also proved in Turing's paper, by an amazingly simple proof that's now part of the intellectual heritage of computer science.[2]:

We argue by contradiction. Let P be a Turing machine that solves the halting problem. In other words, given an input machine M, $P(M)$ accepts if $M(0)$ halts, and rejects if $M(0)$ instead runs forever. Here $P(M)$ means P run with an encoding of M on its input tape, and $M(0)$ means M run with all 0's on its input tape. Then we can easily modify P to produce a new Turing machine Q, such that $Q(M)$ runs forever if $M(M)$ halts, or halts if $M(M)$ runs forever.

Then the question becomes: what happens with $Q(Q)$? If $Q(Q)$ halts, then $Q(Q)$ runs forever, and if $Q(Q)$ runs forever, then $Q(Q)$ halts. The only possible conclusion is that the machine P can't have existed in the first place.

In other words, we've shown that the halting problem is *undecidable*–that is, whether another machine halts or not is not something that is *computable* by Turing machine. We can also prove general uncomputability in other ways. Before we do so, we need to lay some groundwork.

6 There are multiple infinities

In the 1880's, Georg Cantor discovered the extraordinary fact that there are different degrees of infinity. In particular, the infinity of real numbers is greater than the infinity of integers.

For simplicity, let's only talk about positive integers, and real numbers in the interval $[0, 1]$. We can associate every such real number with an infinite binary string: for example, 0.0011101001.... A technicality is that some real numbers can be represented in two ways: for example, $0.100\overline{0}$ is equivalent to $0.011\overline{1}$. But we can easily handle this, for example by disallowing an infinity of trailing 1's.

To prove that there are more real numbers than integers, we'll argue by contradiction: suppose the two infinities are the same. If this is true, then we must be able to create a one-to-one association, pairing off every positive integer with a real number $x \in [0, 1]$. We can arrange this association like so:

[2]This proof also exists as a poem by Geoffrey K. Pullum entitled "Scooping the Loop Snooper": http://www.ncc.up.pt/∼rvr/MC02/halting.pdf

```
1: 0.0000... (rational)
2: 0.1000...
3: 0.0100...
4: 0.101001000100001... (irrational)
5: 0.110010110001001...
```

We can imagine doing this for all positive integers. However, we note that we can construct another real number whose n^{th} digit is the opposite of the n^{th} digit of the n^{th} number. For example, using the above association, we would get 0.11110...

This means that, contrary to assumption, there were additional real numbers in $[0, 1]$ not in our original list. Since every mapping will leave real numbers left over, we conclude that there are more real numbers than integers.

If we try to apply the same proof with rational numbers instead of real numbers, we fail. This is because the rational numbers are *countable*; that is, each rational number can be represented by a finite-length string, so we actually can create a one-to-one association of integers to rational numbers.

7 Infinitely many unsolvable problems

We can use the existence of these multiple infinities to prove that there are uncomputable problems. We'll begin by showing that the number of possible Turing machines is the smallest infinity, the infinity of integers.

We can define a Turing machine as a set of states and a set of transitions from each state to another state (where the transitions are based on the symbol being read). A crucial aspect of this definition is that both sets are finite.

Because of this, the number of Turing machines is *countable*. That is, we can "flatten" each machine into one finite-length string that describes it, and we can place these strings into a one-to-one association with integers, just as we can with rational numbers.

The number of *problems*, on the other hand, is a greater infinity: namely, the infinity of real numbers. This is because we can define a problem as a function that maps every input $x \in 0, 1^*$ to an output (0 or 1). But since there are infinitely many inputs, to specify such a function requires an infinite number of bits. So just like with Cantor's proof, we can show that the infinity of problems is greater than the infinity of Turing machines.

The upshot is that there are far more problems than there are Turing machines to solve them. From this perspective, the set of computable problems is just a tiny island in a huge sea of unsolvability. Admittedly, most of the unsolvable problems are not things that human beings will ever care about, or even be able to define. On the other hand, Turing's proof of the unsolvability of the halting problem shows that at least *some* problems we care about are unsolvable.