Turing Machines

Table of Contents

[The Church-Turing Thesis 8](#_Toc138882370)

[Add Stop Symbol to Left End of Tape 9](#_Toc138882371)

[Algorithms 9](#_Toc138882372)

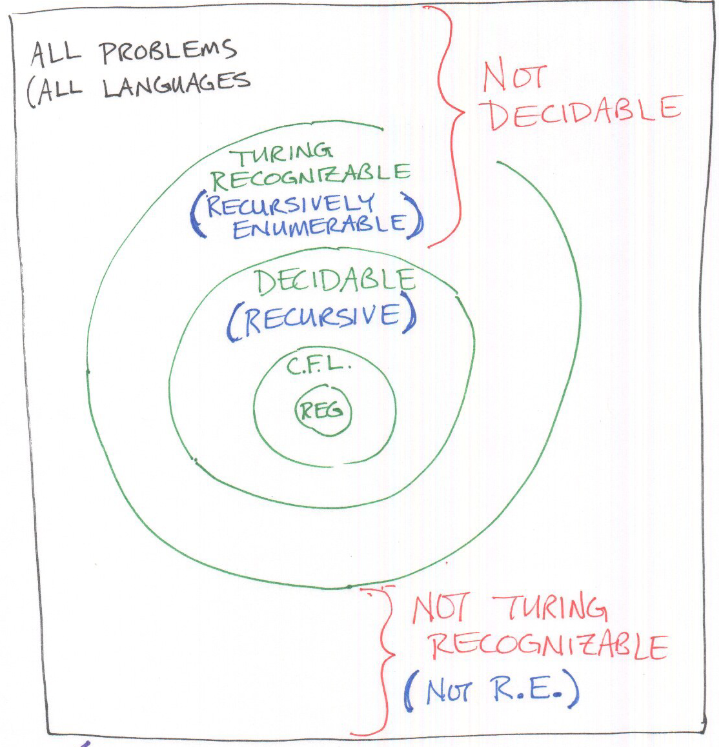
[Marking Symbols 11](#_Toc138882373)

[Non-Deterministic Turing Machines 13](#_Toc138882374)

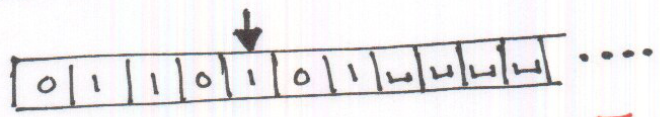
[Algorithms 15](#_Toc138882375)

[Enumerators 17](#_Toc138882376)

So far, we have studied Finite State Machines (FSMs), which work with regular languages, and Pushdown Automata (PDA), which work with context-free languages. The next type of machines we will study are **Turing Machines** (TMs). These machines work with three types of languages on top of the previous two, decidable or recursive languages, Turing-recognizable or recursively-enumerable languages and not Turing-recognizable languages. All of these languages can be visualized using a **language onion**.

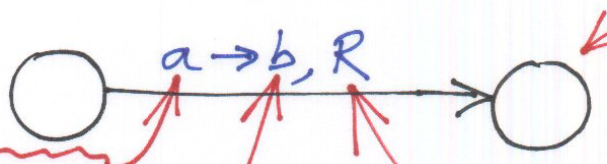


In an FSM, we just had an input string which we could parse. In a PDA, we additionally had a stack which behaved as memory. In TMs, all of this is replaced with a **tape**.

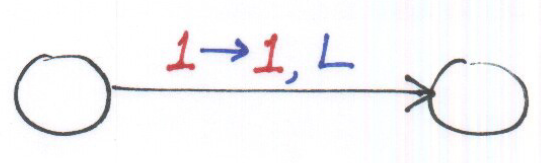
\

The tape is **infinitely long**, and at the end of the input string, the remaining cells are filled with a special **blank symbol**, . The current position being read is marked with a **pointer** called the **tape head**. The tape head can move left and right, read from a cell and write to a cell.

To make the transitions, we still use a **Deterministic Finite State Machine**, which is also sometimes referred to as the control or the program. At every transition, we move the tape head either left or right, indicated using or , read a symbol, suppose , and write a symbol, suppose .



