# CISC 4090: Theory of Computation

## Chapter 3
## The Church-Turing Thesis

Courtesy of Arthur G. Werschulz

Fordham University
Department of Computer and Information Sciences

Spring, 2014



# Section 3.1: Turing Machines



## Turing machines: Context

Models of computation:

- Finite automata: models devices with little memory
- Pushdown automata: models devices with unlimited memory, accessible only in LIFO order.
- Turing machines: models general purpose computers



## Turing machines overview

- Introduced by Alan Turing in 1936
- Unlimited memory
  - Infinite tape that can be
    - moved left/right
    - written
  - Much less restrictive than stack of a PDA
  - A Turing machine can do everything a real computer can do (even though a simple model)
  - However, a Turing machine *cannot* solve all problems!

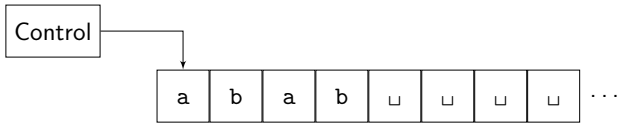
## What is a Turing machine?

Informal description

- Contains an infinite tape
- Tape initially contains the input string, with blanks everywhere else.
- Machine can read and write from tape, and move left or right after each action.
- The machine continues until it enters an *accept* state or a *reject* state, at which point it *immediately* stops and outputs ACCEPT or REJECT.
  Very different from FSAs and PDAs.
- The machine can loop forever.
  - Why can't an FSA or PDA loop forever?
  - **Answer:** FSA/PDA terminates when input string is fully processed, taking only one "action" for each input symbol processed.



## Turing machine






## Designing a Turing machine: Example #1

. . . to recognize the language $B = \{ w\#w : w \in \{0,1\}^* \}$.

- Will focus on informal descriptions, even more than we did with PDAs.
- Imagine that you are standing on an infinite tape with symbols on it, and you want to check to see if the string belongs to $B$.
  - What procedure would you use, given that you can read/write and move the tape in each direction?
  - You have a finite control, so cannot remember much. Hence you must rely on the information on the tape.
  - Try it!



## Turing machine: Example #1

- $M_1$ to recognize $B = \{ w\#w : w \in \{0,1\}^* \}$.
- $M_1$ loops. In each iteration, it matches symbols on each side of the #.
  - It does the leftmost symbol remaining.
  - It thus scans forward and backwards.
  - It crosses off the symbol it's working on. We can assume the TM replaces it with some special symbol x.
  - When scanning forward, it scans to the #, then scans to the first symbol not crossed off.
  - When scanning backwards, it scans past the # and then to the crossed-off symbol.
  - If TM discovers a mismatch, it rejects.
  - If all symbols are crossed off, then accept.
- What are the possible outcomes?
  - Accept or reject.
  - Looping is not possible: Guaranteed to terminate/halt, since each iteration makes progress.

## Sample execution

for the string 011000#011000

Tape head is at red symbol.

```
0 1 1 0 0 0 # 0 1 1 0 0 0 ␣ ␣ ...
x 1 1 0 0 0 # 0 1 1 0 0 0 ␣ ␣ ...
    ...
x 1 1 0 0 0 # x 1 1 0 0 0 ␣ ␣ ...
x 1 1 0 0 0 # x 1 1 0 0 0 ␣ ␣ ...
x x 1 0 0 0 # x 1 1 0 0 0 ␣ ␣ ...
    ...
x x x x x x # x x x x x x ␣ ␣ ...
```



## Formal definition of a Turing machine

- The key is the transition function

$$\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

- Suppose TM is in state $q$, with the head over the tape at symbol $a$. TM now executes. Then TM is in new state $r$, with a new symbol $b$ replacing $a$ on the tape, and tape head moves either left or right.



