Turing Machines

- A Turing machine is similar to a finite automaton with supply of unlimited memory.
- A Turing machine can do everything that any computing device can do.
- There exist problems that even a Turing machine cannot solve.

Turing Machines

- A Turing machine is similar to a finite automaton with supply of unlimited memory.
- A Turing machine can do everything that any computing device can do.
- There exist problems that even a Turing machine cannot solve.

TM versus FA, PDA

1. **write**: A TM tape allows both write and read operations. DFA and NFA have only an input tape which is read-only and the tape head moves from left to right. PDA has both an input tape and a stack tape. We can write and read on the stack tape: when the head moves right, it writes; when the head moves left, it erases the current symbol.
2. **size**: The tape of a TM is infinite; the input tape of FA and PDA is finite; the stack tape of PDA is infinite.
3. **accept**: FA and PAD accept a string when it has scanned all the input symbols and enters a final state; TM accepts a string as long as it enters a final state (one suffices).

Tape of a Turing Machine (TM)

- Memory is modeled by a *tape* of symbols.
- Initially the tape contains only the input string and is blank everywhere else.
- If a TM needs to store info, it may write on the tape.
- To read the info that it has written, TM can move its head back.
- TM continues to move until it enters a state whose next move is not defined.
Example TM computation

Construct a TM $M_1$ that tests the membership in the language $L_1 = \{w\#w \mid w \in \{0, 1\}^*\}$.

- $s_0$: if first symbol is 0 or 1, replace it by $x$, remember it as $a$; if it is $\#$, goto $s_5$; else reject;
- $s_1(a)$: move right until $\#$; if no $\#$ before $\#$, reject;
- $s_2(a)$: move right until 0 or 1; if the current symbol is the same as $a$, then replace it by $x$; else reject;
- $s_3$: move left until $\#$;
- $s_4$: move left until $x$ and goto $s_0$;
- $s_5$: move right until 0, 1, or $\#$; accept if $\#$; reject if 0, 1.

Example TM computation

Construct a TM $M_1$ that tests the membership in the language $L_1 = \{w\#w \mid w \in \{0, 1\}^*\}$.

In other words: we want to design $M_1$ such that $M_1(w) = \text{accept iff } w \in L_1$.

Computations

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ computes as follows:

- $M$ receives as input $w = a_1a_2 \ldots a_n \in \Sigma^*$, $a_i \in \Sigma$, written on the leftmost squares of the tape and the rest of the tape is blank (i.e., filled with $\#$)
- The head starts on the leftmost square of the tape.
- The first blank encountered shows the end of the input.
- Once it starts, it proceeds by the rules defined by $\delta$.
- If $M$ ever tries to move to the left of the leftmost square the head stays in the leftmost square even though $\delta$ indicates $L$.
- The computation continues until $M$ cannot move; if $M$ enters $q_{\text{accept}}$, the input string is accepted. $M$ may go on forever as long as $\delta$ is defined.

Formal definition

A Turing machine is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where $Q, \Sigma, \Gamma$ are finite sets and

1. $Q$ is a set of states
2. $\Sigma$ is the input alphabet and $\#$ $\not\in \Sigma$
3. $\Gamma$ is the tape alphabet, $\#$ $\in \Gamma, \Sigma \subset \Gamma$
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function
5. $q_0 \in Q$ is the initial state
6. $q_{\text{accept}} \in Q$ is the accept state (sometimes denoted $q_a$)
7. $q_{\text{reject}} \in Q$ is the reject state (sometimes denoted $q_r$)

Note: $q_{\text{reject}}$ is optional in the definition.
Example TM computation

- $s_3$: move left until $#$;
  $\delta(s_3, a) = \langle s_3, a, L \rangle$, where $a \in \{0, 1\}$,
  $\delta(s_3, #) = \langle s_4, #, L \rangle$

