Introduction to Languages and Computation

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

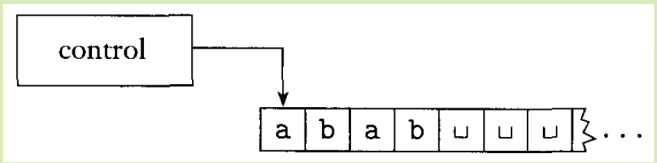
LSSU Math 400

Subsection 1

Turing Machines

The Turing Machine Model

- A much more powerful model than finite automata and pushdown automata is the Turing machine.
- It has an unlimited and unrestricted memory and can do everything that a real computer can do.
- Even a Turing machine cannot solve certain problems; these problems are beyond the theoretical limits of computation.
- The Turing machine model has:



  - An infinite tape as its unlimited memory, which, initially, contains only the input string;
  - A tape head that can read and write symbols and move around on the tape; If the machine needs to store information, it may write it on the tape. To read what it has written, it can move its head back over it.
  - The machine either outputs accept or reject by entering designated accepting or rejecting states, or goes on forever, never halting.

Turing Machines Versus Finite Automata

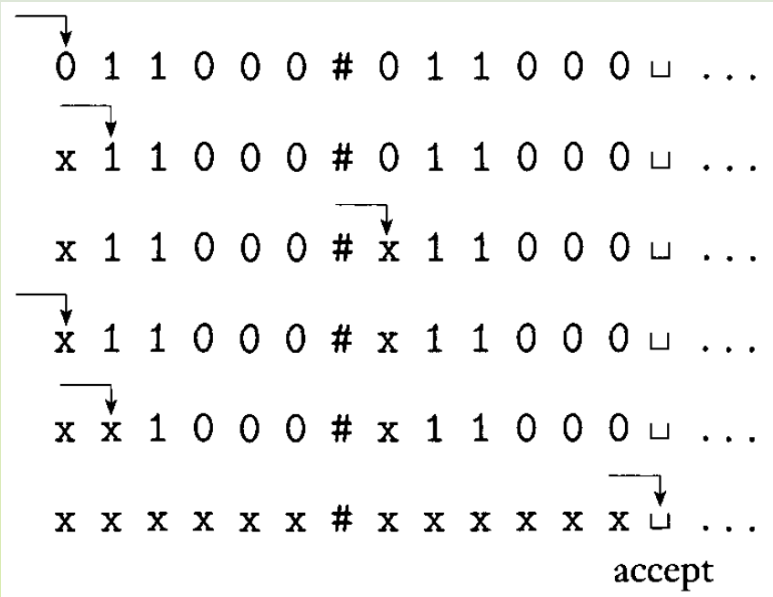
- The differences between finite automata and Turing machines:
  1. A Turing machine can both write on the tape and read from it.
  2. The read-write head can move both to the left and to the right.
  3. The tape is infinite.
  4. The special states for rejecting and accepting take effect immediately.

A Turing Machine M_1

- We introduce a Turing machine M_1 for testing membership in the language $B = \{w\#w : w \in \{0,1\}^*\}$. We want M_1 to accept if its input is a member of B and to reject otherwise.
  To determine whether the input comprises two identical strings separated by a # symbol, M_1 zig-zags to the corresponding places on the two sides of the # and determines whether they match. It does this by placing marks on the tape to keep track of which places correspond.
- M_1: On input string w:
  1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.
  2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, reject; otherwise, accept.

A Computation of M_1

- The following figure contains several snapshots of M_1's tape while it is computing on input 011000#011000:



- The description of Turing machine M_1 sketches the way it functions but does not give all its details.
- We can describe Turing machines formally by specifying each of the parts of the definition of the Turing machine model.
- This is almost never done because formal descriptions tend to be long.

Turing Machines

- The transition function δ takes the form: $Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$.
- $\delta(q, a) = (r, b, \mathrm{L})$ means when the machine is in a certain state q and the head is over a tape square containing a symbol a, the machine writes the symbol b replacing the a, goes to state r and the head moves to the left or right after writing.

Definition (Turing Machine)

A **Turing Machine** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of **states**,
2. Σ is the **input alphabet** not containing the blank symbol $\sqcup$,
3. Γ is the **tape alphabet**, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the **transition function**,
5. $q_0 \in Q$ is the **start state**,
6. $q_{\text{accept}} \in Q$ is the **accept state**, and
7. $q_{\text{reject}} \in Q$ is the **reject state**, where $q_{\text{reject}} \neq q_{\text{accept}}$.

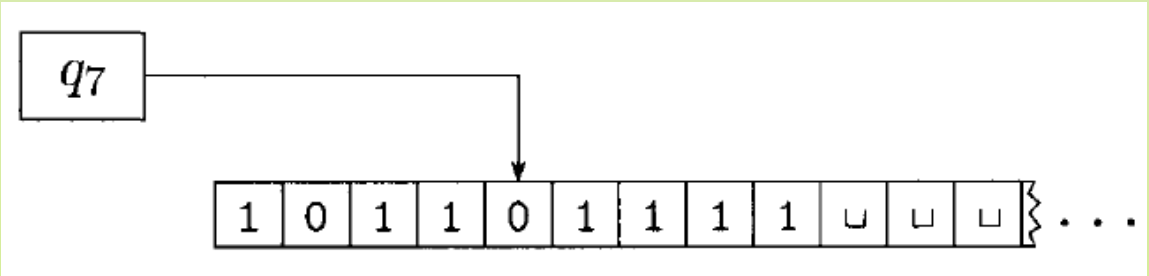
Computation of a Turing Machine

- A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ computes as follows.
  - Initially M receives its input $w = w_1 w_2 \cdots w_n \in \Sigma^*$ on the leftmost n squares of the tape, and the rest of the tape is blank.
  - The head starts on the leftmost square of the tape. The first blank appearing on the tape marks the end of the input.
  - Once M has started, the computation proceeds according to the rules described by the transition function.
  - If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move.
  - The computation continues until it enters either the accept or reject states at which point it halts. If neither occurs, M goes on forever.