## Formal definition of a Turing machine (cont'd)

- A *Turing machine* is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$.
- $Q$ is a finite set of *states*.
- $\Sigma$ is the *input alphabet*, not containing the special symbol $\sqcup$.
- $\Gamma$ is the *tape alphabet*, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$.
- $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*.
- $q_0$, $q_{\text{accept}}$, and $q_{\text{reject}}$ are the *start*, *accept*, and *reject* states.
  - Do we need more than one accept or reject state?
  - No, since once we enter such a state, we can terminate.



## TM computation

- A TM's *configuration* consists of:
  - its current state,
  - the contents of the current tape location, and
  - the current head location.
- As a TM computes, its current configuration can change.

## Turing recognizable and decidable languages

- The set of strings that a Turing machine $M$ accepts is the *language* of $M$, or the *language recognized by* $M$, denoted $L(M)$.
- Definitions:
  - A language is *Turing recognizable* (sometimes called *recursively enumerable*) if some Turing machine recognizes it.
  - A Turing machine that halts on all inputs is a *decider*. A decider that recognizes a language *decides* it.
  - A language is (*Turing-*) *decidable* (sometimes called *recursive*) if it is decided by some Turing machine.
  - Notes:
    - Language is decidable if it's Turing recognizable by a TM that always halts, i.e., there is a decider for that language.
    - Every decidable language is Turing-recognizable.
    - It is possible for a TM to halt only on those strings it accepts.



## Turing machine: Example #2

- Design a TM $M_2$ that decides $A = \{ 0^{2^n} : n \geq 0 \}$, the language of all strings of 0s whose length is a power of two.
- Without designing it, do you think (intuitively) that this can be done? Why?
  - We could write a (C++, Java, Perl, ...) program to do this.
  - Since a TM can do anything a computer can do, this can be done by a TM.
- Solution? Basic idea: Divide by 2 each time and see whether remainder is 1: boxed

**while** *true* **do**
  sweep left to right across the tape, crossing off every other 0;
  **if** *tape has exactly one 0* **then** accept;
  ;
  **if** *tape has an odd number of 0s* **then** reject;
  return the head to the left-hand end of the tape;
  ;
**end**



## Sample execution of TM $M_2$

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | ␣ | ␣ | Number is $4 = 2^2$ |
| x | 0 | 0 | 0 | ␣ | ␣ | |
| x | 0 | x | 0 | ␣ | ␣ | Now we have $2 = 2^1$ |
| x | 0 | x | 0 | ␣ | ␣ | |
| x | 0 | x | 0 | ␣ | ␣ | |
| x | x | x | 0 | ␣ | ␣ | |
| x | x | x | 0 | ␣ | ␣ | now we have $1 = 2^0$ |
| x | x | x | 0 | ␣ | ␣ | seek back to start |
| x | x | x | 0 | ␣ | ␣ | Scan right; one 0, so accept |



## Turing machine: Example #3

- Design TM $M_3$ to decide the language

$$C = \{ a^i b^j c^k : i \times j = k \text{ and } i, j, k \geq 1 \}$$

- What does this tell us about the capability of a TM?
  - That it can do (or at least check) multiplication.
  - As we have seen before, we often use unary.
- How would you approach this?
  (Imagine that we were trying $2 \times 3 = 6$.)

## Turing machine: Example #3 (cont'd)

1. First, scan the string from left to right to verify that it is of the form $\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k$. If it is, scan to start of tape,[1] and if not, reject.
   (Easy to do with finite control/finite automaton.)
2. Cross off the first a and scan until the first b occurs. Shuttle between b's and c's, crossing off one of each until all b's are gone. If all c's have been crossed off and some b's remain, reject.
3. Restore[2] the crossed-off b's and repeat step 2 if there are any a's remaining. If all a's are gone, check if all c's are crossed off—if so, accept; if not, reject.

[1] Some subtleties here, see text. Can either use a special symbol or can backup until we realize tape is stuck and hasn't actually moved left.

[2] How to restore? Have a special cross-off symbol that incorporates the original symbol; just put an X through that symbol.



