Theory of Computation – Background Knowledge

Sets:

N = Natural numbers {1,2,3,4…}

Z = integers {…, -2, -1,0,1,2…}

∅ = Empty set

U = Union

∩ = Intersection

S x T = Cross product

→ N x N = { (i,j) | i,j ≥ 1}

(1, a) = tuple, sequences

P(s) = powerset

Functions:

F(x) = y

F: Domain --> Range

A TM is defined by $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \square, F \rangle$

where

Q	is a finite set of states
Σ	is the input alphabet
Γ	is the **tape alphabet**
δ	is the **transition function** (below)
$q_0 \in Q$	is the initial state
$\square \in \Gamma$	is the **blank symbol**
$F \subseteq Q$	is the set of final states

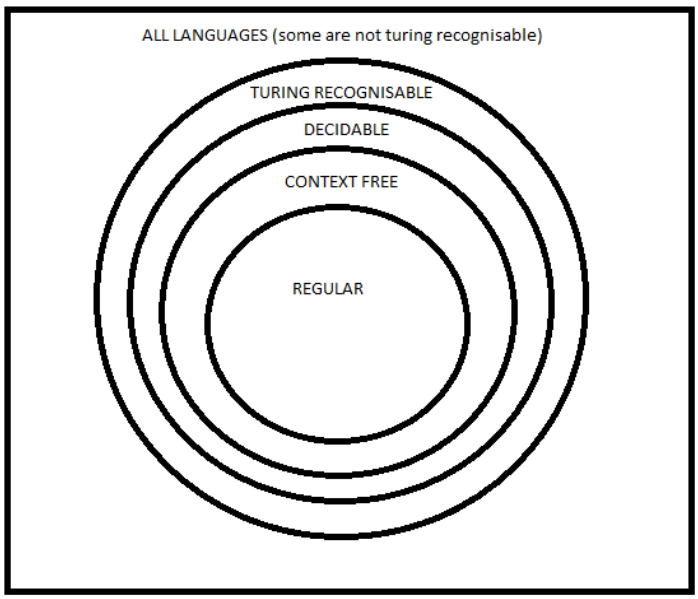
with $\Sigma \subseteq \Gamma - \{\square\}$ and $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$

Turing Machines

Turing machines are a new model of automata that are slightly more elaborate than those reviewed in earlier sections of THE2.

Turing machines are a 'model' for all computers. During this module we will cover three main languages in relevance to Turing machines:

1. Decidable languages
2. Turing recognisable languages
3. Languages that are not Turing recognisable



All Turing machines have the same equivalent power, some textbooks and lecturers will describe and teach Turing machines in different ways, however in the end they are all equivalent and correct

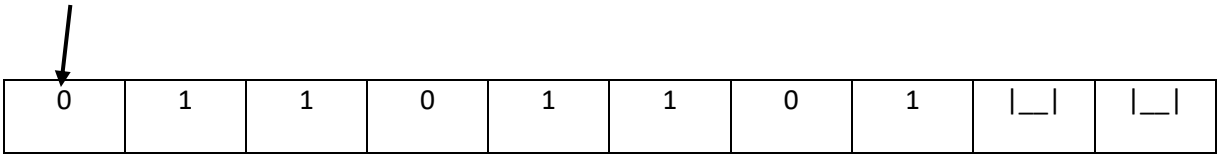
Turing machines use a 'tape' as a data structure. This is a sequence of cells containing a symbol from the 'tape' alphabet. It is infinite in the right direction and finite in the left. An empty square has a 'blank' symbol. The current position is known as the tape head, at any one point in computation the head is placed, it can move to either the right or the left by one step.