If we do not want to change the current input, we can just write the same value that we read.



The alphabet for a TM is typically along the lines of . Additionally, we have the special blank symbol, represented using , which cannot be a part of , i.e., . Along with this, we also have the **tape alphabet**, . This is the set of alphabets that are used in the tape and is a superset of , i.e., . We can actually write any symbol we want onto the tape, even if it is not part of the input. This will become useful when we look at some of the problems we can solve using TMs.

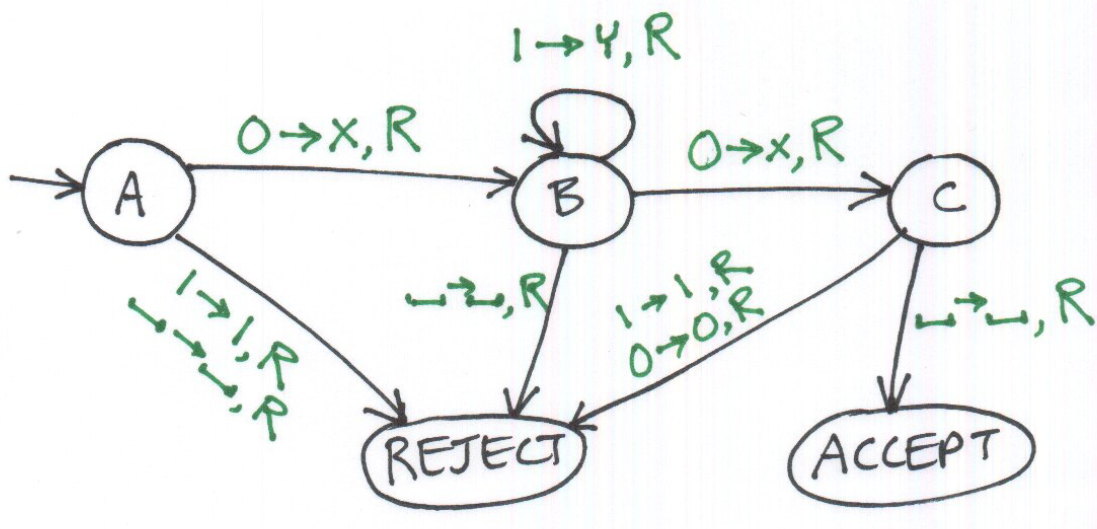
Like all FSMs, the FSMs used with TMs start at an initial state and transition through various states before reaching a final state. The final state in this case can be either an **Accept** state or a **Reject** state.

The computation of a TM can go three ways:

1. The program **halts and accepts**, meaning the machine has entered the Accept state.
2. The program **halts and rejects**, meaning the machine has entered the Reject state.
3. The program **loops**, meaning the machine fails to halt.

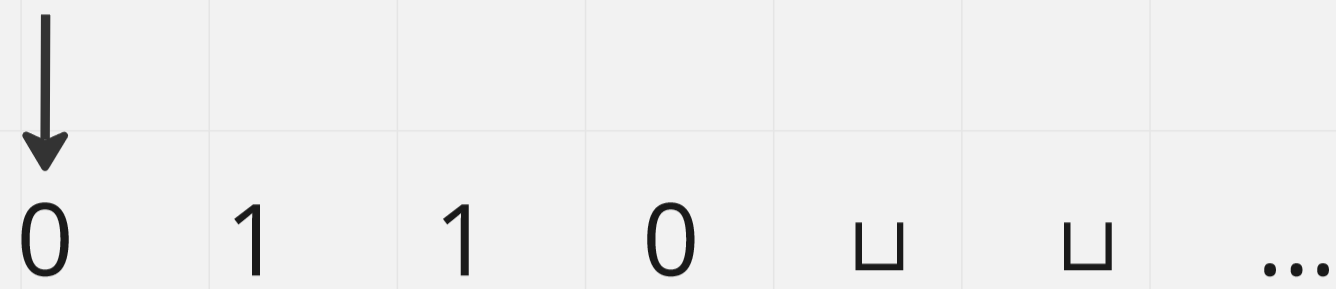
For **decidable languages**, TMs will always halt and either reject or accept the input. For **Turing recognizable languages**, the TMs will halt and accept correct inputs and for incorrect inputs, will either halt and reject or loop. For non-Turing recognizable languages, the TMs will be unable to even reliable recognize correct inputs.

