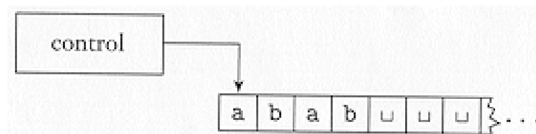
Chapter 4
The Church-Turing Thesis

We begin by studying a much more powerful automata: the *Turing Machine,* which has a unlimited and unrestricted memory. A Turing machine can do anything a real computer can. Thus, it is a “more accurate” model of computation. Anything it cannot solve is regarded beyond the theoretical limits of computation.

The Turing machine model uses an infinite tape as its memory and has a tape head that can read and write symbols and move around on the tape.

Initially, the tape contains only the input string and is blank everywhere else, if the machine needs to store information, it may write this information on the tape. To read the information that it once wrote on the tape, the machine can move its head back over the tape.

The machine keeps on working until it feels that it has some output to send out. Then, it enters designated, accepting and/or rejecting states, to accept and/or reject its input. If it doesn't either accept or reject a string, it may go on forever.



An example

Let $A = \{w\#w|w \in \{0,1\}\}$. we want to design a TM to accept its input if it belongs to A. Below is a high level description of such a machine, M_1.

For a given string, s,
1. Scan the input to be sure that it contains a single # symbol. If not, reject.

2. Move across the tape to corresponding positions on either side of the # symbol to check on whether they match. If they don't, reject the string; otherwise, cross off those two symbols and continue.

3. When all symbols to the left of # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, reject; otherwise, accept the input.

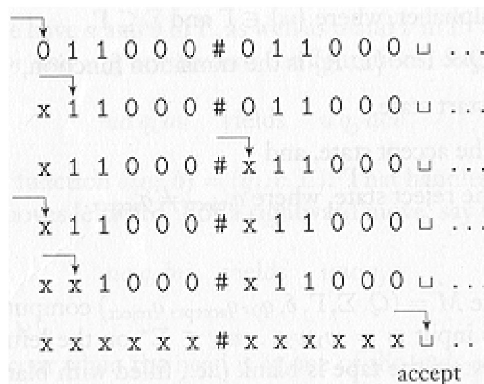
How does M_1 work?

The tape head begins over the leftmost symbol of the input. M_1 will record this symbol and crosses it over by putting an x in its place. Then the head moves to the right until it sees the # and move one symbol further. If it sees a blank before it sees the #, the input is not in the correct format, thus will be rejected.

If the pair of symbols does not match, M_1 rejects the input. Otherwise, M_1 crosses it off by putting in a y and moves the head back to the left until it comes to an x. Now, the head moves to the right again to try to match the corresponding symbols.

If at any point the symbol recorded in the control is a #, all the symbols to the left of # were successfully matched. Then, M_1 will move over any x's to the right of #, and all the ys.

If the first symbol after the last y is either 0 or 1, the input string gets rejected; otherwise, if it is a blank, M_1 accepts the input by entering the accepted state. Below shows how the input 011000#011000 gets processed.



The formal definition

As always, the heart of the formal definition of a TM is the transition function. It takes the form: $Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$. Particularly, $\delta(q, a) = (r, b, L)$ means that if the machine is at state q, and the input symbol is a, then it replaces a with b, enters a new state r, and moves one position to the left.

Definition: 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 states, particularly,
1. Σ is the input alphabet not containing the special blank symbol, $\sqcup$.
2. Γ is the tape alphabet containing $\sqcup$ and everything that occurs in the tape.

An example

Let $L = \{a^n b^n | n \geq 1\}$, we design a Turing machine that accepts it. We firstly give out an algorithm.

For a given string, $s \in \{a, b\}^*$.
1. If the string starts with a b, rejects.