TURING MACHINES

Tape Alphabet:

Typical: ∑ = {0,1}

Also common: ∑ = {0,1, a, b, x, #, $}

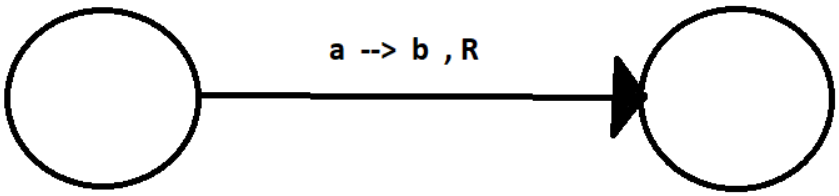


- Blanks show empty parts of the infinite tape
- The tape starts at position 0, the leftmost position, shown by the tape head

Rules of operation:

At each step of the computation we must;

1. Read the current symbol
2. Update the same cell
3. Move exactly one cell left or right (if we are at the leftmost position and try to move left, we will stay at the same position, the tape head does not move).



A = Symbol to read

B = symbol to write

R = Direction to move: L or R

 = states

You can avoid updating the cell by writing to the same symbol that you are reading from;

a -> a, L

- Control of a Turing machine is with a sort of finite machine
  - Initial state
  - Final state (there are two):
    - The accept state
    - The reject states
- Computation can end in three ways:
  - Halt and Accept:
    - Whenever the machine enters the accept state, computation immediately halts
  - Halt and Reject:
    - Whenever the machine enters the REJECT state, computation immediately halts
  - Looping
    - The machine fails to halt

∴ TM is Deterministic.

Turing Machine Examples

Example:

$L = 0^N 1^N$

$\sum = \{0,1\}$

Algorithm:

0	0	0	0	1	1	1	1	|__|	|__|

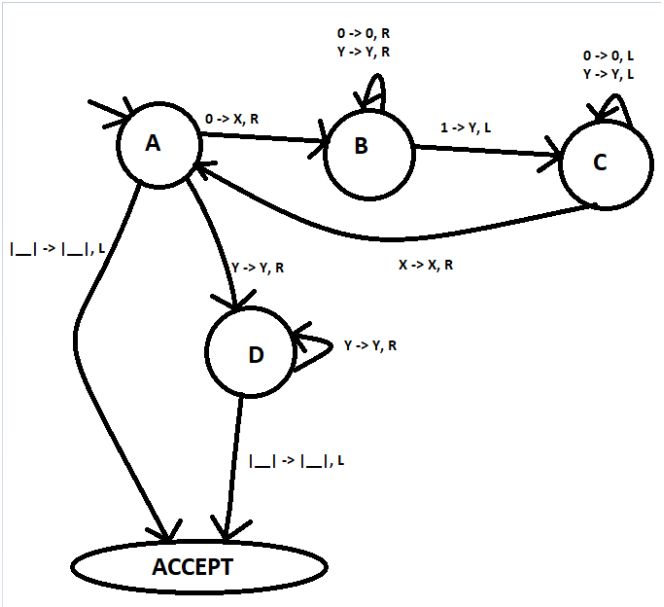
Pseudocode:

Change 0 to X
Move right to first 1
    If NONE: Reject
Change 1 to Y
Move left to leftmost 0
Repeat until no more 0 remain
Ensure no more 1 remain

Computation history

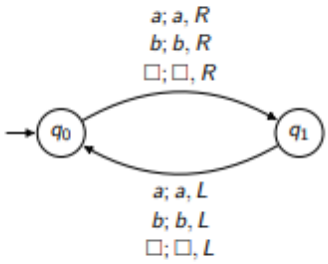
0	0	0	0	1	1	1	1
X	0	0	0	1	1	1	1
X	0	0	0	Y	1	1	1
X	X	0	0	Y	1	1	1
X	X	0	0	Y	Y	1	1
X	X	X	0	Y	Y	1	1
X	X	X	0	Y	Y	Y	1
X	X	X	X	Y	Y	Y	1
X	X	X	X	Y	Y	Y	Y

Tape alphabet: {0,1, x, y, |__|}



Definition of Turing Machines and Related Language Classes

A non-halting TM



Clearly, this runs forever, alternating between states q_0 and q_1, b never altering the tape.