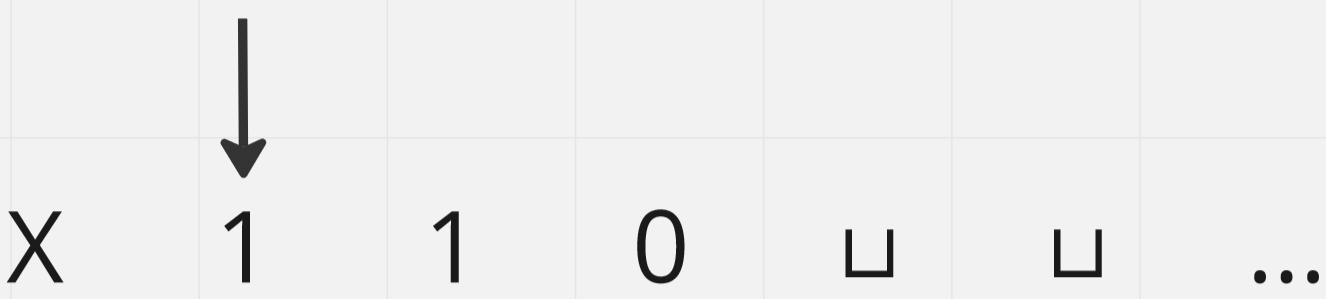
A few examples should make the working mechanism clearer. Suppose we have a language . The TM for this language is shown below:

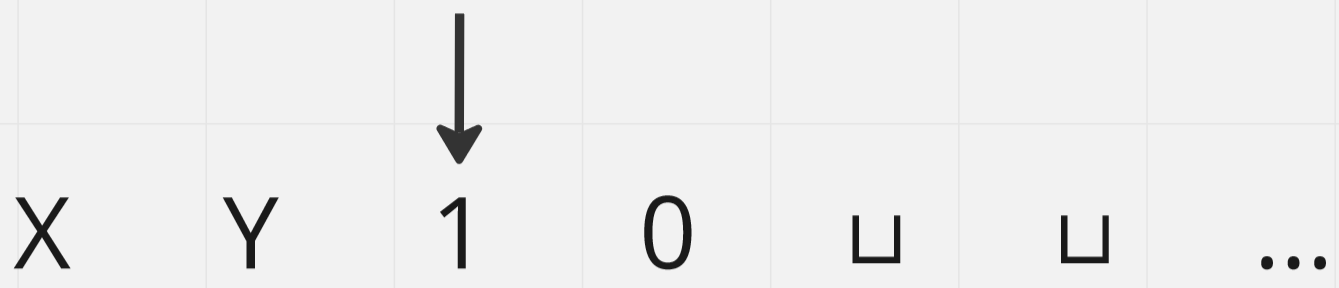


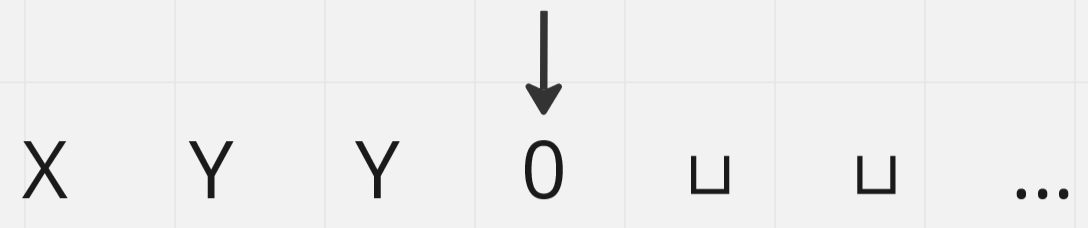
For this machine, .

To understand what is happening, consider the updates made to the tape for the input 0110.











In the above TM, we have explicitly shown the reject state, but this can actually be omitted and will not be shown in the rest of the examples.