## Transducers

- We keep talking about recognizing a language, rather than generating a language. This is common in language theory.
- But now that we are talking about computation, this may seem strange and limiting.
  - Computers typically transform input into output.
  - For example, we are more likely to have a computer perform multiplication than to check that the equation is correct.
  - TMs can also generate or *transduce*.
  - How would you compute $\mathtt{c}^{i\times j}$, given $\mathtt{a}^i$ and $\mathtt{b}^j$?
  - Similar to TM $M_3$: For every a, scan through the b's; for each b, you go to the end of the string and add a c. Thus by zig-zagging $i$ times, you can generate the appropriate number of c's.



## Turing machine: Example #4

- Solve the *element uniqueness problem*:
  - Given a list of strings over $\{0,1\}$, separated by #.
    Accept if all the strings are different.
  - The language is

$$E = \{\, \#x_1\#x_2\#\ldots\#x_l : \text{each } x_i \in \{0,1\}^* \text{ and } x_i \neq x_j \text{ if } i \neq j \,\}$$

- How would you do this?



## Turing machine: Example #4 (cont'd)

1. Place a mark on top of the left-most symbol. If it was a blank, accept. If it was a #, continue. Otherwise, reject.
2. Scan right to next # and place a mark on same. If no # is encountered, we only had $x_1$, so accept.
3. By zig-zagging, compare the two strings to the right of the two marked #s. If they are equal, reject.
4. Move the rightmost of the two marks to the next # symbol to the right. If no # symbol is encountered before a blank, move the left most mark to its right and the rightmost mark to the # after that. This time, if no # is available for the rightmost mark, all the strings have been compared, so accept.
5. Go back to step 3.

## Decidability

- All these examples have been decidable.
- Showing that a language is Turing recognizable, but not decidable, is more difficult. (Cover in Chapter 4.)
- How do we know that these examples are decidable?
  - You can tell that each iteration makes progress towards the ultimate goal, and so you must reach that goal.
  - This would be clear from simply examining the "algorithm".
  - Not hard to prove formally. For example, suppose there are $n$ symbols at the start. If we erase a symbol each and every iteration, will be done in $n$ iterations.



# Section 3.2: Variants of Turing machines



## Variants of Turing machines

- We saw only a few variants of FA and PDA: deterministic and non-deterministic.
- There are several variants of Turing machines.
  - They are all equivalent. (No surprise here: a TM can compute anything that's "computable.")
  - So, choose most convenient variant for task at hand.



## TM variant #1: Let the head stay put

- Current TM model: tape head *must* move either left or right after each step.
- Sometimes convenient to allow tape head to stay put.
- This new variant is equivalent to original model. *Prove it!*
  - To show that two models are equivalent, we need only show that each can simulate the other.
  - Two machines are equivalent if they recognize the same language.
- **Proof:** We can convert any TM with "stay put" transitions to one without same by replacing each "stay put" transition with two additional transitions:
  - Move right.
  - Move left.

## TM variant #2: Multi-tape TMs

- A *multitape* TM is like an ordinary TM, but with several tapes.
- Each has its own read/write head.
- Initially: tape 1 has input string, remaining tapes are blank.
- Transition function allows reading, writing, and moving the heads on some or all tapes simultaneously.
- Multitape TM may be more convenient (think of extra tapes as "scratch paper"), but doesn't add more power.



## Proof of equivalence of variant #2

- We show how to convert a $k$-tape TM $M$ into an equivalent 1-tape TM $S$.
- $S$ simulates the $k$ tapes of $m$ using a single tape with a # as a delimiter to separate the contents of the $k$ tapes.
- $S$ marks the location of the $k$ heads by putting a dot above the appropriate symbols.