The language accepted by a TM

- We can view TMs as language accepters
- a string is accepted if it causes the TM to halt in a final state, when started from the standard initial configuration (q_0, leftmost input symbol)
- a string is rejected when:
  - the TM halts in a non-final state; or
  - the TM never halts

Acceptance in symbols:

$$L(M) = \{w \in \Sigma^+ : q_0 w \vdash^* x_1 q_f x_2 \text{ for some } q_f \in F, x_1, x_2 \in \Gamma^*\}$$

A Turing machine can be described formally as a tuple with such components:

(Q, ∑, Γ, δ, q., qACCEPT, qREJECT)

Q: Set of states

∑: Input alphabet, ∑ ⊆ Γ

Γ: Tape alphabet

The input cannot contain a blank (|_|):

|_| ∉∑ and |_| ∈ Γ

q0 = initial state, q0 ∈ Q

qACCEPT ∈ Q

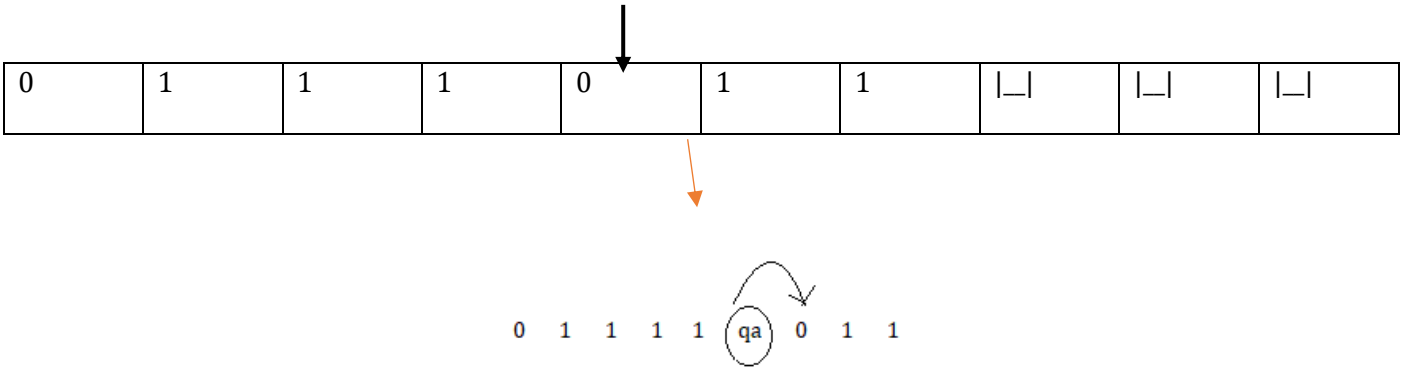
qREJECT ∈ Q

δ: Q x T -> Q x T x { L, R} this is the deterministic transition function (it has to always be deterministic)

Configuration:

- Gives the entire state of the machine, a snapshot at some time in the machine
- You need the following:
  - Contents of the tape
  - Location of the tape head
  - Current state

The easiest way to do this is to provide a configuration as a string like this:



A sequence of configurations, starting with the 'start configuration', and ending with an ACCEPTING/REJECTING configuration, and containing only legal transitions provides a COMPUTATION HISTORY

Decidable Languages:

- Also 'recursive', 'computable', 'solvable' etc.
- When given a string as input, the Turing machine will always halt.
  - The Turing machine will ACCEPT if it in L
  - The Turing machine will REJECT if I not in L

Turing Recognisable Languages:

- Also 'Recursively Enumerable', 'Partially decidable' etc.
- When given a string that is in the language, the Turing machine will always halt and ACCEPT.
- When given a string that in not in the language the Turing machine will either REJECT or LOOP.

Not Turing Recognisable Languages:

- Also 'NOT Recursively Enumerable', 'NOT Partially decidable' etc.
- Cannot recognise members reliably

Turing Machine uses:

1. To 'decide' a language
2. To 'recognise' a language
3. To 'compute' a function

Computable = Decidable. Totally computable; Defined on all inputs.

Partially Computable functions = Undefined on some inputs, 'semi decidable functions.

Church Turing Thesis

During the 1930's, there was disagreement about what computable means. Alonzo Church said it was 'Lambda Calculus' and Turing said it was Turing machines instead.

