# CS 3100 – Models of Computation – Fall 2011 – Notes for L26
# Computability, Undecidability
# Last Updated 10:01am, 11/29/11

**Cardinalities (recap)**

- We can show that the cardinality of the set of finite fractions between 0 and 1 is $\aleph_0$ (try using the S-B theorem)
- Write another proof for the cardinality of the set of all C programs, using the S-B theorem.
- We also solved for the cardinality of the set of all languages over $\Sigma^*$ where $\Sigma$ is some non-empty alphabet (singleton alphabet is OK). Recap that.
- Mismatches of cardinality were argued using the *diagonalization* method. Make sure you understand that.

**Remaining Portions: the Halting Problem; Mapping Reductions; BDDs**

- We will resort to the diagonalization argument in showing the undecidability of the halting problem
- We have to build up our notions before we can do that
- We will show how to relate problems to each other using mapping reductions (so using the unsolvability of one problem, we can show the unsolvability of other problems).
- These arguments are also used for showing NP-completeness
- To appreciate NP-completeness and the power of Boolean reasoning, we will revisit DFA; we can show how they can compactly encode Boolean functions (in the form of "BDDs")

**Languages of Machines**

- So far we studied languages that have a "real purpose" – e.g. language of all strings with an odd number of 1's
- We also studied languages that legally encode machines – e.g. "is this a regular expression" or "is this a CFG" or "is this a PDA description in some encoding format" or "is this a DFA encoding as per JFLAP's conventions"
- We will simply write

$$\{\langle D \rangle \mid D \textit{ is a } DFA\}$$

  to denote "the language of legal DFA encodings."
- Now we can ask more interesting questions:
  - Is this a DFA we are looking at, and *furthermore*, is it those DFAs that do not accept strings with odd 1s.

$$\{\langle A \rangle \mid A \text{ is a DFA that does not accept any string with odd 1s}\}.$$

    or

$$\{\langle D \rangle \mid D \text{ is a DFA that does not accept any string with odd 1s}\}.$$

    (the letter we use does not matter; it is a variable)
- When we say "we," we don't mean humans. Eventually we want to automate things. So we (humans!) will say "is a TM looking at DFAs, claiming to filter out exactly those that do not accept strings with odd 1s?"

- An algorithm has an input, generates an output, is effective (mechanizable), definite (not non-deterministic), and finite (halts). We are mainly focussed on *finiteness* in these discussions (we assume the rest of all problems discussed here).
- We (humans – clear from the context, now on, I hope) want TMs to be *algorithmic* TMs. I.e. when fired up, the TM must do the job and halt in an accepting state saying "yes, found such a DFA" or "no, this ain't such a DFA." Then only it can separate $\Sigma^*$ into two bins: (1) those strings from $\Sigma^*$ that encode DFAs that do not accept odd 1s, and (2) those strings that either don't encode DFAs or encode DFAs that have at least one odd-length string in its language.
- But we will find that life isn't so smooth; some problems do have algorithms; some problems don't have algorithms; and some have *semi-algorithms*!
- The most important modeling trick:

  Problem solving = Language decision!

  That is, to solve a problem, model the problem using a language, and then try and build a TM that decides membership in the language.
- Building a TM is **really not necessary**; instead, just imagine writing a C or Java program (or Python, or ..)
- Examples:
  - Problem with an algorithm (language that is decidable):

    $$\{\langle M\rangle \mid M \textit{ is a } TM\}$$

    Clearly, one can check that a legal TM has been encoded as per some syntactic conventions.