- $s_4$: move left until $x$ and goto $s_0$;
  $\delta(s_4, a) = \langle s_4, a, L \rangle$, where $a \in \{0, 1\}$,
  $\delta(s_4, x) = \langle s_0, x, R \rangle$

- $s_5$: move right until 0, 1, or $\sqcup$; accept if $\sqcup$; reject if 0, 1.
  $\delta(s_5, x) = \langle s_5, x, R \rangle$, $\delta(s_5, \sqcup) = \langle s_a, \sqcup, L \rangle$

Formalizing TM computation

A configuration $C_1$ yields a configuration $C_2$ if the TM can legally go from $C_1$ to $C_2$ in a single computation step

Formally: suppose $a, b, c \in \Gamma$, $u, v \in \Gamma^*$ and $q_i, q_j \in Q$.

1. We say that $ua \; q_i \; bv \; yields \; uac \; q_j \; v$ if
   $\delta(q_i, b) = \langle q_j, c, R \rangle$; (machine moves rightward)

2. We say that $ua \; q_i \; bv \; yields \; u \; q_j \; acv$ if
   $\delta(q_i, b) = \langle q_j, c, L \rangle$; (machine moves leftward)

Configuration

A configuration $C$ of $M$ is a tuple $C = \langle u, q, v \rangle$, where $q \in Q$,
$uv \in \Gamma^*$ is the tape content and the head is pointing to the first symbol of $v$.

- Configurations are used to formalize machine computation and therefore are represented by special symbols.
- Tape contains only $\sqcup$ following the last symbol of $v$. 
Example 2

\( M_2 \) is a Turing machine that decides \( A = \{ 0^{2^n} \mid n \geq 0 \} \).

Some elementary

\( M_2 = \text{"On input string } w \text{"} \)

1. Sweep left to right across the tape, crossing off every other 0; if the number of 0 is odd, reject;
2. If in stage 1 the tape contained a signle 0, accept.
3. Return the head to the left end of the tape.
4. Go to stage 1.

Special configurations

- If the input of \( M \) is \( w \) and initial state is \( q_0 \) then \( q_0 \ w \) is the start configuration.
- \( u a \ q_\text{accept} \ b v \) is called accepting configuration.
- \( u a \ q_\text{reject} \ b v \) is called rejecting configuration.
- \( u a \ q c v \), where \( \delta(q, c) \) is undefined, is also called rejecting configuration.
- Accepting and rejecting configurations are also called halting configurations

Example 2

1. Sweep left to right across the tape, crossing off every other 0; if the number of 0 is odd, reject;
   (a) Mark the first 0 by \( A \): \( \delta(q_1, 0) = (q_2, A, R) \)
   (b) Cross off the next 0 after \( A \): \( \delta(q_2, 0) = (q_3, x, R) \)
   (c) Pass 0 at odd position; cross off 0 at even position.
      \( \delta(q_3, 0) = (q_4, 0, R), \delta(q_4, 0) = (q_3, x, R), \)
   (d) \( x \) is invisible to \( \{ q_2, q_3, q_4 \} : \delta(q, x) = (q, x, R) \) for \( q \in \{ q_2, q_3, q_4 \} \).
   (e) If the number of 0 is odd, reject: \( \delta(q_4, \emptyset) = (q_r, \emptyset, R) \)
2. If in stage 1 the tape contained a signle 0, accept: \( \delta(q_2, \emptyset) = (q_a, \emptyset, R) \)
3. Return the head to the left end of the tape: \( \delta(q_3, \emptyset) = (q_b, \emptyset, L), \delta(q_5, a) = (q_5, a, L) \) for \( a \in \{ 0, x \}. \)
4. Go to stage 1 (b): \( \delta(q_5, A) = (q_2, A, R). \)
Language of $M$

\[ L(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \} \]

Turing-recognizable language: A language $L$ is Turing-recognizable if there is a Turing machine $M$ that recognizes it.