There were also many variations of such Turing machines:

- Can there be one tape only, or many?
- Infinite on both ends?
- Can it only have such a tiny alphabet? {0,1}
- Can the head remain in the same position?
- Can we allow non-determinism?
    - All of these variations are equivalent in computing capability!

The thesis:

Any computation that can be carried out by

mechanical means can be performed by some Turing machine.

Not proven (how would you?), but can be seen as a definition of a mechanical computation, and there is various evidence to support this:

- anything that can be done on an existing digital computer can also be done by a TM
- no suggestion (yet) for a problem solvable by an algorithm which can't be solved by a TM
- there are various alternative models for computation, but none is more powerful than a TM
-

Turing machines with a Stay option

- Instead of insisting that the tape head moves either Left or Right on each transition, we allow the option to Stay in the same position
- Claim: the class of TMs with Stay is equivalent to the class of standard TMs One direction is easy, since a TM with Stay is clearly an extension of the standard TM
- In the other direction: any TM with Stay can be simulated by a standard TM, by replacing each Stay transition with a pair:
    - ❖ a transition that rewrites the symbol and moves R; and
    - ❖ a transition that leaves the tape unchanged and moves L
- where the necessary state change happens across the pair of transitions.

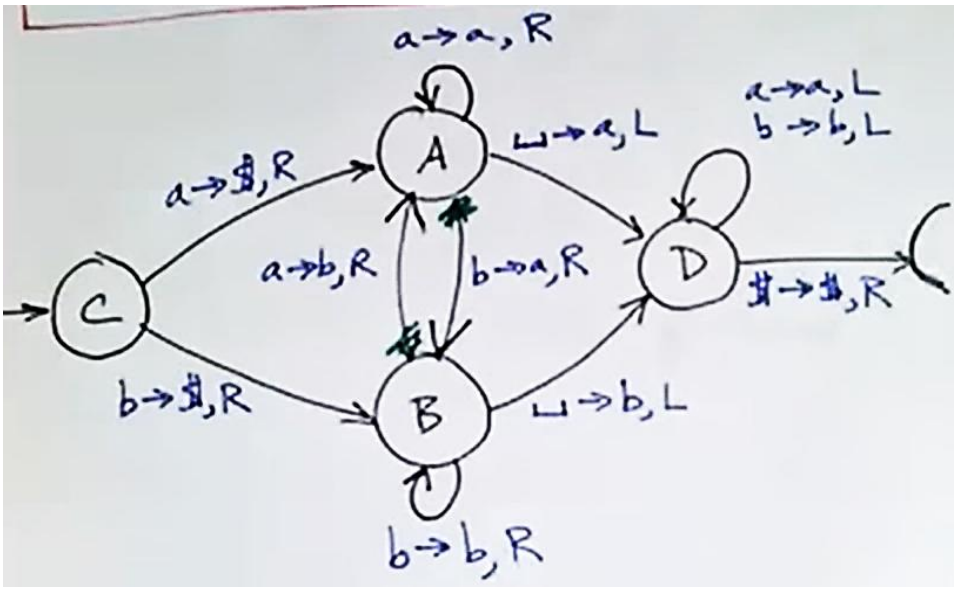
Turing Machine Programming Techniques

Problem 1: How can we detect the left end of the tape?

Goal: We want to put a special symbol $ on the left end and shift the input over 1 cell to the right

a	a	a	b	b	a	|__|	|__|	|__|	.....

$	a	a	a	b	b	a	|__|	|__|	.....



Turing machine process:

States, transition functions (complete TM specification)

Outline of the algorithm 9 tape head movement, data representation)

High level specification (no TM specific details)

EXAMPLE 1:

Build a Turing machine that recognises the language: $0^n\ 1^n\ 0^n$

- build a Turing machine to 'decide' the language
- this language will not loop, and it is not context free, therefore this will prove that :

  context-free languages ⊂ Decidable languages

1. To do this we can incorporate the previous example of $L = 0^N\ 1^N$

0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0

↓

X	X	X	X	X	X	Y	Y	Y	Y	Y	Y	0	0	0	0	0	0

2. The next step is to change this to Y^N0^N
3. Next, we should build the final TM by combining these smaller TMs together into one larger TM

