Chapter 4
Church-Turing Thesis

We begin by studying a much more powerful automata: the Turing Machine, which comes with a unlimited and unrestricted memory.

A Turing machine can do anything a real computer can, if we don’t pay attention to its efficiency. 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 both forward and back on the tape. Thus, its memory really functions like an infinite RAM.
How does it work?

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 and forth over the tape.

The machine keeps on working until it has some output to send out. Then, it enters designated, accepting and/or rejecting states, to either accept or reject its input.

If it doesn’t either accept or reject a string, it may go on forever. This is where the Hell breaks loose....
An example

Let $A = \{w\#w | w \in \{0, 1\}\}$, which is not Context free.

Let’s design a TM to accept $A$, by starting with 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 symbols on either side of the # symbol to check 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.
More details of $M_1$

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 after it sees the $\#$, the input is not in the correct format, thus will be rejected.

Otherwise, $M_1$ crosses it off by putting in a $x$ 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 next pair of corresponding symbols.

It will go over all the way to the last $x$. If there is nothing beyond this last $x$, it rejects the input, otherwise, it puts down another $x$ and comes back.
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 #.

If the first symbol after # and $x$’s to its right 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 $Q, \Sigma, \Gamma$ are all finite states, particularly,

1. $\Sigma$ is the input alphabet not containing the special blank symbol, $\sqcup$.

2. $\Gamma$ 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 start with 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 the \( a \)'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 $\delta$ function is given as follows:

1. $\delta(q_0, b) = (q_5, b, R)$
2. $\delta(q_0, a) = (q_1, x, R)$
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. $\delta(q_3, B) = (q_4, B, L)$

Question: What is going on?
What does each state mean?

In $q_0$ we try to match off another $a$.

In $q_1$, holding an $a$, 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 $q_3$, all the $a$’s 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.

If we come over to $q_4$, we accept the string, and reject it if we end up with $q_5$. 
Homework

1. Use JFlap 7.0 to set up $M_L$ and run it with several input strings. Notice that JFlap does not have a reject state, so rules 1, 7, and 12, where $q_5$ occurs, should be deleted. When the machine, e.g., is in state $q_0$, and the input is $b$, the machine will reject the input, since no matching rule can be applied.

2. Modify it so that 1) it will accept $L = \{a^n b^n | n \geq 0\}$, and 2) it will then restore the original string.

3. Design a TM that copies a string. For example, if it finds a string $abaa$ on the input tape, it should leave $abaa abaa$ in the tape when it stops.

Send in .jff files....
How does a TM work?

Initially, $M$ receives its input $w = w_1w_2\cdots w_n \in \Sigma^*$ on the leftmost $n$ squares of the tape, and the rest of the infinite tape is blank.

The tape head starts on the leftmost square. As `␣` $\not\in \Sigma$, so the first `␣` marks the end of the input. (?)

Once $M$ starts, the computation proceeds according to $\delta$.

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:** 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 \textit{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\_\_\_\_\_\_\_.

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

Pretty much similar to the accept relation in the FA case (Cf. Page 17 in FA notes), 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 $s$,

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 ‘#’.

Thus, a TM is an algorithm....

**Homework:** 3.5.
Enumerable languages

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

In other words, a language $L$ is enumerable if there exists a TM $M$ that $L = L(M)$, the collection of all the words that $M$ accepts.

Procedurally, given a word $w \in \Sigma^*$, if $w \in L$, $M$ will accept it. If $w \not\in L$, $M$ will not accept it, i.e., $M$ will not get into the accepting state.

**Question:** Is there a problem?

**Answer:** Yes, a big one.
Decidable languages

There could be two cases when $M$ does not accept $w$: we have no problem if it gets the rejecting state. But, if it gets into an infinite loop, we have no way to know if it has fallen into it, or it is just still in the process of working this out.

As a result, we prefer those machines that halt at, i.e., either accept or reject, all input strings. In other words, for all words $w \in \Sigma^*$, after working for a finite amount of time, $M$ either gets into the accepting state, or rejecting state. Such a machine is called a decider. A decider that accepts some language also is said to decide that language.

**Definition:** A language is decidable if some Turing Machine decides it. Procedurally, given a word $w \in \Sigma^*$, if $w \in L$, $M$ will accept it. If $w \notin L$, $M$ will reject it.

**Homework:** 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, $n \not\in S$.

Realistically, the language is not even enumerable, in the sense that the above system will not pick up every ex-con, since, for some reasons, the guy has never been caught.

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.
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 language.

For example, 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 change let a Turing Machine accept more languages?

The answer is ‘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 right, the other moves it back.

\[ \delta(q, a) = (p, b, S) \equiv (\delta'(q, a) = (r, b, R), \delta'(r, x) = (p, x, L)), \]

where \( x \in \Sigma \).

We will further 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 if and only if some multitape TM accepts it.

**Proof:** Since 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 $k$ 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:
   \[ \# \cdot w_1 \cdots w_n \# \cdot B \# \cdot B \# \cdots \#
   \]

2. To simulate a single move, $S$ scans its tape from the first $\#$, to the $(k + 1)^{st}$ $\#$, to determine the symbols which should be under the $k$ dotted 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

We know that $NFA \equiv DFA$, but $NPDA \neq PDA$, we now show that $NTM \equiv TM$.