Accepting an input $w$

A Turing machine $M$ accepts the input $w$ if a sequence of configurations $C_1, C_2, \ldots, C_n$ exists such that:

1. $C_1$ is the start configuration, $C_1 = (\epsilon, q_0, w)$
2. Each $C_i$ yields $C_{i+1}$, $i = 1, 2, \ldots, n-1$
3. $C_n$ is an accepting configuration.

Fail to accept

- TM fails to accept $w$ by entering $q_{reject}$ and thus rejecting, or by looping
- Sometimes it is difficult to distinguish a machine that fail to accept from one that merely takes long-time to halt.

Note

When we start a TM on an input $w$ three cases can happen:

1. TM may accept $w$
2. TM may reject $w$
3. TM may loop indefinitely, i.e., TM does not halt.

Note: looping does not mean that machine repeats the same steps over and over again; looping may entail any simple or complex behavior that never leads to a halting state.

Question: is this real? I.e., can you indicate a computation that takes infinite many steps without repetition?
Turing-decidability

- A decider that recognizes some language is also said to *decide* that language.
- A language is called *Turing-decidable* or simply *decidable* if some TM decides it.

Decider

- A TM that halts on all inputs is called a decider.

Higher Level Descriptions

- We can give a formal description of a particular TM by specifying each of its seven components.
- Defining $\delta$ can become cumbersome. To avoid this we use higher level descriptions which are precise enough for the purpose of understanding.
- We want be sure that every higher level description is actually just a short hand for its formal counterpart.

Note

- Any regular language is Turing-decidable.
- Any context-free language is Turing-decidable.
- Every decidable language is Turing-recognizable (a language is Turing-recognizable if it is recognized by a TM).
- Certain Turing-recognizable languages are not decidable (to be decidable means to be recognized by a TM which halts on all inputs).
Analyzing $M_3$

- In stage 1 $M_3$ operates as a finite automaton; no writing is necessary as the head moves from left to right:
  1. $\delta(q_0, \sqcup) = (q_a, \sqcup, R)$, $\delta(q_0, b) = (q_4, b, R)$
  2. $\delta(q_0, a) = (q_1, A, R), \delta(q_1, b) = (q_2, b, R)$, $\delta(q_1, \sqcup) = (q_a, \sqcup, R)$
  3. $\delta(q_2, b) = (q_2, b, R), \delta(q_2, c) = (q_3, c, R)$
  4. $\delta(q_3, c) = (q_3, c, R)$
  5. $\delta(q_4, b) = (q_4, b, R), \delta(q_4, \sqcup) = (q_a, \sqcup, R)$

Example 3

- $M_3$ is a Turing machine that performs some elementary arithmetic. It decides the language $C = \{a^i b^j c^k \mid i \times j = k, i, j, k \geq 0\}$

  - On input string $w$
    1. Scan the input from left to right to be sure that it is a member of $a^*b^*c^*$; reject if it is not; accept if it is $\epsilon, a^+$ or $b^+$.
    2. Set the head pointing at the first $a$ on the tape
    3. Cross off an $a$ and scan to the right until a $b$ occurs. Shuttle between the $b$’s and $c$’s crossing off one of each until all $b$’s are gone
    4. Restores the crossed off $b$’s and repeat stage 2 if there is another $a$ to cross off. If all $a$’s are crossed off, check on whether all $c$’s are crossed off. If yes accept, otherwise reject.

Stage 3 cross $a$

- $\delta(q_5, A) = (q_6, \sqcup, R)$
- $\delta(q_6, a) = (q_7, A, R)$
- $\delta(q_6, b) = (q_8, b, L), \delta(q_8, \sqcup) = (q_7, E, R)$
- $\delta(q_7, a) = (q_7, a, R), \delta(q_7, B) = (q_7, B, R), \delta(q_7, b) = (q_9, B, R)$
- $\delta(q_8, b) = (q_9, b, R), \delta(q_9, C) = (q_9, C, R), \delta(q_9, c) = (q_{10}, C, L)$

Stage 2 finding the first $a$

- $\delta(q_3, \sqcup) = (q_5, \sqcup, L)$
- $\delta(q_5, x) = (q_5, x, L)$ for $x \in \{a, b, c\}$
Element distinctness problem

Given a list of strings over \{0, 1\} separated by \#$\$, determine if all strings are different.