Configurations of a Turing Machine

- A setting of the current state, the current tape contents, and the current head location is called a **configuration** of the Turing machine.
- For a state q and two strings u and v over the tape alphabet Γ we write uqv for the configuration where:
  - the current state is q,
  - the current tape contents are uv,
  - the current head location is the first symbol of v.

  The tape contains only blanks following the last symbol of v.
- Example: $1011q_701111$ represents the configuration



Formalizing Computation

- Configuration C_1 **yields** configuration C_2 if the Turing machine can legally go from C_1 to C_2 in a single step:
  - Suppose that we have a, b and c in Γ, as well as u and v in Γ^* and states q_i and q_j.
  - In that case uaq_ibv and uq_jacv are two configurations.
  - Say that
    $$uaq_ibv \quad \text{yields} \quad uq_jacv$$
    if in the transition function $\delta(q_i, b) = (q_j, c, \mathrm{L})$. That handles the case where the Turing machine moves leftward.
  - For a right move,
    $$uaq_ibv \quad \text{yields} \quad uacq_jv$$
    if $\delta(q_i, b) = (q_j, c, \mathrm{R})$.
  - When the head is at the left-hand end, the configuration q_ibv yields q_jcv if the transition is left moving; it yields cq_jv for the right-moving transition.
  - For the right-hand end, the configuration uaq_i is equivalent to $uaq_i\sqcup$ because blanks follow the part represented in the configuration.

Halting Configurations and Acceptance

- The **start configuration** of M on input w is the configuration $q_0 w$; in the start state q_0 with the head at the leftmost position.
- In an **accepting configuration** the state of the configuration is q_{accept}.
- In a **rejecting configuration** the state of the configuration is q_{reject}.
- Accepting and rejecting configurations are **halting configurations** and do not yield further configurations.
- Because the machine is defined to **halt** when in the states q_{accept} and q_{reject}, the transition function could equivalently be expressed as $\delta : Q' \times \Gamma \to Q \times \Gamma \times \{\text{L}, \text{R}\}$, where Q' is Q without q_{accept} and q_{reject}.
- A Turing machine M **accepts** input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists, where
  1. C_1 is the start configuration of M on input w,
  2. each C_i yields C_{i+1}, and
  3. C_k is an accepting configuration.

Turing Recognizable Languages

- The collection of strings that M accepts is **the language of** M, or **the language recognized by** M, denoted $L(M)$.

Definition (Turing-Recognizable Language)

A language is called **Turing-recognizable** (or **recursively enumerable**) if some Turing machine recognizes it.

- When we start a Turing machine on an input, three outcomes are possible:
  - accept;
  - reject;
  - loop, where "loop" means that the machine does not halt.
- A Turing machine M can fail to accept an input by entering the q_{reject} state and rejecting, or by looping.

Turing Decidable Languages

- Turing machines that halt on all inputs, i.e., that never loop, are called **deciders** because they always make a decision to accept or reject.
- A decider that recognizes some language also is said to **decide** that language.

Definition (Turing-Decidable Language)

A language is called **Turing-decidable** or, simply, **decidable** (or **recursive**) if some Turing machine decides it.

- Every decidable language is Turing-recognizable.
- There are Turing recognizable but not Turing decidable languages, as will be seen later.

Higher-Level versus Formal Descriptions

- As we did for finite and pushdown automata, we can formally describe a particular Turing machine by specifying each of its seven parts.
- Doing this is cumbersome for all but the tiniest Turing machines.
- One usually gives only higher level descriptions because they are precise enough and much easier to understand.
- It is important to remember that every higher level description is actually just shorthand for its formal counterpart.

