

CMSC 31100 Week 1: Computability

Lecturer: Janos Simon

October 1, 2006

What can we compute?

In the 30's Alan Turing introduced the concept of a *Turing Machine*:

- Infinite tape
- Finite alphabet
- Finite number of states
- Instructions are quintuples: given the current state and the character in the current cell in the tape, specify the new character in the current cell, the new state, and whether the head moves left, right, or stays in place.

The Church-Turing thesis: Something is computable if and only if it is computable by a Turing Machine.

This is a proposed definition of *computable*. There are several statements we can prove using this definition, which lead us to believe this is a good definition.

- Everything that a *computer* (i.e. a person who does not use intuition, but only mechanical instructions) can compute is computable.
- There exists a *Universal Turing Machine* U , that is \exists TM U s.t. \forall TM M , and \forall inputs x to M , $U(M,x)$ simulates $M(x)$, that is, has the same output.
- There are problems that are non-computable, for example *the halting problem*.

Non-computable problems

The halting problem: given a TM M , and an input x , does $M(x)$ halt?

Claim: The halting problem is undecidable.

Proof: Suppose, by contradiction, that there exists a TM that can solve the halting problem. Then we could construct a TM D such that $D(M) = 1$ if $M(M)$ does not halt, and $D(M)$ loops infinitely if $M(M)$ halts. Now look at $D(D)$. If $D(D)$ halts, then $D(D) = 1$, but then $D(D)$ does not halt, which is a contradiction. So $D(D)$ does not halt. But then $D(D) = 1$ and halts, which is a contradiction.

Variations of the halting problem: Once we have one undecidable problem, we can show many problems are undecidable, by showing that if we could solve them, we could solve the halting problem.

- Given a TM M , does there exist an input x such that M halts on x ?

Proof of undecidability: Suppose, by contradiction, there exists a TM S such that $S(M) = 1$ if $\exists x$ s.t. $M(x)$ halts, and $S(M) = 0$ otherwise. Then, given an instance (N, y) of the halting problem, build a TM M , which ignores its input, and simulates $N(y)$. M halts on some input iff M halts on every input iff $N(y)$ halts. But we can't solve the halting problem, so we have a contradiction.

- Given a TM M , does M halt on every input? Same proof.
- Given two TM M_1 , and M_2 , do they halt on the same set of inputs?

Proof of undecidability: Given an instance (N, y) of the halting problem, build machine M_1 like in the previous proof, and M_2 that loops forever. Then M_1 and M_2 halt on the same set of inputs iff $N(y)$ does not halt.

- Given a TM M , does $M(x)$ halt in time $\leq 23|x|, \forall x$?

Proof of undecidability: Build a *clock* that counts until $|x|$. Simulate $N(y)$ until the clock runs down. If $N(y)$ halts before the clock expires, then loop forever. Details are omitted.

All the undecidability proofs we have discussed use the undecidability of the halting problem. So what if we assumed we had an oracle for the halting problem, would there be undecidable problems then? The answer is still yes, the proof is extra credit homework.

Criticisms to the Church-Turing thesis

Too weak

Some criticize the Church-Turing thesis by saying that the model of computation is too weak. Here are some alternative stronger notions.

- **Computably enumerable sets:**

A language L is said to be *computably enumerable* (CE) if \exists TM N s.t. $N(x) = 1$ if $x \in L$, and $NOT(N(x) = 1)$ otherwise. For strings not in L $N(y)$ may output a 0, or may loop forever—it can be anything, except it cannot be 1.

Here are some simple properties of CE sets:

- If L is a CE set, and \exists a computable time bound for the computation of x for every $x \in L$, then L is computable.
- If L is CE and \bar{L} is CE, then L is computable.
- An easy consequence of the last statement is that the complement of the halting problem is not CE.

- **Communicating Sequential Processes** How can a TM represent the computation done by an operating system? A TM computes a function, there is an input and an output, we don't really care what happens in between. An operating system is different: it doesn't have an input or an output, it is supposed to run forever, and respond to various signals.

- *Communicating Sequential Processes* is the name given to a model of computation that is better suited for modeling an operating system.
- In this model, there are many automata that run for ever, and interact with each other (by sending messages).
- A good computation in this model will have properties like:
 - * all automata who want to run will eventually run,
 - * something will happen infinitely often,
 - * at some point something will happen.
- These properties are non-computable by a TM.

These (and other) models of computation, that are incomputable in the TM sense, can still be useful. In fact we are often interested in solving specific instances of non-computable problems, however there is a sense in which these problems are *too hard* to compute in general.

Too strong

There is also some criticism towards the Church-Turing thesis saying that it is too strong. In fact many of the problems that are “computable” could never be computed in practice, as they would require way too much time. The **generalized Church-Turing thesis** defines computable as computable by a TM in polynomial time. Without going that far, it can be argued that NP is what is computable, since we can check whether an answer is correct (if we happen to stumble upon a solution). Another possibility would be to define computable to be BPP (Bounded-error Probabilistic Polynomial-time), because in practice we can never compute something with absolute certainty (hardware can fail), so it seems reasonable to tolerate small errors in the computation. In fact, other complexity classes (e.g. quantum polynomial time, or PSPACE) could be used to define what is computable.

The Church-Turing thesis seems to be the most theoretically sound model for computability. This does not mean, however that we’re not interested in weaker or stronger models of computation.