A TM that solves this problem accepts the language

\[ E = \{ \#x_1\#x_2\#\ldots\#x_k \mid x_i \in \{0, 1\}^*, x_i \neq x_j \text{ for } i \neq j \} \]

Example 4

\[ M_4 = (Q, \Sigma, \Gamma, \delta, q_s, q_0, q_f) \text{ is the TM that solves the element distinctness problem} \]

\[ M_4 \text{ works by comparing } x_1 \text{ with } x_2, \ldots, x_k, \text{ then by comparing } x_2 \text{ with } x_3, \ldots, x_k, \text{ and so on} \]

Marking tape symbols

- In stage two the machine places a mark above a symbol, \#$\$ in this case.
- In the actual implementation the machine has two different symbols, \#$\$ and \$\$ in the tape alphabet \(\Sigma\).
- Thus, when machine places a mark above symbol \(x\) it actually write the marked symbol of \(x\) at that location.
- Removing the mark means write the symbol at the location where the marked symbol was.
- Assumption: all symbols of the tape alphabet have marked versions too.

Informal description

\[ M_4 = "\text{On input } w:} \]

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 \(x_k\) was present, so accept.
3. By zig-zagging, compare the two strings to the right and to the left of the marked \#. 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 strings have been compared, so accept.
5. Go to stage 3"
Standardizing our model

Question: *What is the right level of detail to give when describing a Turing machine algorithm?*

*Note:* This is a common question asked especially when preparing solutions to various problems such as exercises and problems given in assignments and exams during the process of learning Theory of Computation.

---

High level operations of TMs

- Compare if two strings are the same or not.
- Compute the addition, substraction, multiplication, division, power, log, etc. of numbers in unary form.
- Shift a string right (or left).
- Maintain a base-$b$ counter.

---

Equivalence of TMs

To show that two models of TM are equivalent we need to show that we can simulate one by another.

---

Answer

The three possibilities are:

1. *Formal description:* spells out in full all 7 components of a Turing machine. This is the lowest, most detailed level of description.
2. *Implementation description:* use English prose to describe the way Turing machine moves its head and the way it stores data on its tape. No details of state transitions are given.
3. *High-level description:* use English prose to describe the algorithm, ignoring the implementation model. No need to mention how machine manages its head and tape.
Variants of Turing Machine

The transition function of a standard TM in our definition forces the head to move to the left or right after each step. Let us vary the type of transition function permitted.

Suppose that we allow the head to stay put, i.e.,

\[ \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\} \]

A transition can be represented by two standard transitions: one that move to the left followed by one that moves to the right. Since we can convert a TM which stay put into one that doesn't have this facility, the extension does not increase its power.

Multitape Turing Machines

A multitape TM is like a standard TM with several tapes. Each tape has its own head for reading/writing. Initially, the input is on tape 1 and other tapes are blank.

Transition function allows for reading, writing, and moving the heads on all tapes simultaneously, i.e.,

\[ \delta : q_i \times a_1, \ldots, a_k \rightarrow q_j \times b_1, \ldots, b_k, L, R, \ldots, L \]

Multi-tape TM: An Example

Let

\[ L = \{ 0^{a_1} b_2 c \mid c = \lfloor \log a b \rfloor, a > 1, b > 0 \} \]

The second tape containing \( x \) and the third tape containing \( k \), where \( x = a^k+1 \), based on the following algorithm:

\[
\text{Return } k
\]

while \( x \leq b \)

{x := x * a; k := k + 1;}

return k.

Multi-tape TMs are more powerful than standard TMs. They allow for faster computation, especially when the input is on one tape and the working space is on the other tapes.