2. For every a, move across the tape to look for a matching b. If we could not find such a b, reject.

3. When all a's are matched, check for any remaining b. If any such b remains, reject; otherwise, accept the input.

An implementation

Let $\mathcal{M}_L = (Q, \Sigma, \Gamma, \delta, q_0, \{q_4\}, \{q_5\})$, where $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, x, y, B\}$, and the δ function is given as follows:

$$
\begin{aligned}
1.\ \delta(q_0, a) &= (q_1, x, R) \\
2.\ \delta(q_0, b) &= (q_5, b, R) \\
3.\ \delta(q_0, y) &= (q_3, y, R) \\
4.\ \delta(q_1, a) &= (q_1, a, R) \\
5.\ \delta(q_1, b) &= (q_2, y, L) \\
6.\ \delta(q_1, y) &= (q_1, y, R) \\
7.\ \delta(q_1, B) &= (q_5, B, L) \\
8.\ \delta(q_2, a) &= (q_2, a, L) \\
9.\ \delta(q_2, x) &= (q_0, x, R) \\
10.\ \delta(q_2, y) &= (q_2, y, L) \\
11.\ \delta(q_3, y) &= (q_3, y, R) \\
12.\ \delta(q_3, b) &= (q_5, b, R) \\
13.\ \delta(q_3, B) &= (q_4, B, L)
\end{aligned}
$$

In the above example, in q_0 we try to match off another a. In q_1 we trace through the whole string, trying to find a b to match with the a; if we cannot find such a b, we abort the whole process by rejecting the string. In q_2, we have found the b we are looking for, so we go back to the beginning, trying to match off more a's. In q_3, we find out that all as have been matched, so we try to get to the end of the string to see if there are still b's there. If there is, the string is rejected; otherwise, it is accepted.

Homework: Run this TM on several input strings. Modify it so that 1) it will accept $L = \{a^n b^n | n \geq 0\}$. 2) it will restore the original string.

Question: Can you design one that accepts $L = \{a^n b^n c^n | n \geq 0\}$?

How does a TM work?

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 tape head starts on the leftmost square. As $\sqcup \notin \Sigma$, so the first $\sqcup$ marks the end of the input.(?)

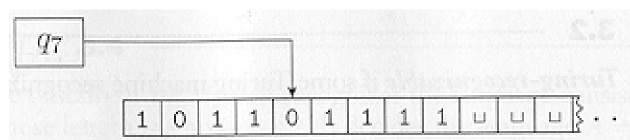
Once M starts, the computation proceeds according to δ. If M ever tries to move its head off the left end, the head stays in the same place for that move. The computation continues until it either enters the accept or the reject state, at which it *halts.* If neither occurs, M goes on forever.

Homework: Exercise 3.1.

Configuration

At any point, the *configuration* of M is represented as uqv, which means that M is in the state, q, and the current tape content is uv, and the tape head is located at the first symbol of v. The tape contains all blanks after v.

For example, $1011q_701111$ represents a configuration in which the tape contains a string 101101111, the current state is q_7, and the head is located at the second 0.



The *yield* relation

Let $a, b \in \Sigma$, and $u, v \in \Sigma^*$. We have the following:

1. The configuration uaq_ibv yields to uq_jacv, if $\delta(q_i, b) = (q_j, c, L)$.
2. The configuration uaq_ibv yields to $uacq_jv$, if $\delta(q_i, b) = (q_j, c, R)$.
3. The case that when the head is in the left end might be special: we have that q_ibv yields to q_icv even if the transition says to move to the left, but,
4. the case that when the head in the right end is not special, as $uaq_i \equiv uaq_i\sqcup$.

The *start configuration* for any TM on input w is q_0w. The state in the *accepting*(*rejecting*) configuration is $q_{\text{accept}}(q_{\text{reject}})$, respectively, and are both *halting* configurations. They don't yield to any other configurations.

The *accept* relation

A Turing Machine *accepts* an input s, 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 s,
2. each C_i yields to C_{i+1}, and
3. C_k is the accepting configuration.

The collection of strings that M accepts is *the language of* M, denoted $L(M)$. For example, $L(M_1)$ is the collection of all the strings over $\{0, 1\}$ such that each of them consists of two identical strings separated by a $\#$.