The need to replace s with s and s with s is not very clear in this example, since it is a simple one. A more complicated example should make the need for it clear.

We can actually model this language as an algorithm first, instead of directly making an FSM.

Repeat until no more '0's  
 Change '0' to 'X'  
 Move right to first '1'  
 If none  
 REJECT  
 Change '1' into 'Y'  
 Move left to leftmost '0'  
Make sure no more '1's remain

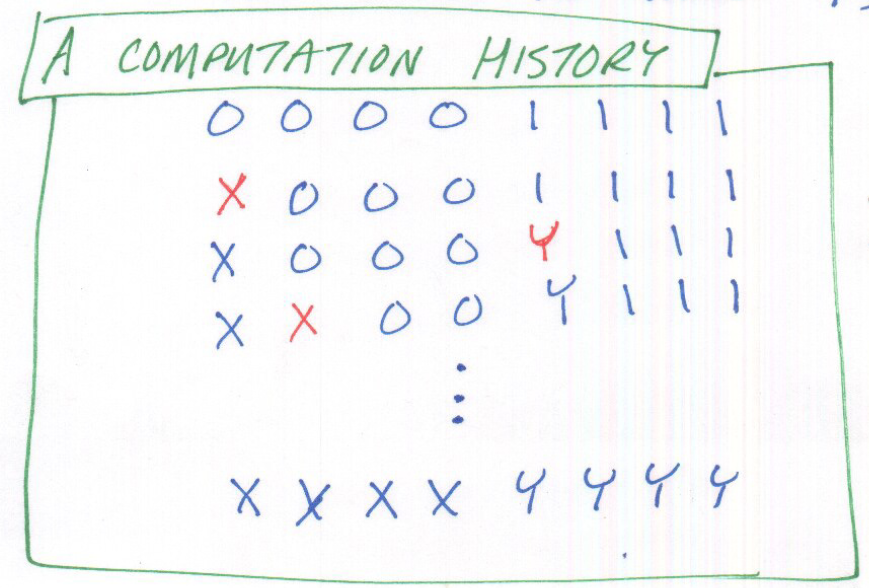
PSEUDOCODE

The fact that we could model an algorithm for this language indicates a critical part of TMs. So far, we have only dealt with languages and checking if provided strings are a part of the language. Using TMs however, we should be thinking of strings as **problems**. When we are checking if a string is part of a language, we are actually trying to determine if a given situation is solvable or not. We will get back to exactly what this means later on.

The FSM for the above algorithm is shown below:

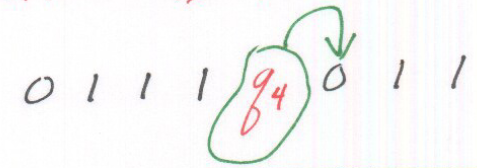


For the string ,



This is why the and are important. When going backwards, we are using them as markers to determine if we are in the correct position yet.

A **computation history**, such as the one shown above, is a sequence of **configurations** containing only legal transitions. Each configuration is a string that gives the current state, location of the tape head and the entire contents of the tape. It is shown like this:



Given all of this information, we can formally define a TM as . Here, .

Turing machines are used to:

* ‘Decide’ a language
* ‘Recognize’ a language
* Compute a function

A **computable function**, also called a decidable function, is one which is defined for all inputs. This is opposed to a **partially-computable function**, a.k.a. a semi-decidable function, which is undefined for some inputs.