    Another example:

    $$\{\langle M, w\rangle \mid M \textit{ is a } TM \textit{ and } w \textit{ is a string in the tape alphabet of } M\}$$

    Even this is algorithmically checkable. One agrees on a convention to "lay down" a TM description of $M$ on a tape, then put a separator, then put down $w$.
  - Problem with a semi-algorithm:

    $$A_{TM} = \{\langle M, w\rangle \mid M \textit{ is a } TM \textit{ and } w \textit{ is a string in the tape alphabet of } M \textit{ and } M \textit{ accepts } w\}$$

    This has a semi-algorithm. Simply let a "master" TM (who is doing the language filtering) run $M$'s steps on $w$ (work like an $M$-interpreter. If and when $M$ is found to accept $w$, declare "$\langle M, w\rangle$" is in $A_{TM}$.
  - Problems with *not even a semi-algorithm*:

    $$EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1 \textit{ and } M_2 \textit{ are language equivalent } TMs\}$$

    Can't conclusively prove halting, in an identical manner, of $M_1$ and $M_2$ over all of $\Sigma^*$.
  - Problems with *not even a semi-algorithm*:

    $$NEQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1 \textit{ and } M_2 \textit{ are language non} - \textit{equivalent } TMs\}$$

    Can't conclusively prove halting, in a non-identical manner (if $M_1$ accepts, $M_2$ does not accept), of $M_1$ and $M_2$ over all of $\Sigma^*$.

- Note that TMs $M$ run on $w$ have three outcomes, **only two of which are discernible**:
  - Accept
  - Reject
  - Loop (is an outcome, but we don't know we can't tell that we are doing that!!!)
- Warning: Don't simply run a given $M$ on a string $w$. There may be better algorithms around. The *existence of algorithms* or *existence of semi-algorithms* can often be concluded smartly. Look for cut-offs or other ways to "dovetail and run other computations in parallel."
  Examples:
  - Wrong semi-algorithm: Generate one string $w_1$ from $\Sigma^*$. Deploy $M$ on $w_1$. If and when $M$ is found to reject $w_1$, move on to another string $w_2$, etc. *What is wrong?*[1]

$$NE_{TM} = \{\langle M \rangle \mid M \text{ is a } TM \text{ with a non} - \text{empty language}\}$$

    A semi-algorithm exists. One can keep cranking out strings "along the x axis" and run $M$ on each string so far generated "along the $t$ (or time) axis" for some fixed number of steps (usually 1) before generating another string. So we generate one string, run one more step of $M$ over all strings, then generate one more string, etc. If $L(M)$ is non-empty, we will eventually hit upon some string $w_a$ and have allocated the right number of steps of $M$'s run on $w_a$ to have sent $M$ to its accept state.
- Note that if we have a semi-algorithm for a problem and its complement (language and its complement), then we will have a *full* algorithm.

$$NE_{TM} = \{\langle M \rangle \mid M \text{ is a } TM \text{ with a non} - \text{empty language}\}$$

- Questions on the decidability of $NOODD_{DFA}$ and $A_{CFG}$: find an *algorithm* to describe the membership (or not) of a candidate string in these languages in a *finite* manner – *i.e.* finish the work and halt with a "yes/no." All algorithms in our discussion are described using pseudo-code and/or precise English bulletted steps.
- An enumerator for $NEQ_{CFG}$ guarantees to list each unequal grammar pair in a finite amount of time. Think of the difficulty you are faced with (over 15 mins or so of deep thought about the matter) as to why $EQ_{CFG}$ – language-equivalent grammar pairs – can't be enumerated.
- The existence of a semi-algorithm means "one has proved, or nobody knows whether there is a full algorithm." As usual, we use the tighest classification, never calling 1 a complex number or {} a context-free language.
- The words decidable and undecidable are opposites; undecidable means *not decidable*.
- The word semi-decidable means undecidable, but *partially decidable* (*i.e.* solvable in the true sense, but not the false sense, or vice versa; one has to clarify which sense holds). $EQ_{CFG}$ is not decidable but semi-decidable: this is because one can guarantee halting at least when the CFGs are language *inequivalent*. For this to happen, they must disagree on at least one string.
- Problem: $ALL_{DFA}$**:** Given a DFA, is its language $\Sigma^*$? Is this problem / language decidable?
- How about $A\varepsilon_{CFG}$**:** Given a CFG, does it generate $\varepsilon$?
- How about $INFINITE_{DFA}$: Given a DFA, does it have an infinite language?
- How about

$$EMPTY_{CFG} = \{\langle G \rangle \mid \text{is a CFG and L(G)} = \emptyset\}.$$

[1] $M$ may loop on $w_1$. Looping is not detectable. Looping may be an ever increasing exaggerated zig-zag over TM's tape, with nothing repeating! While we are at it, we **can** detect that a PDA is looping, as well as an LBA is looping - how? Unfortunately, looping in TMs is not detectable. This is our main theorem about the "Halting problem."