Homework: Exercise 3.5.

Enumerable vs. decidable

Definition: A language is *enumerable* if some Turing Machine accepts it.

Given an enumerable language, L, and a string, s, besides accepting or rejecting s, the corresponding TM might *loop.* As it is difficult to distinguish a machine that is looping from a one that just takes long time, we prefer those machines that halt at, i.e., either accept or reject, all input strings. Those machines are called *deciders.* A decider that accepts some language also is said to *decide* that language.

Definition: A language is *decidable* if some Turing Machine decides it.

Homework: Exercise 3.4.

An example

Let S be the collection of the names of all the people who committed some crimes before.

Theoretically, such a language is decidable, FBI's system might be the decider: Given a name of a suspect, n, it will look at all the records. If the name turns up, $n \in S$; otherwise, $n \notin S$.

Realistically, the language is not even acceptable. As for some reasons, the guy has never been caught, thus, his/her name has never been recorded into *any* machine. Thus, if the name shows up in some machine, $n \in S$; otherwise, we don't know.

We will show that some languages are *theoretically* undecidable.

Examples of TMs

Let's display a TM, M_2, that does some arithmetic, particularly, multiplication. The language is $\{a^i b^j c^k | i \times j = k, i, j, k \geq 1.\}$. M_2 works as follows:

1. Scan the input from left to right to be sure that the input is in the form of $a^* b^* c^*$ and reject it if it isn't.

2. Return the head to the left end.

3. Cross off an a and scan to the right until a b occurs. Shuttle between b's and c's, crossing off one of each until all b's are gone.

4. Restore the crossed off b's and repeat step 3 if there is another a to cross off. If there is no such a left, check on whether all c's have been crossed off. If yes, accept; otherwise, reject.

Variants of TMs

There are many variants of Turing Machines. An astonishing fact is that the original model and those "reasonable" variants all have the same power, i.e, they accept the same languages. To illustrate this *robustness*, we vary the type of transition rules allowed in a Turing Machine. In the original model, we only allow the machine to move either left or right, what happens if we also allow it to stay in the same place? Will this make a Turing Machine to accept more languages? No. For any transition rule to make the machine stay in the same place, we can replace it with two rules, one moves the head to the left, the other moves it back.

We will show the equivalence of the original model and several other models, by simulating each other.

Multitape TMs

A *multitape TM* is like an ordinary TM, with several tapes, each of which has its own tape head to read and write. Initially, the input is put on the first tape, while the other tapes are empty. The transition function is changed to allow for reading, writing, and moving the heads on all the tapes simultaneously. Formally, we have

$$\delta : Q \times \Gamma^k \;\mapsto\; Q \times \Gamma^k \times \{L, R\}^k,$$

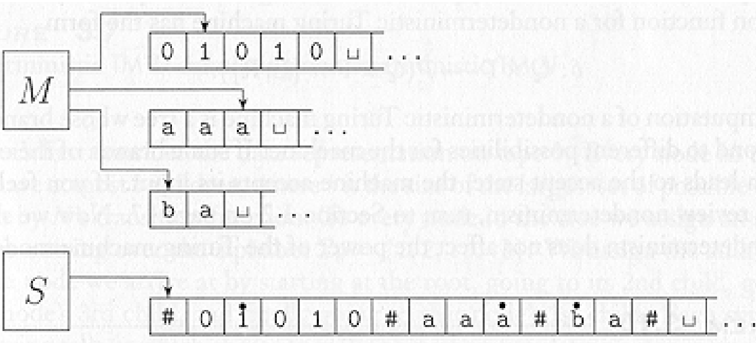
where k is the number of tapes.

The expression $\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$ means that, if the TM is in state q_i, and heads 1 through k are reading symbols $a_1, \ldots, a_k$, then it goes to state q_j, writes $b_1, \ldots, b_k$, and moves each head to either left or right, as specified.