## The Church-Turing Thesis

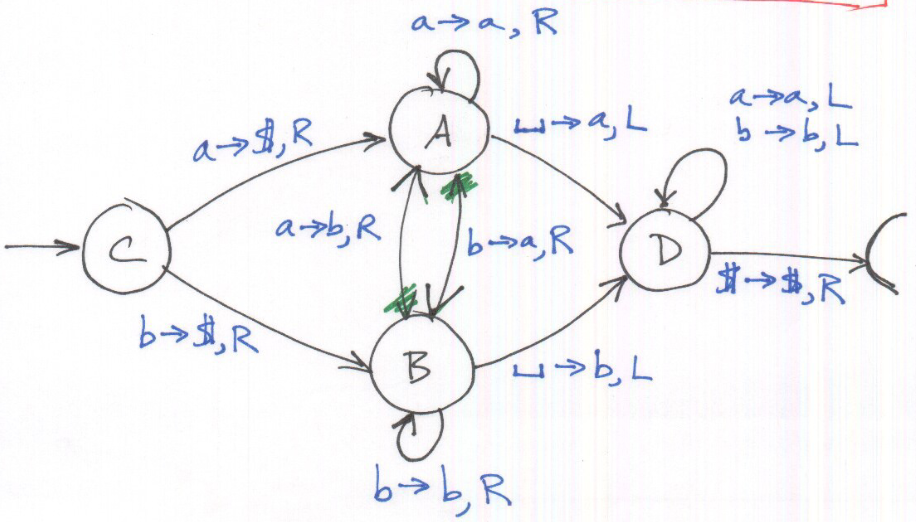
For a long time, scientists had difficulties defining what exactly ‘computable’ meant. Two prominent theories arose to answer this. The first was by Alonzo Church, who proposed **lambda calculus**. The second was by Alan Turing, who proposed **Turing Machines**. Essentially, he suggested that if a problem is **algorithmically computable**, it is computable by a Turing Machine. In addition, lambda calculus and Turing Machines are also equivalent with regard to what they can compute.

There are several variants to TMs:

* Use multiple tapes instead of a single one
* Make the tape infinite on both ends
* Use a small set of alphabets or a large one
* Keep the head at the same place and use markers
* Use non-determinism

## Add Stop Symbol to Left End of Tape

If we want to place a special symbol, suppose $, to the left end of the tape to indicate the starting position, we need to shift the input by 1 cell. We can do this using a TM.



## Algorithms

Turing machines are actually fairly low level. Just like we start with machine code and slowly progress to higher level representations like assembly code, C code and finally algorithms, we start with Turing machines, with a complete specification, and slowly progress to an outline of an algorithm, which might still mention a tape head or such, and finally a high-level specification of an algorithm, which makes no references to a TM at all.

Example

Suppose we want to build a TM for . This is not a CFL, so being able to build this will also prove that CFLs are a proper subset of decidable languages.

We have already seen a TM that can turn into . We can use this as a subroutine.

Step 1: Convert into .

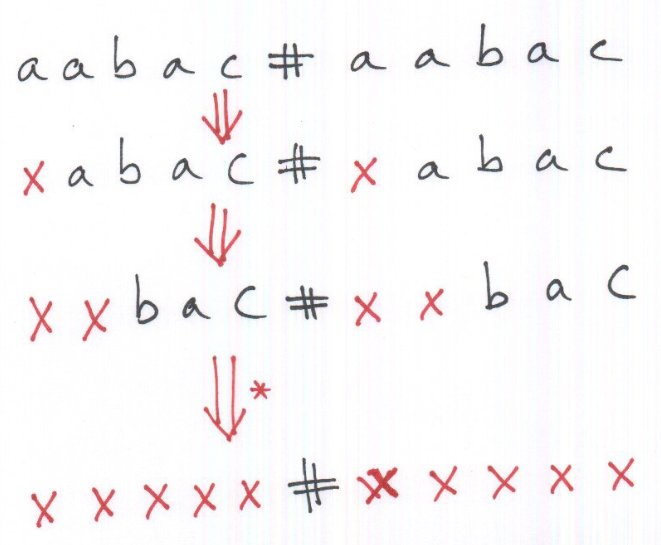
Step 2: Convert into .

Step 3: Take the two TMs that were used to complete the two steps above and stick them together.

Example

Compare if two strings, separated by a #, are the same, i.e., .

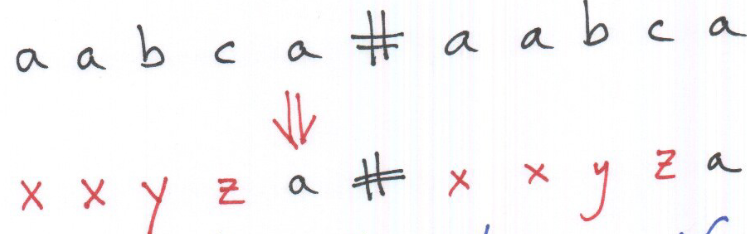
Solution: Turn each symbol into X after it has been examined.