EXAMPLE 2:

A Turing machine to decide {w # w |w ∈ {a, b, c}*} (Compare two strings)

- Use a new symbol such as 'x'
- Turn each symbol into an x after it has been examined

a	a	b	a	c	#	a	a	b	a	c

x	a	b	a	c	#	a	a	b	a	c

x	a	b	a	c	#	x	a	b	a	c

↓

x	x	x	x	x	x	x	x	x	x	x

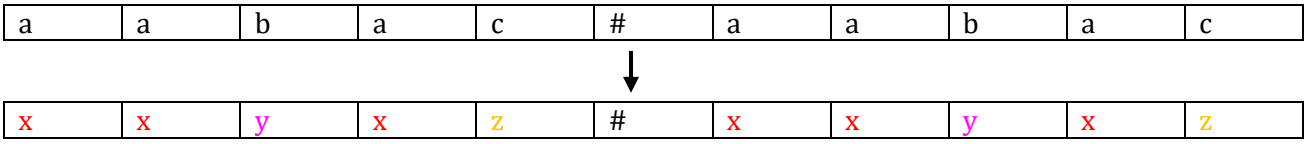
Problem:

Can we do this task non-destructively? Without losing the original strings?

Solution:

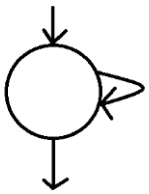
Mark each symbol to keep track of what we have already done, add some new symbols to help:

- a -> x
- b -> y
- c -> z



Later we can restore the strings:

- x -> a, R
- y -> b, R
- z -> c, R



marking symbols:

- mark each symbol with a dot
- this can be used to remember the location of something on a tape
- you can also use a dotted arrow to mark a head
- example of use may be something such as: 'let P be the point to the begging of the second string, let q point to ….'

$$\Gamma = \{a, b, c, \dot{a}, \dot{b}, \dot{c}, \#\}$$

Multi Tape Turing Machines

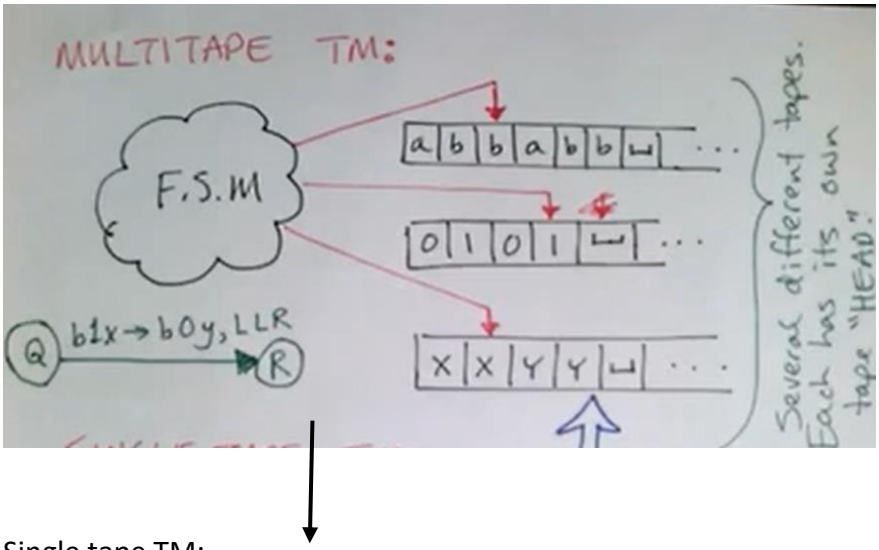
Theorem:

- Every multi tape Turing machine has an equivalent single tape Turing machine
- 'equivalent' means that it decides/recognises the same languages.
- It is not about speed, efficiency or ease of programming

Proof:

Given a multi tape Turing machine, show how to build an equivalent single tape Turing machine.

- Need to store all tapes on a single tape (show data representation)
- Each tape has a tape head (show how we store this information)
- Need to transform a move in the multi tape into one or more moves in the single tape TM



- If the pointer is at b1x, we can
- move in the set direction LLR and
- replace with b0y.
- here K=3 tapes.