Theorem: A language is enumerable iff some multitape TM accepts it.

Proof: As an ordinary TM is automatically a multitape TM, the only thing we need to show is that if a multitape TM, M, accepts a language, so does an ordinary one, S.

We show how to let S store all the information that M stores in its k tapes. We use $k+1$ #'s as delimiters to separate the respective contents and use corresponding "dotted" tape symbols to keep track of the locations of those heads.



The proof

Below is the description of S, the single tape TM:

1. First S uses its tape to represent all the information stored in M in the following format: $\# \dot{w}_1 \cdots \dot{w}_n \# \dot{B} \# \dot{B} \# \cdots \#$
2. To simulate a single move, S scans its tape from the first #, to the $k+1^{\text{st}}$ #, to determine the symbols which should be under the k heads. Then, S scans the tape again to update the tapes according to the various transition rules.
3. If at any point, S tries to move across a #, S has to write a ␣ on this tape cell and shifts all the tape contents, from this cell on, to its right by one cell. Then it continues as before. □

Non-deterministic TMs

A non-deterministic TM, at any point, may proceed according to several possibilities. Thus, the transition function, δ, has the following format:

$$\delta : Q \times \Gamma \;\mapsto\; \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

Hence, the computation of such a TM is a tree., in which each branch is a configuration. If any branch of such a tree accepts an input, the TM will accept the input.

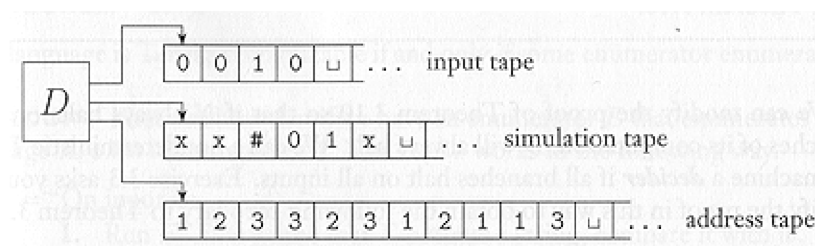
Theorem: A language is enumerable iff some non-deterministic Turing Machine accepts it.

Again, we only need to prove the other half of the result.

Beginning with the starting configuration, we will let D, the simulating TM, try all branches of the NTM. If D ever finds one branch that accepts the input, so does D. If all branches lead to rejection, D rejects. Otherwise, D will go on for ever.

The only way not to miss any possible halting outcome is to traverse the tree in *breadth first search.* The key is to use an *address tape* which tells the location of next node to be traversed. Such a tape contains strings over $\Sigma^b = \{1, 2, \ldots, \}$, where b is the size of the largest set of possible choices decided by δ. For example, 231 means that starting at the root, we will try the second child, then that node's third child, finally, goes to the latter's first child. An empty string represents a root.

Proof: D uses three tapes, as shown below. Tape 1 always contains the input, Tape 2 maintains a copy of N's tape on some branch of its non-deterministic computation. Tape 3 keeps track of D's location in N's non-deterministic tree.



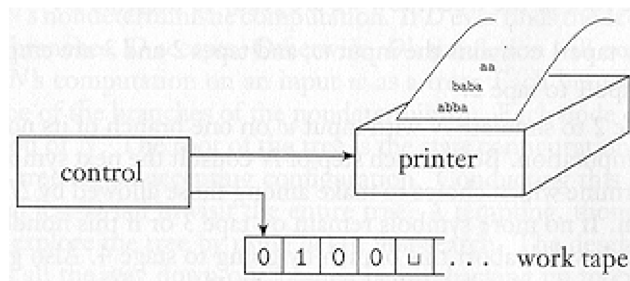
The deterministic TM D works as follows:

1. Initially Tape 1 contains the input w, and both Tape 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 non-deterministic computation, by using the address stored in Tape 3. If Tape 3 is empty, or the choice is invalid, abort this step and goes to Step 4. If this branch rejects the input, goes to Step 4, as well.
4. Replace the string in Tape 3, with the lexicographically next string. Simulate the next branch of N's computation by going back to Step 2.