Example

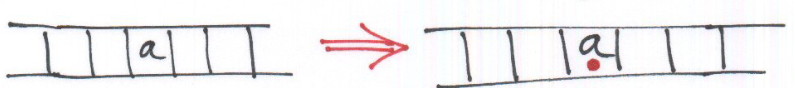
Repeat the same problem as above, but this time without destroying the original string.

Solution: Use a separate symbol to replace each symbol in the original string, so that it can be restored.

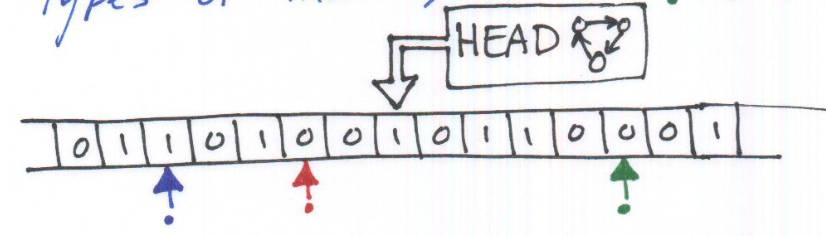


## Marking Symbols

Another technique that is frequently used is to mark symbols with a dot.



We could also use multiple markers.

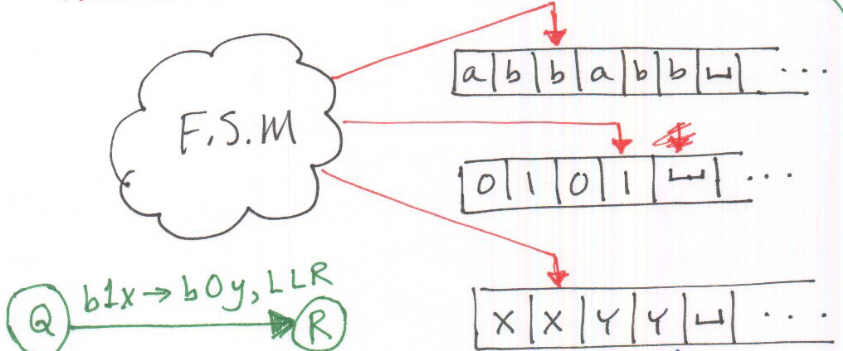


This could be used to form a ‘multi-tape’ without actually having multiple tapes.

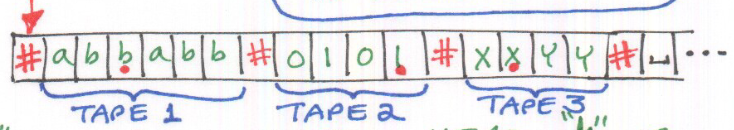
From the above statement, we come to a **theorem**:

‘Every multi-tape Turing machine has an equivalent single-tape Turing machine.’

Suppose we have a multi-tape Turing machine with three tapes.



For the single-tape equivalent, we need separate markers for each of the tape-head positions (in addition to the actual tape head arrow).

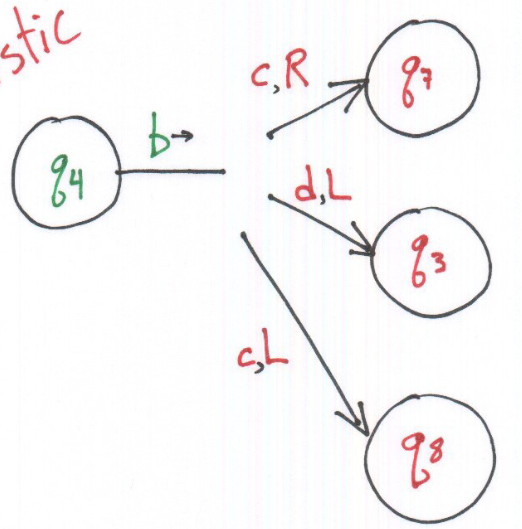


First, we scan the tape once to determine which symbols are under each of the marked positions. Once this is determined, we make a transition. This involves another scan which is used to update the values at the marked positions and also move the marks.