A Turing Machine M_2: Informal Description

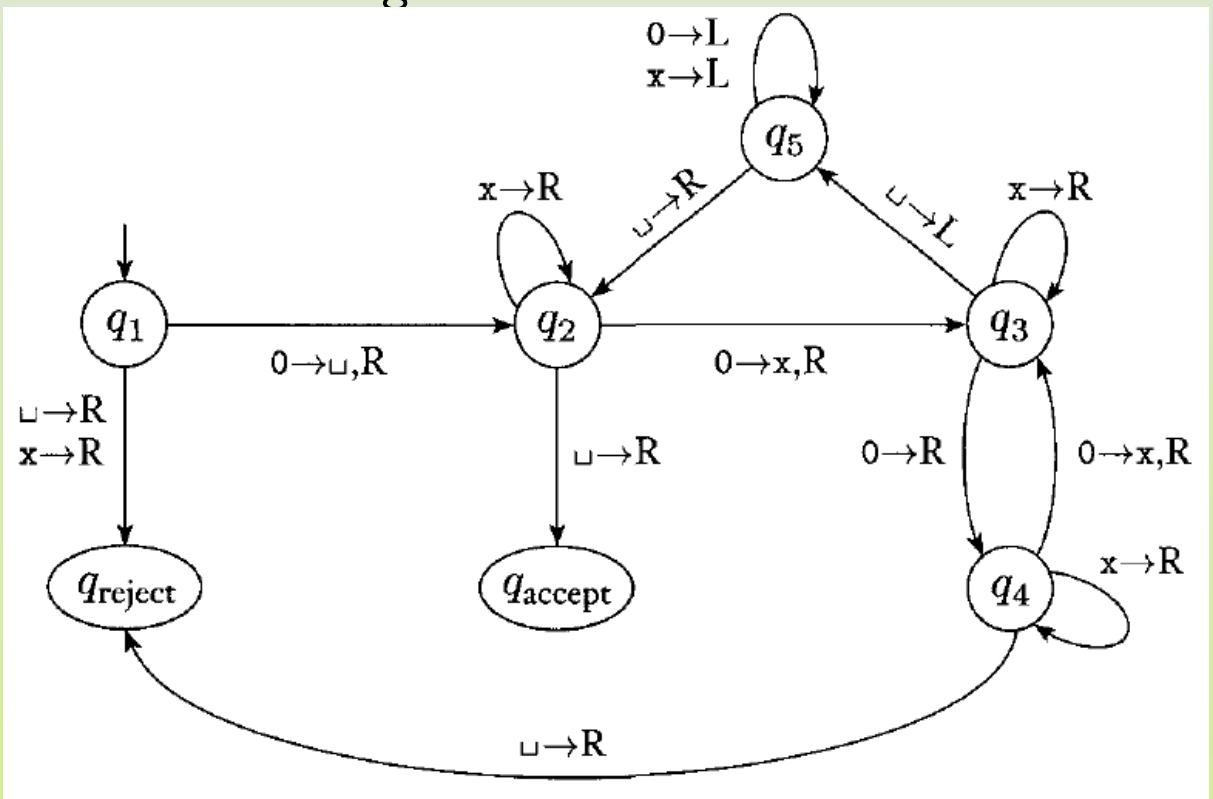
- A Turing machine (TM) M_2 that decides $A = \{0^{2^n} : n \geq 0\}$, the language consisting of all strings of 0s whose length is a power of 2:

  M_2 : On input string w

  1. Sweep left to right across the tape, crossing off every other 0.
  2. If in stage 1 the tape contained a single 0, accept.
  3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, reject.
  4. Return the head to the left-hand end of the tape.
  5. Go to stage 1.
- Each iteration of stage 1 cuts the number of 0s in half.
- As the machine sweeps across the tape in stage 1, it keeps track of whether the number of 0s seen is even or odd.
  - If that number is odd and greater than 1, the original number of 0s in the input could not have been a power of 2. The machine rejects.
  - If the number of 0s seen is 1, the original number must have been a power of 2. So in this case the machine accepts.

A Turing Machine M_2: Formal Description

- The formal description: $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$:
  - $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
  - $\Sigma = \{0\}$,
  - $\Gamma = \{0, x, \sqcup\}$,
  - δ is given with a state diagram:



- The start, accept, and reject states are q_1, q_{accept} and q_{reject}.

Remarks on the State Diagram

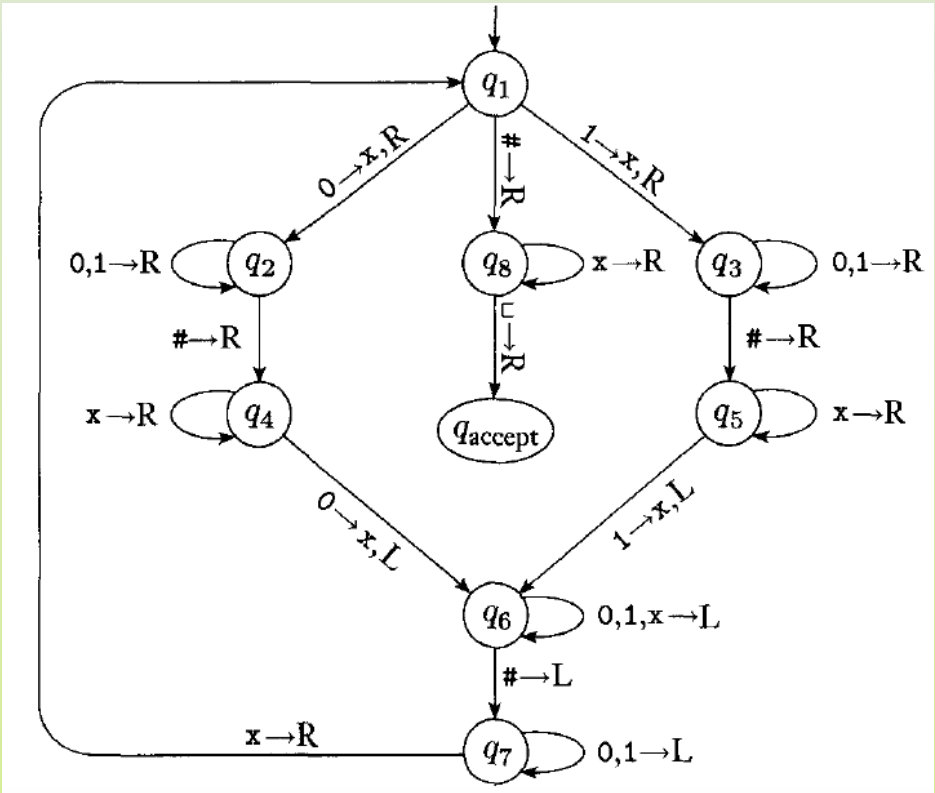
- The label $0 \to \sqcup, \mathrm{R}$ appears on the transition from q_1 to q_2. This label signifies that, when in state q_1 with the head reading 0, the machine goes to state q_2, writes $\sqcup$, and moves the head to the right. In other words, $\delta(q_1, 0) = (q_2, \sqcup, \mathrm{R})$.
- For clarity we use the shorthand $0 \to \mathrm{R}$ in the transition from q_3 to q_4, to mean that the machine moves to the right when reading 0 in state q_3 but does not alter the tape, so $\delta(q_3, 0) = (q_4, 0, \mathrm{R})$.
- This machine begins by writing a blank symbol over the leftmost 0 on the tape so that it can find the left-hand end of the tape in Stage 4. Normally a more suggestive symbol, such as #, is used for the left-hand end delimiter, but using a blank here helped keep the tape alphabet and state diagram small.