Thus, D accepts the same language as that of N. By the previous result, there is also a single tape TM that accepts the original language. $\square$

Enumerators

Roughly speaking, an *enumerator* is a Turing Machine that is attached to a printer. Every time the TM accepts a string, it can print it out. The language accepted by an enumerator is the collection of all the strings it ever prints out.



The concept of enumerable language is rather input oriented, while the concept of an enumerator is output oriented. However, this difference does not matter.

Theorem: A language is enumerable iff some enumerator enumerates it.

Proof: Assume a language is accepted by an enumerator, E, we construct the following TM, M, that works in the following way.

For any 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.

Thus, M accepts the same set of strings that E enumerates.

Now, assume that a TM M accepts a language, we construct the following enumerator, E. Let $s_1, \ldots, s_n, \ldots$ be a list of all the possible strings over Σ^*, E works in the following way:
1. Repeat the following for $i = 1, 2, 3, \ldots,$
2. Run M for i steps for $s_1, s_2, \ldots, s_i$.
3. If any computation accepts, print out the corresponding s_i. $\square$

Homework: Exercises 3.2 and 3.3.

More equivalences

Besides those presented so far, many other models for general computation have been presented. Some of them are very much like TMs, some others look quite different. All of them allow unrestricted and unlimited memory. Remarkably, all models with this feature turns out to be equivalent in power.

This result shows that although there are different ways to characterize the general computation model, the class of algorithms they are describing is unique and natural.

For example, although, we can use either Pascal or LISP to write programs and these two languages look quite different from each other. However, as we can write a compiler of Pascal in LISP and write an interpreter of LISP in Pascal, the programs in either language can be turned into its equivalent in the other language, those two languages are really equivalent.

The definition of algorithms

Informally, an *algorithm* is a collection of simple instructions for carrying out some task. This notion is frequently used in place of a procedure or a recipe.

For a long time, there has been no precise definition of algorithm. People only use it in a vague and intuitive sense. For example, if we know, intuitively, there exists a way to do something, we say something is “computable.” It is quite fine in this positive sense. The problem is how to use it in the negative sense. For example, when we try to prove something is not “computable”, how could we show there does not exist an algorithm? In these places, we need a precise mathematic definition of algorithm.

Herbert's tenth problem

In 1900, mathematician David Hilbert presented an address at the International Congress of Mathematicians in Paris, in which he identified 23 problems and posed them as a challenge for the coming century.

His tenth problem is to devise an algorithm that tests if a polynomial has an integral root. For example, given the following polynomial:

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

has a root at $x = 5, y = 3$, and $z = 0$.

In his address, he implicitly assumed that such an algorithm, or "process", must exist, the only problem is how to find it. Now, we know that no such algorithm exists. It is algorithmically unsolvable.

It is acceptable,

Let D be $\{p|p$ is a polynomial with an integral root $\}$. Then, Hilbert's tenth problem is simply asking if D is decidable. We show that it is enumerable, i.e., there is a Turing machine, M, that will accept it.

We begin to work with a simpler language D_1, which is defined as $\{p|p$ is a polynomial over x with an integral root $\}$. Below is the Turing machine that accepts it.

$M_1 =$ "The input is a polynomial p over the variable x.

1. Evaluate p with x set successively to the values 0, 1, -1, ...

2. If at any point the polynomial evaluates to 0, accept."

but not decidable.

M could be similarly constructed. Both M and M_1 are enumerators, or recognizers, but not deciders for the respective languages. However, M_1 can be converted to a decider, by calculating the bounds within which the roots of a *single variable* polynomial must lie, and evaluating the polynomial against only that bound. As a matter of fact, the involved range is

$$(-k\frac{c_{\text{max}}}{c_1}, k\frac{c_{\text{max}}}{c_1}),$$

where k is the number of terms, c_{max} is the coefficient with largest absolute value, and C_1 is the coefficient of the highest order term.