Single tape TM:

#	a	b	b	a	b	b	#	0	1	0	1	#	x	x	y	y	#
		•									•			•			

TAPE 1 TAPE 2 TAPE 3

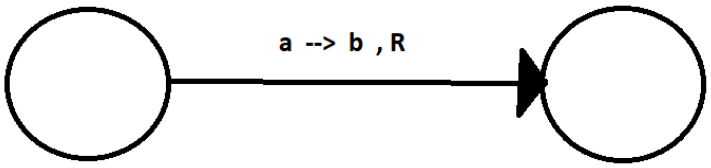
- Add dots to show where each head 'K' is.
- To simulate a transition from the state Q, we must scan our tape to see which symbols are under our tape heads
- Once we determine this, and we are ready to 'make' the transition we must scan across the tape again to update the cells and move the dots
- Whenever one head moves off the right end, we must shift our tape so we can insert the |__|

Non-Determinism in Turing Machines

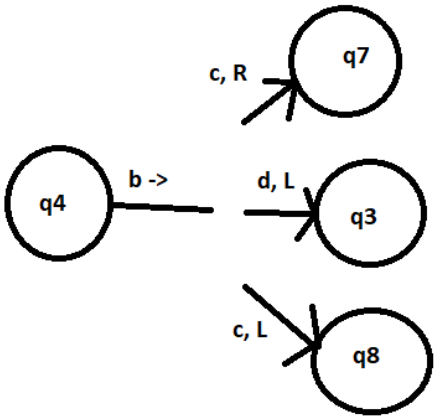
Transition function must be modified for ND TMs:

$$\delta: Q \times T \rightarrow P(Q \times T \times \{L, R\})$$

DETERMINISTIC



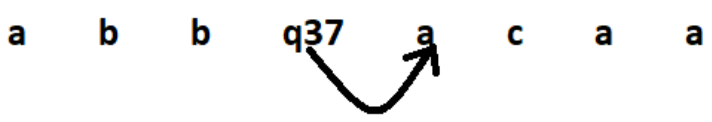
NON-DETERMINISTIC



Configuration:

- A way to represent the entire state of a TM at one moment of computation
- A string which captures
  - The current state
  - The current position of the head
  - The entire tape contents