- The universality of the language of a CFG is undecidable. A language $L$ is universal if $L = \Sigma^*$.
- The equivalence of two CFGs is undecidable.
- In-equivalence of two CFGs is undecidable.
- Whether a given Turing machine accepts string $w$ is undecidable.
- Whether a given Turing machine halts on string $w$ is undecidable.
- The emptiness of the language of a Turing machine is undecidable.
- Whether a given Turing machine's language is context-free is undecidable.

A language $L$ is TR if it is the language of *some* Turing machine $M$. We write $L_M$ for emphasis. A language $L$ is RE if there exists a Turing machine $M$ that can enumerate the strings in $L$ (say, on an "output tape") such that any member $x \in L$ is guaranteed to appear in a finite amount of time.
Some clarifications:

- Suppose we are required to show that $L$ is TR. We must seek a Turing machine that
  - Accepts a candidate string $x$ in $\Sigma^*$
  - Engages in the process of determining whether $x \in L$
  - Halts in the "Accept" state if $x \in L$
  - Is not required to behave in any specific way (may halt in a non-accepting state—*or in other words, reject* $x$—if $x \notin L$. Or it may loop forever.
- Suppose we are required to show that $L$ is RE. We must seek a Turing machine that
  - After being fired up, may never stop
  - The TM is expected to print out on the tape each member of $L$ in a finite amount of time, typically one string after the other.

For instance, $NEQ_{CFG} = \{\langle G_1, G_2\rangle \mid G_1, G_2 \text{ are CFGs and } L(G_1) \neq L(G_2)\}$ is TR.
**Many TMs may be "up to the job":** Notice that if $L$ is $L_M$ for one TM $M$, then it is the language of an infinite number ($\aleph_0$) of other machines, $M^{'}$, as we can simply pad $M$ with $i$ "no op" instructions, for every $i \in Nat$. Each such artificially bloated TM is indeed considered a distinct TM!
A recursively enumerable (RE) language is the language that an *enumerator* Turing machine can enumerate. An enumerator Turing machine is a Turing machine that has no input tape, but has an output tape. In addition, it may employ a working tape. It keeps generating (finite) strings, and appends each generated string to the output tape. $NEQ_{CFG}$ is RE.
**A language is TR if and only if it is RE.** The main argument is that given an enumerator (in the sense of RE), we can build a recognizer (in the sense of TR), and vice versa. The main reason one should study both these definitions is to gain mathematical maturity as well as gain experience building various types of TMs. It also improves one's ability to come up with innovative proofs either using the TR type of TMs or the RE types of TMs.
A recursive language $L_M$ is the language of a Turing machine $M$ that, given any $x \in L_M$, accepts $x$, and given any $y \notin L_M$, rejects $y$. In other words, $M$ does not loop on any input. Note that it is possible to have another machine $N$ such that $L(M) = L(N)$ and $N$ loops on inputs $y \notin L_M$; however, so long as there exists *one* decider $M$, we can conclude that $L_M$ is recursive (or decidable).
Another *very important* characterization of recursive languages is this: $L$ is recursive if and only if $L$ and $\overline{L}$ are RE (equivalently, are TR).
The cardinality of the set of all Turing machine descriptions is $\aleph_0$. This is because a Turing machine can be described through a finite number of bits that model its states and its transitions, and such a description can be read as a natural number (these numbers are known as Gödel numbers). On the other hand, there are $\aleph_1$ languages over $\Sigma$. Therefore, there are non-TR (non-RE) languages. This means that there exist languages in which membership testing cannot be carried out by *any* Turing machine. $EQ_{TM}$ is neither RE nor co-RE (both itself and its complement are non-RE).