However, it has been shown that it is *impossible* to calculate such bounds for *multivariable* polynomial.

The Thesis

The definition of algorithm came in the 1936 papers of A. Church and A. Turing. Church used a notational system, called the λ-calculus to define algorithms. Turing used his "machines".

These two definitions were shown to be equivalent to each other, as well as to many other alternative definition of "computation". This connection between an informal notion and the precise definition has come to be called the *Church-Turing Thesis:* The informal notation of algorithms equals to that of Turing Machines algorithms.

What does it mean?

This Thesis can't be proved. It can only either be accepted or rejected. It proposes that we give the necessarily information notion of algorithms some precise meaning.

On the other hand, if we accept this thesis, then whenever we see some intuitive procedures, we agree that it actually characterizes a formal TM. However, all the necessary details must be able to be fit in, whenever needed.

Thus, from now on, we will focus on algorithms, rather on the detailed description of Turing machines.

Coding of inputs

The input to a TM is always a string. If we want to provide an object other than a string as an input, we must *encode* this object into a string.

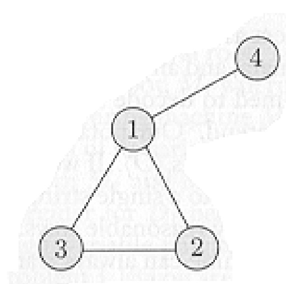
Strings can easily represent numbers, polynomials, graphs, grammars, automata, and any combination of those objects. A TM can be constructed to both encode an object into a string and decode it back so that it can be interpreted in the intended way.

Our notation for the encoding of an object, O, into a string is $< O >$. If we have several objects, $O_1, \ldots, O_k$, we denote their encoding into a single object is $< O_1, \ldots, O_k >$.

How to encode a graph?

A graph is typically represented as an ordered pair, whose first element is the collection of all the nodes, while the second element is the collection of all the edges, each of which is also a pair of nodes.

Hence, one way to encode a graph is to represent it as a list of its nodes, followed by another list, which represents the edges. For example, below is a graph.



The code for such a graph will be $< G >= (1,2,3,4)((1,2),(2,3),(3,1),(1,4))$.

Checking input

When M receives its input, it has to check the input to make sure that it represents a graph, i.e., the input consists of two lists. The first one should be a list of distinct numbers, and the second should be a list of pairs of numbers.

It also checks other things. For example, no repetition in the first list, and everything that occurs in the second list must occur in the first list.

An example

Let A be the language consisting of all strings representing graphs that are connected, i.e., between any two nodes, there is a path. We write

$$A = \{< G > | G \text{ is a connected graph.}\}.$$

The following is a TM, M, that decides A.

Given an input $< G >$, 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 its is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they are all marked, if they are, accept; otherwise reject.

Details of M

Below is the detailed description of M, which checks the connectedness of a graph, G.

1. M marks the first node, with a dot on the leftmost digit.
2. M scans the list of nodes to find an undotted one, n_1, and flags it, by underlining the first digit. If all the nodes are dotted, M accepts the input. Otherwise, M scans the list of nodes to find a dotted one, n_2, by underlining it, too.
3. For each edge, M checks if n_1 and n_2 are the nodes that occur in that edge. If they are, M dots n_1, removes underlining and goes back to Step 2. If they aren't and the edge list is not over, yet, M checks for the next edge; otherwise, (n_1, n_2) is not an edge in G.

Thus, M moves the underlining of n_2 to the next dotted node, and repeat this step. If there are no more dotted node left, n_1 is not connected to any dotted node. Thus, M removes the underlining of n_1, and underline the next undotted node and goes back to Step 2.

If there are no undotted nodes left, M has no new node to dot. Thus, it goes to Step 4.

4. If everything is dotted, accept; otherwise, M will reject the input.

Maybe it is time to see an example, then quit; or simply quit.