A sample run of this machine on input 0000, i.e., with starting configuration $q_1 0000$:

$q_1 0000$	$\sqcup q_5 \mathrm{x} 0 \mathrm{x} \sqcup$	$\sqcup \mathrm{x} q_5 \mathrm{xx} \sqcup$
$\sqcup q_2 000$	$q_5 \sqcup \mathrm{x} 0 \mathrm{x} \sqcup$	$\sqcup q_5 \mathrm{xxx} \sqcup$
$\sqcup \mathrm{x} q_3 00$	$\sqcup q_2 \mathrm{x} 0 \mathrm{x} \sqcup$	$q_5 \sqcup \mathrm{xxx} \sqcup$
$\sqcup \mathrm{x} 0 q_4 0$	$\sqcup \mathrm{x} q_2 0 \mathrm{x} \sqcup$	$\sqcup q_2 \mathrm{xxx} \sqcup$
$\sqcup \mathrm{x} 0 \mathrm{x} q_3 \sqcup$	$\sqcup \mathrm{xx} q_3 \mathrm{x} \sqcup$	$\sqcup \mathrm{x} q_2 \mathrm{xx} \sqcup$
$\sqcup \mathrm{x} 0 q_5 \mathrm{x} \sqcup$	$\sqcup \mathrm{xxx} q_3 \sqcup$	$\sqcup \mathrm{xx} q_2 \mathrm{x} \sqcup$
$\sqcup \mathrm{x} q_5 0 \mathrm{x} \sqcup$	$\sqcup \mathrm{xx} q_5 \mathrm{x} \sqcup$	$\sqcup \mathrm{xxx} q_2 \sqcup$
		$\sqcup \mathrm{xxx} \sqcup q_{\mathrm{accept}}$

Formal Description of M_1

- The machine $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$, informally described previously, for deciding the language $B = \{w\#w : w \in \{0,1\}^*\}$:
  - $Q = \{q_1, \ldots, q_{14}, q_{\text{accept}}, q_{\text{reject}}\}$,
  - $\Sigma = \{0, 1, \#\}$, and $\Gamma = \{0, 1, \#, x, \sqcup\}$,
  - δ is given by



  - The start, accept, and reject states are q_1, q_{accept}, and q_{reject}.

Some Remarks on M_1

- In the state diagram of TM M_1, the label $0, 1 \rightarrow \mathrm{R}$ on the transition going from q_3 to itself means that the machine stays in q_3 and moves to the right when it reads a 0 or a 1 in state q_3. It does not change the symbol on the tape.
- Stage 1 is implemented by states q_1 through q_6, and Stage 2 by the remaining states.
- To simplify the figure, we do not show the reject state or the transitions going to the reject state. Those transitions occur implicitly whenever a state lacks an outgoing transition for a particular symbol. Thus, because in state q_5 no outgoing arrow with a # is present, if a # occurs under the head when the machine is in state q_5, it goes to state q_{reject}. For completeness, we say that the head moves right in each of these transitions to the reject state.

A Turing Machine M_3: Informal Description

- The TM M_3 decides $C = \{a^i b^j c^k : i \times j = k \text{ and } i, j, k \geq 1\}$.

  M_3: On input string w

  1. Scan the input from left to right to determine whether it is a member of $a^*b^*c^*$ and reject if it is not.
  2. Return the head to the left-hand end of the tape.
  3. Cross off an a and scan to the right until a b occurs. Shuttle between the b's and the 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.
  4. Restore the crossed off b's and repeat Stage 3 if there is another a to cross off. If all a's have been crossed off, determine whether all c's also have been crossed off. If yes, accept; otherwise, reject.

A Turing Machine M_3: The Operation

- In Stage 1 the machine operates like a finite automaton. The head moves from left to right, keeping track by using its states to determine whether the input is in the proper form.
- In Stage 2 the TM finds the left-hand end of the input tape by marking the leftmost symbol in some way when the machine starts with its head on that symbol. Then the machine may scan left until it finds the mark when it wants to reset its head to the left-hand end.
- Another method takes advantage of the fact that, if the machine tries to move its head beyond the left-hand end of the tape, it stays in the same place. Thus, to detect the left-hand end, the machine can write a special symbol over the current position, while recording the symbol that it replaced in the control. Then, if an attempt to move the head to the left still leaves the head over the special symbol, the head must have been at the left-hand end. Before going farther, the machine must be sure to restore the changed symbol to the original.

A Turing Machine M_4: Informal Description

- TM M_4 solves the **element distinctness problem**. It is given a list of strings over $\{0,1\}$ separated by #s and its job is to accept if all the strings are different, i.e., the language is

$$E = \{\#x_1\#x_2\#\cdots\#x_i : \text{each } x_i \in \{0,1\}^* \text{ and } x_i \neq x_j, \text{ for all } i \neq j\}.$$

- Machine M_4 works by
  - first comparing x_1 with x_2 through x_i,
  - then by comparing x_2 with x_3 through x_i,
  - and so on.