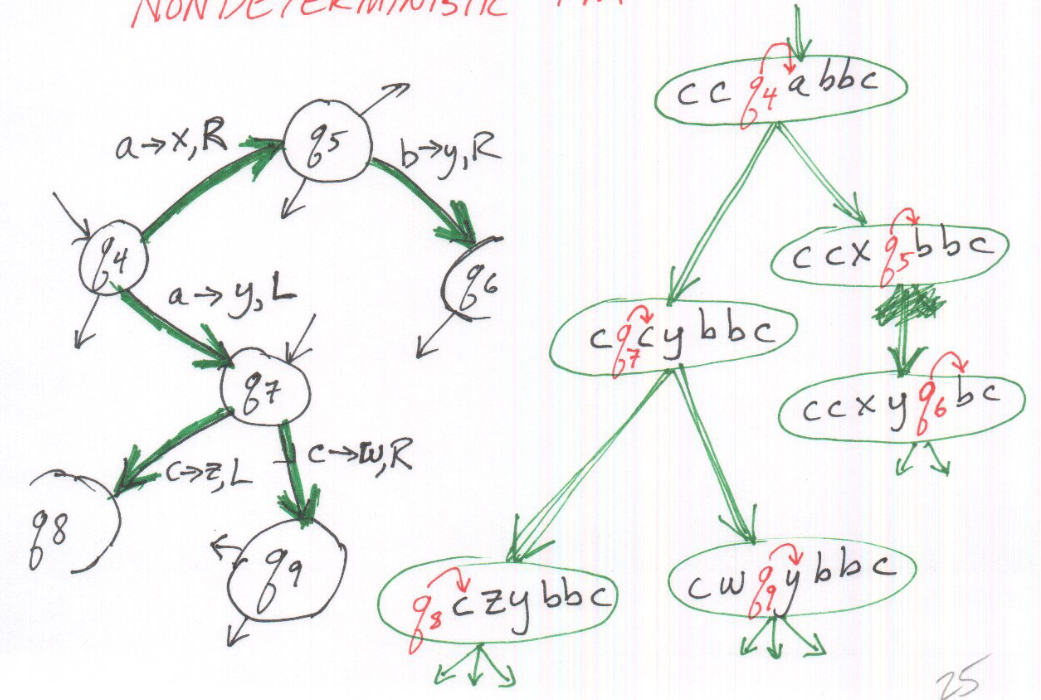
If, during the transition, we find that a section of the tape has ended (as marked by the # symbol), but other sections have not, we have to add symbols to the end of that section of the tape so that we can continue working. Since we are adding more symbols, we need to shift the rest of the contents of the tape to the right.

## Non-Deterministic Turing Machines

In non-deterministic machines, we have multiple potential transitions for each input. The transition function is given by .



This translates to multiple possible successor configurations, creating a tree of configurations.



While some of the branches in this tree will halt, others will not. We need to find the Accept states without falling into a loop.

There are three possible outcomes to a non-deterministic computation tree. If there is any branch in the tree that ends in an Accept state, then the non-deterministic TM will go to that state. If all the branches end in Reject states, then the non-deterministic TM will reject the input. Only if no Accept state is reached and there is some other branch that never halts will the non-deterministic TM loop.

Theorem: Every non-deterministic TM has an equivalent deterministic TM.

To prove this theorem, we have to use proof by construction. Given a non-deterministic TM, we need to construct an equivalent deterministic TM. To this, we need to search the tree looking for an Accept state. We use BFS to do this due to the potential of ending up in an infinitely long branch, which makes DFS useless. This is easiest if we use a multi-tape implementation.

We use three tapes for this implementation:

* **Input Tape** – This is the initial input and is never modified, only read.
* **Simulation Tape** – We use this as if it were the tape of a deterministic TM.
* **Address Tape** – This tape tells us which choice to make at every transition, thus controlling the search.