## Proof of equivalence of variant #2 (cont'd)

- On input of $w = w_1 w_2 \dots w_n$, machine $S$ will look like

$$\#\dot{w}_1 w_2 \dots w_n \# \dot{\sqcup} \# \dot{\sqcup} \dots \# \dot{\sqcup} \#$$

- To simulate a single move, $S$ scans its tape from the first # to the $(k+1)$st #, to determine the symbols under the virtual heads. Then $S$ makes a second pass to update the heads and contents, based on $M$'s transition function.
- If at any point, $S$ moves one of the virtual heads to the right of a #, this action means that $M$ has moved the head to a previously-unread blank portion of the tape. So $S$ writes a blank symbol onto this cell, and shifts everything to the right on the tape one unit to the right.



## TM variant #3: Nondeterministic TM

- Can add *nondeterminism* to Turing machines.
- Similar to what was done in adding nondeterminism to other models, e.g., moving from DFAs to NFAs: allow transition function to choose a *subset* of states.
- Transition function for deterministic TM is $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$.
- *Nondeterministic Turing machine* has transition function $\delta\colon Q \times \Gamma \to \mathscr{P}(Q \times \Gamma \times \{L, R\})$.
- Computation of NDTM: tree whose branches correspond to different possibilities for the machine.
- If some branch of computation leads to accept state, the NDTM accepts its input.

## TM variant #3: Nondeterministic TM (cont'd)

- **Proof sketch of equivalence:** Simulate any non-deterministic TM $N$ with a deterministic TM $D$.
- $D$ will try all possible branches.
- We can view branches as representing a tree, which we can explore.
  - Using depth-first search is a bad idea. This fully explores one branch before going to next. If that one loops forever, will never even try most branches.
  - Use breadth-first search. This method guarantees that all branches will be explored to any finite depth, and hence will accept if any branch accepts.
  - Hence, the DTM will accept if the NTM does.
- Text goes on to show that this can be done using 3 tapes (input, handle current branch, track position in computation tree).



## Enumerators

- An *enumerator* $E$ is a TM with a printer attached.
- Can send strings to be output to the printer.
- Input tape is initially blank.
- *Language enumerated by* $E$ is the set of strings printed out.
- $E$ may not halt, and may print out infinitely-many strings.
- **Theorem:** A language is Turing-recognizable iff some enumerator enumerates it.



## Proof of enumerator equivalence

- First direction: If enumerator $E$ enumerates language $A$, then a TM $M$ recognizes $A$.
  For every $w$ generated by $E$, the TM $M$ will run $E$ (a TM) and check to see if the output matches $w$. If it ever matches, then accept.
- Other direction: If a TM $M$ recognizes a language $A$, we construct an enumerator $E$ for $AA$ as follows:
  - Let $s_1, s_2, s_3, \ldots$ be the list of all possible strings in $\Sigma^*$.
  - For $i = 1, 2, \ldots$
    - Run $M$ for $i$ steps on each input $s_1, s_2, \ldots, s_i$.
    - If any computation is accepted, then print corresponding $s_j$.
  - Loop over $i$ implements a breadth-first search, which eventually generates everything without getting stuck.



## Equivalence with other models

- Many TM variants proposed, some of which may appear very different.
  - All have unlimited access to unlimited memory.
  - All models with this feature turn out to be equivalent, under reasonable assumptions (i.e., that one can only do a finite amount of work in one step).
- Thus TMs are universal model of computation. The classes of algorithms are the same, independent of the specific model of computation.
- To get some insight, note that all programming languages are equivalent. For example, one can write a compiler for any language in any other language (assuming basic constructs are available).

## Section 3.3: Definition of an Algorithm



## What is an algorithm?

- How would you describe an algorithm?
- An algorithm is a collection of simple instructions for carrying out some task.
  - A procedure or recipe.
  - Algorithms abound in mathematics.
  - Ancient algorithms (thousands of years old):
    - Finding prime numbers (Eratosthenes).
    - Computing greatest common divisor (Euclid).