Some problems, such as the Halting Problem, can be solved more efficiently on a multitape TM. For example, the Halting Problem can be solved in polynomial time on a multitape TM, but it is undecidable on a standard TM.

A multitape TM is like a standard TM with several tapes, each of which can be accessed independently. This allows for more efficient computation, especially when the input is on one tape and the working space is on the other tapes.

However, multitape TMs are not more powerful than standard TMs. They can solve the same problems as standard TMs, but they may do so more efficiently. For example, the Halting Problem can be solved in polynomial time on a multitape TM, but it is undecidable on a standard TM.
**Theorem 3.13**

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

**Proof:** We show how to convert a multitape TM $M$ into a single tape TM $S$. The key idea is to show how to simulate $M$ with $S$.

---

**Simulating $M$ with $S$**

Assume that $M$ has $k$ tapes

- $S$ simulates the effect of $k$ tapes by storing their information on its single tape
- $S$ uses a new symbol $\#$ as a delimiter to separate the contents of different tapes
- $S$ keeps track of the location of the heads by marking with a $\bullet$ the symbols where the heads would be.

---

**General Construction**

$S = "$On input $w = a_1a_2\ldots a_n$

1. Put $S(tape)$ in the format that represents $M(tapes)$:
   
   $S(tape) = \# a_1\ldots a_n\# \ldots \# \ldots \#$

2. Scan the tape from the first $\#$ (which represent the left end) to the $(k+1)$-st $\#$ (which represent the right-hand end) to determine the symbols under the virtual heads. Then $S$ makes the second pass over the tape to update it according to the way $M$’s transition function dictates.

3. If at any time $S$ moves one of the virtual heads to the right of $\#$ it means that $M$ has moved on the corresponding tape onto the unread blank portion of that tape. So, $S$ writes a $\sqcup$ on this tape cell and shifts the tape contents from this cell until the rightmost $\#$, one unit to the right. Then it continues to simulates as before".

---

**Example simulation**

Figure 1 shows how to represent a machine with 3 tapes by a machine with one tape.

![Diagram](image-url)

Figure 1: Multitape machine simulation
Corollary 3.15

A language is Turing recognizable iff some multitape Turing machine recognizes it.

**Proof:**
- *if:* a Turing recognizable language is recognized by an ordinary Turing machine, which is a special case of a multitape Turing machine.
- *only if:* follows from the equivalence of a Turing multitape machine $M$ with the Turing machine $S$ that simulates it. That is, if $L$ is recognized by $M$ then $L$ is also recognized by $S$.

Nondeterministic TM

- A NTM is defined in the expected way: at any point in a computation the machine may proceed according to several possibilities.
- Formally, $\delta : Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\})$.
- Computation performed by a NTM is a tree whose branches correspond to different possibilities for the machine.
- If some branch of the computation tree leads to the *accept* state, the machine accepts the input.

Theorem 3.16

Every nondeterministic Turing machine, NTM, has an equivalent deterministic Turing machine, DTM.

**Proof idea:** show that we can simulate a NTM $N$ with DTM, $D$.

**Note:** in this simulation $D$ tries all possible branches of $N$'s computation. If $D$ ever finds the accept state on one of these branches then it accepts. Otherwise $D$ simulation will not terminate.

More on NTM simulation

- $N$’s computation on an input $w$ is a tree, $N(w)$.
- Each branch of $N(w)$ represents one of the branches of the nondeterminism.
- Each node of $N(w)$ is a configuration of $N$.
- The root of $N(w)$ is the start configuration.

**Note:** $D$ searches $N(w)$ for an accepting configuration.
A better idea

Design $D$ to explore the tree by using a breadth-first search

This strategy explores all branches at the same depth before going to explore any branch at the next depth. Hence, this method guarantees that $D$ will visit every node of $N(w)$ until it encounters an accepting configuration.