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

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

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 if and only if 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 $\delta$.

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. $D$ accepts $w$ if $N$ accepts using the current address 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.
Enumerators

With the original model of TM, we focus on its input. Roughly speaking, an enumerator is a Turing Machine that is attached to a printer.

Every time the TM accepts a string, the machine prints it out. The language accepted by an enumerator is the collection of all the strings it ever prints out.

Thus, the concept of enumerable language in term of a TM is rather input oriented; while the concept of an enumerator is output oriented. However, this difference does not matter.

**Theorem:** A language is enumerable if and only if some enumerator enumerates it.
One side is easy...

**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 a string when and if $E$ prints it out, namely, it accepts the same set of strings that $E$ enumerates.

In other words, for any enumerator $E$, we can construct a TM $M$ such that $L(M) = L(E)$. 
The other side is not so easy...

Now, assume that a TM \( M \) accepts a language, we construct the following enumerator, \( E \).

**Question:** Given a word \( w \in \Sigma^* \), should \( E \) print it out?

We give it to \( M \), and let it run. If \( M \) accept it, we print it out; otherwise, if \( M \) rejects it, we won’t. But, what happens if \( M \) runs it for a long time...

**A Shakespearean moment:** Do we know \( M \) has got into an infinite loop, so, we should not print it out; or is it still working on it, so we should wait a bit longer? Scratching....
What to do...

Let $s_1, \ldots, s_n, \ldots$ be a list of all the possible strings over $\Sigma^*$, we can construct the enumerator $E$ as follows:

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 of the above computation accepts $s_i$, print it out.

Since if a word $w$ is accepted by $M$, it will be accepted within a finite amount of time, thus it will be accepted with a certain fixed $i$. On the other hand, if $M$ gets into an infinite loop, it will never be accepted by $M$, thus will not be printed.

Thus, $E$ will print out a string if $M$ accepts it.

**Homework:** Exercise 3.7.
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 such as Church’s λ-calculus. All of them allow unrestricted and unlimited memory.

Remarkably, all models with this feature turns out to be equivalent to each other in the sense that whatever you can do with one of the models, you can do the same with any of the others.

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

For example, we can use either C or Java to write programs and these two languages look quite different from each other.

However, as we can write a compiler of C in Java and write an interpreter of Java in C, the programs in either language can be turned into its equivalent in the other language.

Hence, those two languages are really equivalent: If you can write a program in one language, you can write another in the other language to achieve the same effect.

Homework: 3.2 and 3.3.
The definition of algorithms

Informally, an *algorithm* is a collection of simple instructions for carrying out some task. This notion is informally 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 that something is not “computable”, how could we show there does not exist an algorithm?

Could that be the case that you have not worked hard and/or long enough?

In these places, we need a precise mathematic definition of algorithm.
Hilbert’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, i.e., the last one.

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.

It was shown back in 1970 that no such algorithm exists. In other words, 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.

But, \( M_1 \) 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, \( c_{\text{max}} \) is the coefficient with largest absolute value, and \( C_1 \) is the coefficient of the highest order term.

For \( 6x^3 - 31x^2 - x - 10 \), \( k = 4, c_1 = 6 \), and \( c_{\text{max}} = 31 \).

However, it has been shown in 1970 that it is impossible to calculate such bounds for multi-variate polynomial.
The Thesis

The definition of algorithm came in the 1936 papers by Church and Turing, respectively.

Church used a notational system, called the $\lambda$-calculus to define algorithms and 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 of algorithms and the precise definition has come to be called the *Church-Turing Thesis*: The informal notation of algorithms equals that of Turing Machines algorithms.
What does it mean?

This Thesis cannot be proved. It can only either be accepted or rejected.

It proposes that we give the necessarily informal notion of algorithms some precise meaning.

A Turing machine is certainly an algorithm. 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, we must fill in all the necessary details, when needed.

This Thesis has not failed us so far....

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 $\langle O \rangle$. If we have several objects, $O_1, \ldots, O_k$, we denote their encoding into a single object as $\langle O_1, \ldots, O_k \rangle$. 
How to encode a graph?

A graph $G(V, E)$ is typically represented as an ordered pair, whose first element, $V$, is the collection of all the nodes, while the second element, $E$, 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 $\langle G \rangle = (1, 2, 3, 4) ((1, 2), (2, 3), (3, 1), (1, 4))$. 
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 nodes.

It also checks other things. For example, no repetition in the first list, and every component 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

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

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

Given an input $\langle G \rangle$, the encoding of a graph $G$,

1. Select the first node of $G$ and mark it.

2. Repeat the following steps until no new nodes are marked.

3. For each node in $G$, mark it if this node is associated via 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 a 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.
Otherwise, if the edge list is not over, yet, $M$ checks for the next edge. If there is no more edge left, $(n_1, n_2)$ is not an edge in $G$, $M$ moves the underlining of $n_2$ to the next dotted node, and repeat this step.

If there are no more dotted nodes 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.

*By the Thesis, there is no need to dig any further. This problem is solvable.*