A Turing Machine M_4: Informal Description (Cont'd)

M_4: On input w

1. Place a mark on top of the leftmost tape symbol. If that symbol was a blank, accept. If that symbol was a #, continue with the next stage. Otherwise, reject.
2. Scan right to the next # and place a second mark on top of it. If no # is encountered before a blank symbol, only x_1 was present, so accept.
3. By zig-zagging, compare the two strings to the right of the 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 symbol, move the leftmost mark to the next # 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 to Stage 3.

The Technique of Marking Tape Symbols

- This machine illustrates the technique of marking tape symbols.
- In Stage 2, the machine places a mark above a symbol, # in this case.
- In the actual implementation, the machine has two different symbols, # and $\dot{\#}$, in its tape alphabet.
- Saying that the machine places a mark above a # means that the machine writes the symbol $\dot{\#}$ at that location.
- Removing the mark means that the machine writes the symbol without the dot.
- In general, we may want to place marks over various symbols on the tape. To do so, we include versions of all these tape symbols with dots in the tape alphabet.

Subsection 2

Variants of Turing Machines

Variants of Turing Machines and Robustness

- There exist many alternative definitions of Turing machines, including versions with multiple tapes or with nondeterminism.
- They are called **variants** of the Turing machine model.
- The original model and its reasonable variants all have the same power, i.e., they recognize the same class of languages.
- This invariance to certain changes in the definition is termed **robustness**.
- Both finite automata and pushdown automata are somewhat robust, but Turing machines have an astonishing degree of robustness.
- If we allow the Turing machine the ability to stay put, the transition function becomes $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}$. This feature does not allow Turing machines to recognize additional languages, because we can convert any TM with the "stay put" feature to one that does not have it. We do so by replacing each stay put transition with two transitions, one to the right and the second back to the left.

Multitape Turing Machines

- A **multitape Turing machine** is like an ordinary Turing machine with several tapes.
- Each tape has its own head for reading and writing.
- Initially the input appears on Tape 1, and the others start out blank.
- The transition function is changed to allow for reading, writing, and moving the heads on some or all of the tapes simultaneously. Formally, it is $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}^k$, where k is the number of tapes. The expression

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, \mathrm{L}, \mathrm{R}, \ldots, \mathrm{L})$$

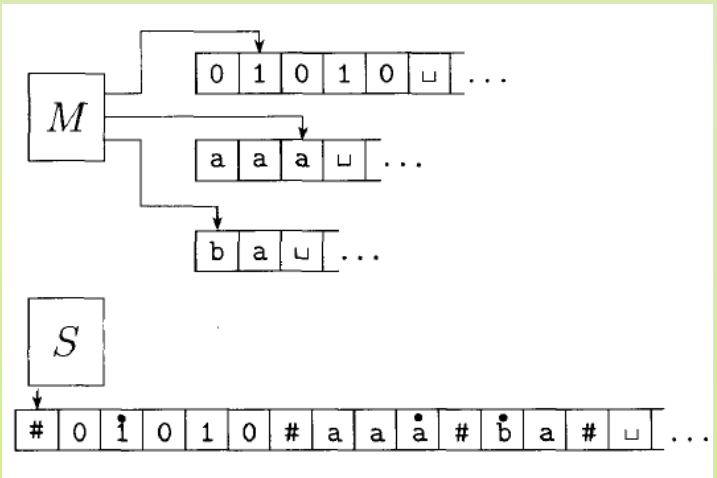
  means that, if the machine is in state q_i and heads 1 through k are reading symbols a_1, through a_k, the machine goes to state q_j, writes symbols b_1, through b_k, and directs each head to move left or right, or to stay put, as specified.
- Multitape Turing machines appear to be more powerful than ordinary Turing machines, but they are equivalent in power.

Equivalence of Multitape and Single-Tape TMs

Theorem

Every multitape Turing machine has an equivalent single-tape Turing machine.

- We show how to convert a multitape TM M to an equivalent single-tape TM S. We show how to simulate M with S. Say that M has k tapes. Then S simulates the effect of k tapes by storing their information on its single tape. It uses the new symbol # as a delimiter to separate the contents of the different tapes.
  In addition to the contents of these tapes, S must keep track of the locations of the heads. It does so by writing a tape symbol with a dot above it to mark the place where the head on that tape would be:



Simulation Algorithm

S: On input $w = w_1 \dots w_n$

1. First S puts its tape into the format that represents all k tapes of M. The formatted tape contains $\#\dot{w}_1 w_2 \cdots w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \cdots \#$.
2. To simulate a single move, S scans its tape from the first $\#$ to the $(k+1)$st $\#$ in order to determine the symbols under the virtual heads. Then S makes a second pass to update the tapes according to the way that M's transition function dictates.
3. If at any point S moves one of the virtual heads to the right onto a $\#$, this action signifies that M has moved the corresponding head onto the previously unread blank portion of that tape. So S writes a blank symbol on this tape cell and shifts the tape contents, from this cell until the rightmost $\#$, one unit to the right. Then it continues the simulation as before.

Corollary

A language is Turing-recognizable if and only if some multitape Turing machine recognizes it.

Nondeterministic Turing Machines

- A **nondeterministic Turing machine** is defined in the expected way: At any point in a computation the machine may proceed according to several possibilities.
- The transition function for a nondeterministic Turing machine has the form $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})$.
- The **computation** of a nondeterministic Turing machine is a tree whose branches correspond to different possibilities for the machine.
- If some branch of the computation leads to the accept state, the machine **accepts** its input.
- We show that nondeterminism does not affect the power of the Turing machine model.