A tempting bad idea

Design $D$ to explore $N(w)$ by a depth-first search

A depth-first search goes all the way down on one branch before backing up to explore next branch. Hence, $D$ could go forever down on an infinite branch and miss an accepting configuration on an other branch.

Deterministic simulation of NTM

$D$ has three tapes:
1. Tape 1 always contains the input (and the code of $N$) and is never altered.
2. Tape 2 (called simulation tape) maintains a copy of the $N$’s tape on some branch of its nondeterministic computation.
3. Tape 3 (called address tape) keeps track of $D$’s location in $N$’s nondeterministic computation tree.

Formal proof

Figure 2: Deterministic TM $D$ simulating $N$
Note

- Each symbol in a node address 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 doesn't correspond to any node.
- Tape 3 contains a string over \( \Sigma_b \) which represents a branch of \( N \)'s computation from the root to the node addressed by that string, unless the address is invalid.
- The empty string \( \epsilon \) is the address of the root.

Address tape

- Every node in \( N(w) \) can have at most \( b \) children, where \( b \) is the size of the largest set of possible choices given by \( N \)'s transition function.
- Hence, to every node we assign an address that is a string in the alphabet \( \Sigma_b = \{1, 2, \ldots, b\} \).
- Example: we assign the address 231 to the node reached by starting at the root, going to its second child, and then going to that node's third child and then going to that node's first child.

Corollary 3.18

A language is Turing-recognizable iff some nondeterministic TM recognizes it.

Proof:
- if: any deterministic TM is automatically an nondeterministic TM
- only if: follow from the fact that any NTM can be simulated by a DTM.

The description of \( D \)

1. Initially tape 1 contains \( w \) and tape 2 and 3 are empty.
2. Copy tape 1 over 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.
   - If a rejecting configuration is reached go to stage 4; if an accepting configuration is encountered, accept the input.
4. Replace the string on tape 3 with the next string as if it is a counter and go to stage 2.
**Enumerators**

- An enumerator is a variant of a TM with an attached printer (or an output tape)
- The enumerator uses the printer as an output device to print strings
- Every time the TM wants to add a string to the list of recognized strings it sends it to the printer

**Note:** Some people use the term recursively enumerable language for languages recognized by enumerators

---

**Corollary 3.19**

A language is decidable iff some NTM decides it.

---

**Theorem 3.21**

A language is Turing-recognizable iff some enumerator enumerates it

**Proof:**

- if: if we have an enumerator $E$ that enumerates a language $A$ then a TM $M$ recognizes $A$. $M$ works as follows:
  
  $M = \"$On input $w:\$"
  
  1. Run $E$. Every time $E$ outputs a string $x$, compare it with $w$.
  2. If $w = x$, $M$ accept; else go to 1.

  Clearly $M$ accepts those strings that appear on $E$’s list.

---

**Computation of an Enumerator**

- An enumerator starts with a blank input tape
- If the enumerator does not halt it may print an infinite list of strings
- The language recognized by the enumerator is the collection of strings that it eventually prints out.

**Note:** an enumerator may generate the strings of the language it recognizes in any order, possibly with repetitions.
Another Proof

only if: If $M$ recognizes a language $A$, we can construct an enumerator $E$ for $A$. For that consider $s_1, s_2, \ldots$, the list of all possible strings in $\Sigma^*$, where $\Sigma$ is the alphabet of $M$. Given a pair of integers $\langle i, j \rangle$, define $Next(\langle i, j \rangle) =$ if $i = 1$ then $\langle j + 1, 1 \rangle$ else $\langle i - 1, j + 1 \rangle$.

$E =$
1. Let $\langle i, j \rangle = \langle 1, 1 \rangle$.
2. Simulate $M$ on $s_j$ at most $i$ steps.
3. If $M$ accepts $s_j$ with exactly $i$ steps, prints out $s_j$.
4. $\langle i, j \rangle = Next(\langle i, j \rangle)$, go to 2.