## Hilbert's Problems

- In 1900, David Hilbert proposed 23 mathematical problems for the next century.
- Hilbert's Tenth Problem:
  - Devise an algorithm for determining whether a multivariate integer polynomial has an integral root, i.e., whether there exist $x_1, x_2, \ldots, x_n \in \mathbb{Z}$ such that

$$p(x_1, x_2, \ldots, x_n) = 0.$$

  - Instead of "algorithm", Hilbert said "a process by which it can determined by a finite number of operations".
  - For example, the polynomial $6x^3yz^2 + 3xy^2 - x^3 - 10$ has an integral root $(x, y, z) = (5, 3, 0)$.
  - He assumed that such a method exists.
  - He was wrong.



## Church-Turing Thesis

- Can't prove nonexistence results for algorithms without precise definition of "algorithm".
- Definition provided in 1936:
  - $\lambda$-calculus (A. Church).
  - TMs (A. Turing).
  - These were shown to be equivalent.
  - Church-Turing thesis: "computable" means "TM-computable". Connection between intuitive and formal notions of algorithm.
  - In 1970, Yuri Matiyasevich (PhD dissertation) showed that no algorithm exists for testing whether an arbitrary integer polynomial has integral roots. (Sometimes called "DPRM theorem".)
    Of course, there's always an answer ("yes" or "no"); but there's no algorithm that gives us the answer.

## More on Hilbert's Tenth Problem

- Hilbert essentially asked whether

  $D = \{$ multivariate integer polynomials $p : p$ has an integral root $\}$

  is decidable (not just Turing-recognizable).
- Can you come up with a procedure to answer this question?
  - Univariate case: Try all possible integers $0, \pm 1, \pm 2, \ldots$.
    If one of them works, answer is "yes".
  - Multivariate case: Lots of combinations.
  - The difficulty here?
    - May not terminate in a "reasonable" amount of time.
    - You don't *know* if that's because it actually doesn't terminate, or because it didn't run long enough.
  - This shows that problem is Turing-recognizable, but can't be used to determine whether it's Turing-decidable.
  - One might've hoped that something else would work.
    DPRM dashes that hope.



## More on Hilbert's Tenth Problem (cont'd)

- For the univariate case, there is actually an upper bound on the root of the polynomial.
  So in this case, previous algorithm always terminates, and so univariate problem is solvable.
- Think about the significance of the fact that you can prove that something cannot be computed.
  - Does *not* that you're not smart enough to compute it!
  - More in Chapter 4.



## Ways to describe Turing machines

- As we have seen before, we can specify the design of a machine (FA, PDA) formally or informally.
  - Ditto for TM.
  - Informal description still describes the implementation of the machine; just less formally.
- With a TM, we can actually go up an additional level of informality.
  - Don't need to describe machine in terms of tape heads and the like.
  - Can describe algorithmically.
  - We will describe TMs ...
    - informally at the implementation level, or
    - algorithmically.



## Turing machine terminology

- For the algorithmic level.
- Input to a TM: a *string*.
  - Other objects (graphs, lists, etc.) must be encoded as a string.
  - We let $\langle O \rangle$ denote the encoding of object $O$.
- We implicitly assume that TM checks input to make sure it follows the proper encoding, rejecting same if not proper.

## Example: Algorithmic level

- Let $A$ be language of all strings representing graphs that are *connected*, i.e., any node can be reached from any other node.

$$A = \{ \langle G \rangle : G \text{ is a connected undirected graph} \}.$$

- Give me a high-level description of the TM $M$ for deciding $A$.



## Example: Algorithmic level (cont'd)

Algorithmic description of $M$?

algoruled
**parameter**: the encoding $\langle G \rangle$ of an undirected graph $G = (V, E)$
**return** : ACCEPT if $G$ is connected, REJECT otherwise.

select and mark the first node in $V$;
**repeat**
  **foreach** *node* $v \in V$ **do**
    **if** $\exists$ *edge* $(v, w)$ *with previously-marked* $w \in V$ **then**
    mark $v$;
    ;
  **end**
**until** *no new nodes are marked*;
**foreach** $v \in V$ **do**
  **if** $v$ *is not marked* **then** r;
  eturn(REJECT) ;
**end**
return(ACCEPT);

**Function** connected($\langle G \rangle$)