Equivalence of Nondeterministic and Deterministic TMs

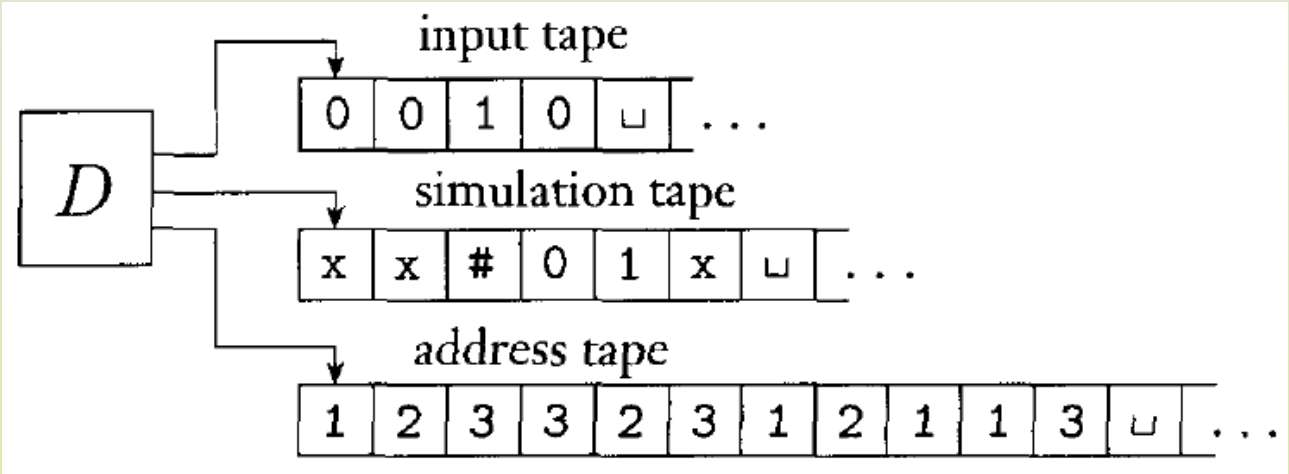
Theorem

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

- We can simulate any nondeterministic TM N with a deterministic TM D. D tries all possible branches of N's nondeterministic computation. If D ever finds the accept state, D accepts. Otherwise, D's simulation will not terminate.
- We view N's computation on an input w as a tree. Each branch of the tree represents one of the branches of the nondeterminism. Each node of the tree is a configuration of N. The root of the tree is the start configuration. The TM D searches this tree for an accepting configuration. Conducting this search carefully is crucial lest D fail to visit the entire tree. So D is designed to explore the tree by using breadth first search.

Configuration of D

- The simulating deterministic TM D has three tapes. This arrangement is equivalent to having a single tape. The machine D uses its three tapes in a particular way:



  - Tape 1 always contains the input string and is never altered.
  - Tape 2 maintains a copy of N's tape on some branch of its nondeterministic computation.
  - Tape 3 keeps track of D's location in N's nondeterministic computation tree.

Description of the Address Tape

- Every node in the tree can have at most b children, where b is the size of the largest set of possible choices given by N's transition function.
- To every node in the tree we assign an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$. E.g., we assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's 1st child.
- Each symbol in the string tells us which choice to make next when simulating a step in one branch in N's nondeterministic computation.
- Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration. In that case the address is invalid and does not correspond to any node.
- Tape 3 contains a string over Σ_b. It represents the branch of N's computation from the root to the node addressed by that string, unless the address is invalid.
- The empty string is the address of the root of the tree.

Description of D

D: On input w

1. Initially Tape 1 contains the input w, and Tapes 2 and 3 are empty.
2. Copy Tape 1 to Tape 2.
3. Use Tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N consult the next symbol on Tape 3 to determine which choice to make among those allowed by N's transition function.
   - If no more symbols remain on Tape 3 or if this nondeterministic choice is invalid, abort this branch by going to Stage 4.
   - Also go to Stage 4 if a rejecting configuration is encountered.
   - If an accepting configuration is encountered, accept the input.
4. Replace the string on Tape 3 with the lexicographically next string. Simulate the next branch of N's computation by going to Stage 2.

Turing Recognizability and Turing Decidability

Corollary

A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.

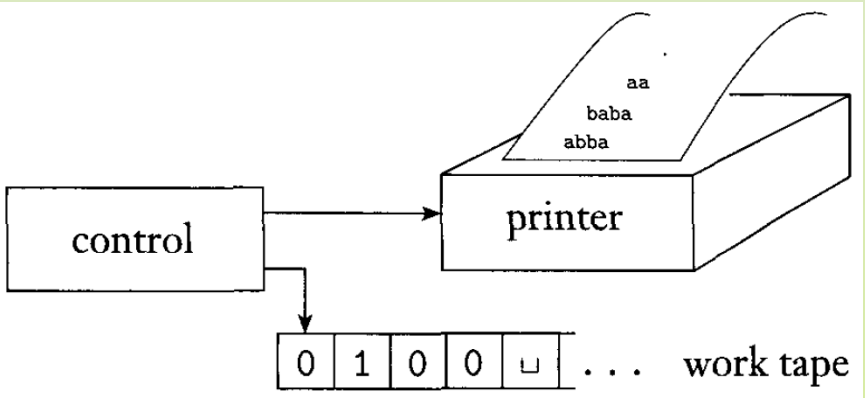
- Any deterministic TM is automatically a nondeterministic TM, and so one direction of this theorem follows immediately.
  The other direction follows from the theorem.
- We can modify the proof so that, if N always halts on all branches of its computation, D will also always halt.
- We call a nondeterministic Turing machine a **decider** if all branches halt on all inputs.

Corollary

A language is decidable if and only if some nondeterministic Turing machine decides it.