The algorithm for this is as follows:

INITIALLY: Tape 1 contains the input  
 Tape 2 and 3 are empty  
  
Repeat  
 Copy Tape 1 into Tape 2  
 Run the simulation  
 Use Tape 2 as "The Tape"  
 When choices occur (i.e., when non-deterministic branch points are

encountered), consult Tape 3  
 Tape 3 contains a "Path". Each number tells which choice to make  
 Run the simulation all the way down the branch, as far as the

address/path goes (or the computation dies out)  
 Try the next branch  
 Increment the address on Tape 3  
If ACCEPT is ever encountered  
 HALT and ACCEPT  
If all branches REJECT or die out  
 HALT and REJECT

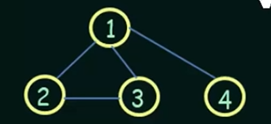
PSEUDOCODE

## Algorithms

Any problem can be solved using a Turing machine. The problem is thus a **language** and a single instance of the problem is a **string**, i.e., we encode the problem into a string.

For example, consider the problem of figuring out whether or not a graph is connected. We can encode this as follows:

If is connected, then the TM will accept the string. Otherwise, it will reject it. This is a decidable language so the TM will never loop.



One possible way to represent this problem is to use , with the first group representing the nodes and the second group representing the list of edges. Here, . The tape contains the string character by character as we see above.

In reality though, we are unlikely to make this Turing machine from scratch. This is because a TM is actually a very low-level specification, comparable to using machine code as opposed to a programming language. Above **TM specification**, we could have the **implementation level**, which consists of representing the contents of the tape, the motion of the head, etc. We, however, will be using a **high-level representation**, algorithms, which consists of just using **pseudocode**.

The algorithm for detecting a connected graph will be as follows:

Select a node and mark it  
Repeat until no more nodes can be marked  
 For each node N  
 If N is unmarked AND there is an edge from N to an already marked node  
 Then mark node N  
 END  
For each node N  
 If N is unmarked  
 Then REJECT  
END  
ACCEPT

PSEUDOCODE

Compared to the above, the **implementation-level** would look like this:

* Check that input describes a valid graph
  + Check node list
    - Scan “C”, followed by digit
    - Check that all nodes are different, i.e., no repeats
  + Check edge list
  + etc.
* Make first node
  + Place a dot under the first node in the node list
* Scan the node list to find a node that is not marked
* etc.

## Enumerators

An **enumerator** is just like a TM in that it also has an **infinite tape** and an **FSM** as a control. However, it also has a **printer**. The tape is initially empty. The printer prints out strings from a language one by one and the FSM processes it. The language could be **infinite**, print the strings in **any order** and even **repeat strings**.

Since the printer could print strings in any order, we could end up with longer strings at the start. This is problematic, since those strings will take too long to process. Instead, we can use **non-determinism** to process all of the strings simultaneously. First, we list out all of the strings, , e.g, . Suppose we have a string which will cause an infinite loop but we have another string which will be accepted. To make sure is not prevented from being accepted by , we process each string a little bit in turn, i.e. the order becomes (partial), (partial), , (partial), (partial) and so on. The algorithm for this is shown below:

For i = 1, 2, 3, ... (infinite loop)  
 For j = 1 to i  
 Simulate M, the Turing Machine  
 Use as input  
 Run simulation for i steps  
 If M accepts within i steps  
 Then print   
 END  
END

PSEUDOCODE

A visualization of the algorithm would look like this:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | … |
|  |  |  | A | A | A | A | … |
|  |  |  |  |  |  |  |  |
|  |  |  |  | A | A | A | … |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

Thus, is accepted after iterations, while is accepted after iterations. Notice that the iterations after this do not exclude the accepted strings. They are still processed and remail in the accepted state. Once we have reached the end of the input to some string (and accepted it), we have an infinite number of blanks, so we can continue doing this.

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

Proof:

Given an enumerator , we can construct a Turing-Machine as follows:

On input W,  
 Run E  
 Compare each output string to W  
 If we find a match, accept it.

PSEUDOCODE

Given a TM , we can construct an enumerator as follows:

Construct E using M as a subroutine.  
Run M on all possible strings in .  
If M ever accepts a string, print it out.

PSEUDOCODE

In the above process, we might run into the problem of having loop forever with some string. The solution to this is to run the strings in parallel.