Note: no string prints out more than once.

Proof, continuation

only if: If $M$ recognizes a language $A$, we can construct an enumerator $E$ for $A$. For that consider $s_1, s_2, \ldots$, the list of all possible strings in $\Sigma^*$, where $\Sigma$ is the alphabet of $M$.

$E =$
1. Let $i = 1$.
2. For $j = 1$ to $i$, simulate $M$ on $s_j$ at most $i$ steps.
3. If $M$ accepts $s_j$ with $i$ steps or less, prints out $s_j$.
4. $i = i + 1$, go to 2.

Note: a string may print out multiple times.

Algorithms

Informally speaking an algorithm is a collection of simple instructions for carrying out a task.
In everyday life algorithms are called procedures or recipes.
Algorithms abound in contemporary mathematics.

Equivalence with other models

There are many other models of general purpose computation. Example: recursive functions, normal algorithms, semi-Thue systems, $\lambda$-calculus, etc.
Some of these models are very much like Turing machines; other are quite different
All share the essential feature of a TM: unrestricted access to unlimited memory
All these models turn out to be equivalent in computation power with TM
Church-Turing Thesis

- Other formal definitions of algorithms have been provided by: Kleene using recursive functions, Markov using rewriting (derivation) rules with a grammar called normal algorithms.
- Essentially all these formal concepts of algorithm are equivalent among them and are equivalent with Turing Machines.
- **Church-Turing Thesis**: Every computing device can be simulated by a Turing machine.

Algorithm as a Turing Machine

- Alonzo Church and Alan Turing in 1936 came with formal definitions for the concept of algorithm.
- Church used a notational system called \( \lambda \)-calculus to define algorithms.
- Turing used his “Turing Machines” to define algorithms.
- These two definitions were shown to be equivalent.

Encoding and Decoding Objects

- Our notation for encoding an object \( O \) into its string representation is \( \langle O \rangle \).
- If we have several objects \( O_1, O_2, \ldots, O_k \) we denote their encoding into a string by \( \langle O_1, O_2, \ldots, O_k \rangle \).
- Encoding itself can be done in many ways. It doesn’t matter which encoding we pick because a Turing machine can always translate one encoding into another.
- A (part of) Turing machine may be programmed to decode the input representation so that it can be interpreted the way we intend.

Formal Definition

- **Definition**: an algorithm is a decider TM in the standard representation.
- The input to a Turing machine is always a string.
- If we want an object, other than a string as input, we must first represent that object as a string.
- Strings can easily represent polynomials, graphs, grammars, automata, and any combination of these objects.
A TM deciding $A$

$M = \text{“On input } \langle G \rangle, \text{ the encoding of } G$\n
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."

Example TM

- Let $A$ be the language consisting of all strings representing undirected graphs that are connected.
- Recall: a graph is connected if every node can be reached from every other node.
- Notation: $A = \{ \langle G \rangle \mid G \text{ is a connected undirected graph} \}$

Graph encoding, $\langle G \rangle$

- The encoding $\langle G \rangle$ of a graph as a string is a list of nodes followed by a list of edges.
- Each node is a decimal number, and each edge is a pair of decimal numbers that represent the nodes that edge connects.
- Example encoding: the graph in Figure 3 is encoded by the string: $\langle G \rangle = (1, 2, 3)((1, 2), (2, 3), (3, 1), (1, 4))$.

Implementation details

Consider the graph:

![Figure 3: A connected graph](image-url)
Checking the encoding

When $M$ receives the input $\langle G \rangle$ it first checks to determine that the input is a proper encoding of some graph:

1. Scan the tape to be sure that there are two lists and that they are in proper form.
2. The first list should be a list of distinct decimal numbers; the second list should be a list of pairs of decimal numbers.
3. The list of decimal numbers should contain no repetitions.
4. Every node on the second list should appear in the first list too.

Note: element distinctness problem can be used to formate the lists and to implement the checks above.