Enumerators

- Some people use the term **recursively enumerable language** for Turing recognizable language.
- That term originates from a type of Turing machine variant called an **enumerator**, which is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings. Every time the Turing machine wants to add a string to the output list, it sends the string to the printer:



  - An enumerator E starts with a blank input tape.
  - If the enumerator does not halt, it may print an infinite list of strings.
  - The **language enumerated by** E is the collection of all the strings that it eventually prints out.
  - E may generate the strings of the language in any order, possibly with repetitions.

Enumerators as Recognizers

Theorem

A language is Turing-recognizable if and only if some enumerator enumerates it.

- Let E be an enumerator that enumerates a language A. A TM M recognizing A works as follows:
  M: On input w
  1. Run E. Every time that E outputs a string, compare it with w.
  2. If w ever appears in the output of E, accept.

  Clearly, M accepts those strings that appear on E's list.
- If TM M recognizes a language A, the following E enumerates A. Say that $s_1, s_2, s_3, \ldots$ is a list of all possible strings in Σ^*.
  E: Ignore the input.
  1. Repeat the following for $i = 1, 2, 3, \ldots$.
  2. Run M for i steps on each input, $s_1, s_2, \ldots, s_i$.
  3. If any computations accept, print out the corresponding s_j.

  If M accepts a string s, s will appear on the list generated by E.

Equivalence of Computational Models: Algorithms

- Many other models of general purpose computation have been proposed: All share the essential feature of Turing machines, i.e., unrestricted access to unlimited memory.
- They turn out to be equivalent in power, so long as they satisfy reasonable requirements, e.g., the ability to perform only a finite amount of work in a single step.
- A similar phenomenon occurs with programming languages: Many, such as Pascal and LISP, look quite different from one another in style and structure. But every algorithm that can be programmed in one of them can be programmed in all others.
- Equivalence of computational models has the same root: Any two computational models that satisfy certain reasonable requirements can simulate one another and hence are equivalent in power.

Even though we can imagine many different computational models, the class of algorithms that they describe remains the same.

Subsection 3

The Definition of Algorithm

Algorithms Informally

- Informally speaking, an algorithm is a collection of simple instructions for carrying out some task.
- Algorithms sometimes are called procedures or recipes.
- The mathematical literature contains descriptions of algorithms for a multitude of tasks, such as finding prime numbers and greatest common divisors.
- Even though algorithms have had a long history in mathematics, the notion of algorithm itself was not defined precisely until the twentieth century.
- Before that, mathematicians had an intuitive notion of what algorithms were, sufficient for describing them, but insufficient for gaining a deeper understanding of algorithms.

Hilbert's Tenth Problem

- In 1900, David Hilbert identified twenty-three challenging mathematical problems for the coming century.
- The tenth problem on his list concerned algorithms.
- A **polynomial** is a sum of terms, where each **term** is a product of certain variables and a constant called a **coefficient**. For example, $6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z = 6x^3yz^2$ is a term with coefficient 6, and $6x^3yz^2 + 3xy^2 - x^3 - 10$ is a polynomial with four terms.
- For this discussion, we consider only integer coefficients.
- A **root** of a polynomial is an assignment of values to its variables so that the value of the polynomial is 0.
- A root is **integral** if all the variables are assigned integer values.
- **Hilbert's tenth problem** was to devise an algorithm that tests whether a polynomial has an integral root.
- We now know no algorithm exists for this task; it is algorithmically unsolvable. Proving that no algorithm exists requires having a clear definition of algorithm.

The Church-Turing Thesis

- The definition came in 1936 by Alonzo Church and Alan Turing.
  - Church used a notational system called the λ-calculus to define algorithms.
  - Turing did it with his "machines".
  - These two definitions were shown to be equivalent.

  This connection between the informal notion of algorithm and the precise definition has come to be called the **Church-Turing thesis**.

Intuitive notion of algorithms	equals	Turing machine algorithms.

- In 1970, Yuri Matiyasevich, building on work of Martin Davis, Hilary Putnam, and Julia Robinson, showed that no algorithm exists for testing whether a polynomial has integral roots.
- In the next set of slides we develop the techniques for proving that problems are algorithmically unsolvable.

Single-Variable Polynomials

- Let $D = \{p : p \text{ is a polynomial with an integral root}\}$.
- Hilbert's tenth problem asks whether the set D is decidable.
  - The answer is negative.
- In contrast we can show that D is Turing-recognizable.
- We first consider a simpler problem, an analog of Hilbert's tenth problem for polynomials that have only a single variable, such as $4x^3 - 2x^2 + x - 7$.

$$D_1 = \{p : p \text{ is a polynomial over } x \text{ with an integral root}\}.$$

- Here is a TM M_1 that recognizes D_1:

  M_1: The input is a polynomial p over the variable x.

  1. Evaluate p with x set successively to $0, 1, -1, 2, -2, 3, -3, \ldots$ If at any point the polynomial evaluates to 0, accept.
- If p has an integral root, M_1 eventually will find it and accept. If p does not have an integral root, M_1 will run forever.

Comparison with Multi-Variable Polynomials

- For the multivariable case, we can present a similar TM M that recognizes D. M goes through all possible settings of its variables to integral values.
- Both M_1 and M are recognizers but not deciders.
- We can convert M_1 to be a decider for D_1: We can calculate bounds within which the roots of a single variable polynomial must lie and restrict the search to these bounds.
  - We can show that the roots of such a polynomial must lie between the values $\pm k \frac{c_{\max}}{c_1}$, where k is the number of terms in the polynomial, $c_{\max}$ is the coefficient with largest absolute value, and c_1 is the coefficient of the highest order term.
  - If a root is not found within these bounds, the machine rejects.
- Matiyasevich's theorem shows that calculating such bounds for multi-variable polynomials is impossible.

Turing Machines as Algorithms

- Even though we continue to speak of Turing machines, the real focus will be on algorithms.
- Turing machines serve as a precise model for defining algorithms.
- To standardize the way we describe Turing machine algorithms, we may adopt one of three possibilities:
  - The first is the formal description that spells out in full the Turing machine's states, transition function, and so on. It is the lowest, most detailed, level of description.
  - The second is a higher level of description, called the implementation description, in which one describes the way that the Turing machine moves its head and the way that it stores data on its tape. At this level we do not give details, such as states or the transition function.
  - Third is the high-level description, wherein we use English prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head.
- We adopt the high-level description from now on.

Encoding Inputs in Informal Descriptions

- The input to a Turing machine is always a string. An object other than a string must first be represented as a string. A Turing machine may be programmed to decode the representation so that it can be interpreted in the way we intend.
- Our notation for the encoding of an object O into its representation as a string is $\langle O \rangle$. If we have several objects $O_1, O_2, \ldots, O_k$, we denote their encoding into a single string by $\langle O_1, O_2, \ldots, O_k \rangle$.
- The encoding itself can be done in many reasonable ways. It does not matter which one we pick because a Turing machine can always translate one such encoding into another.
- For descriptions, we break the algorithm into stages, each usually involving many individual steps of the Turing machine's computation.
  - The first line of the algorithm describes the input to the machine.
  - If the input description is simply w, the input is taken to be a string.
  - If the input is an encoding $\langle A \rangle$ of an object A, the Turing machine first implicitly tests whether the encoding is valid and rejects if it is not.

Connected Graphs

- Let A be the language consisting of all strings representing undirected graphs that are connected.
- A graph is **connected** if every node can be reached from every other node by traveling along the edges of the graph.
- We write

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

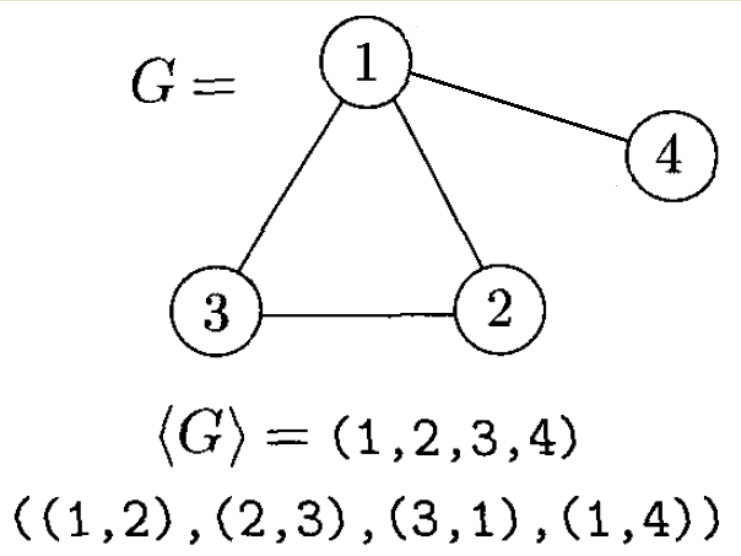
- The following is a high-level description of a TM M that decides A.

  M: On input $\langle G \rangle$, the encoding of a graph G,

  1. Select the first node of G and mark it.
  2. Repeat the following stage until no new nodes are marked:
     3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
  4. Scan all the nodes of G to determine whether they all are marked. If they are, accept; otherwise, reject.

Implementation-Level Details: An Encoding

- $\langle G \rangle$ encodes the graph G as a string. It may be a list of the nodes of G followed by a list of the edges of G.
  - Each node is a decimal number.
  - Each edge is the pair of decimal numbers that represent the nodes at the two endpoints of the edge.



Implementation-Level Details: Verifying Validity of Input

- When M receives the input $\langle G \rangle$, it first checks to determine whether the input is the proper encoding of some graph: M scans the tape to be sure that there are two lists and that they are in the proper form.
  - The first list should be a list of distinct decimal numbers.
  - The second should be a list of pairs of decimal numbers.
- Then M checks several things.
  - First, the node list should contain no repetitions.
  - Second, every node appearing on the edge list should also appear on the node list.
- If the input passes these checks, it is the encoding of some graph G. This verification completes the input check, and M goes on to Stage 1.

Implementation-Level Details: Algorithm

- For Stage 1, M marks the first node with a dot on the leftmost digit.
- For Stage 2, M scans the list of nodes to find an undotted node n_1 and flags it by marking it differently - say, by underlining the first symbol.
  Then M scans the list again to find a dotted node n_2 and underlines it, too. Now M scans the list of edges. For each edge, M tests whether the two underlined nodes n_1 and n_2 are the ones appearing in that edge.
  If they are, M dots n_1, removes the underlines, and goes on from the beginning of Stage 2.
  If they are not, M checks the next edge on the list. If there are no more edges, $\{n_1, n_2\}$ is not an edge of G.
  Then M moves the underline on n_2 to the next dotted node and now calls this node n_2.

Implementation-Level Details: Algorithm (Cont'd)

- Continuing with Stage 2:
  It repeats the steps in this paragraph to check, as before, whether the new pair $\{n_1, n_2\}$ is an edge. If there are no more dotted nodes, n_1 is not attached to any dotted nodes.
  Then M sets the underlines so that n_1 is the next undotted node and n_2 is the first dotted node and repeats the steps in this paragraph. If there are no more undotted nodes, M has not been able to find any new nodes to dot, so it moves on to Stage 4.
- For Stage 4, M scans the list of nodes to determine whether all are dotted.
  If they are, it enters the accept state.
  Otherwise it enters the reject state.