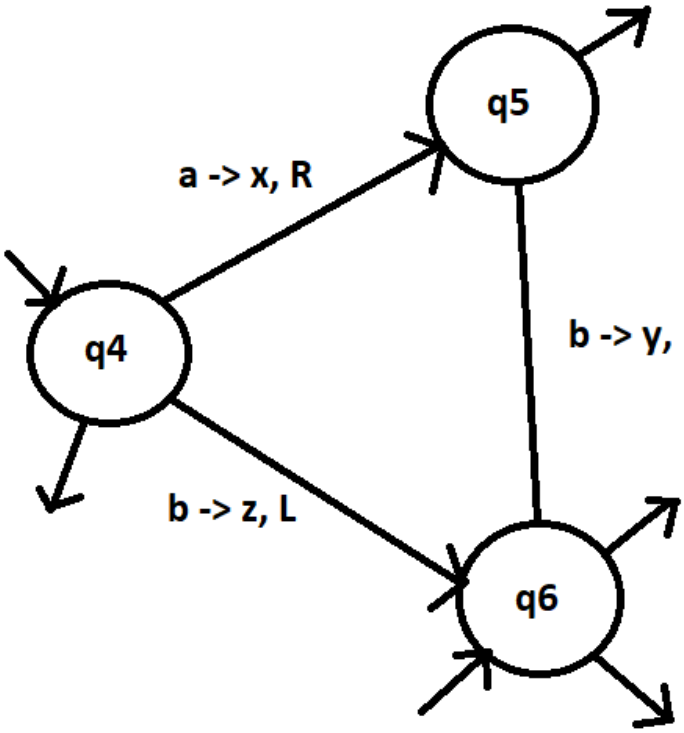
a	b	b	a	c	a	a	|__|	……

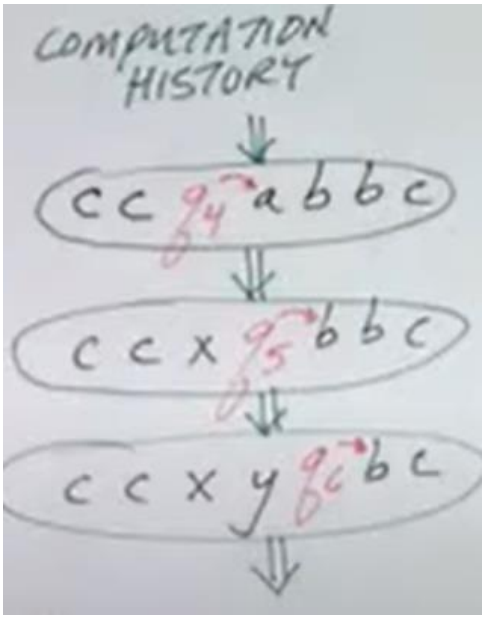


With non-determinism:

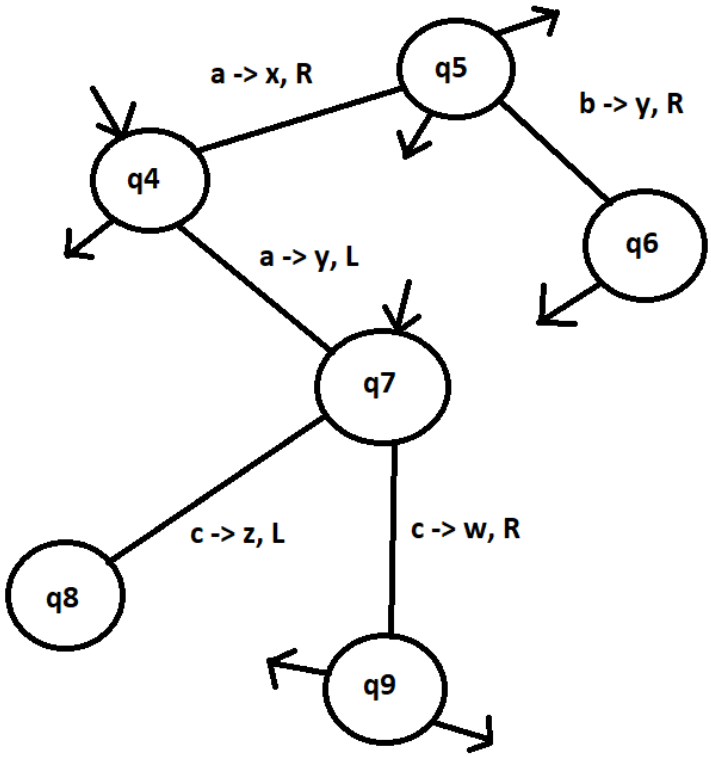
At each moment in the computation, there must be more than one successor configuration

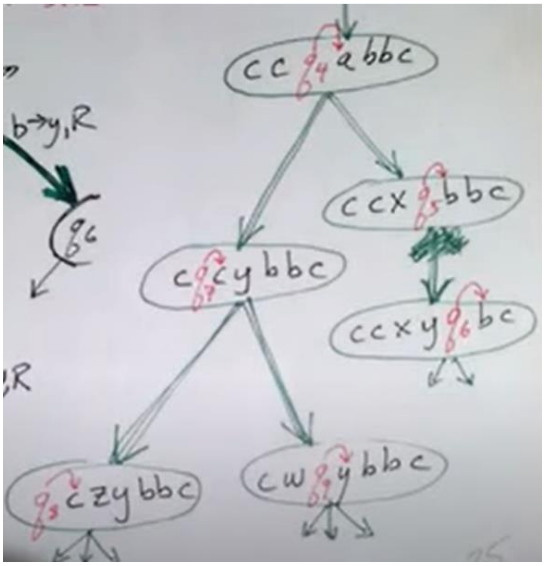
Deterministic TM:





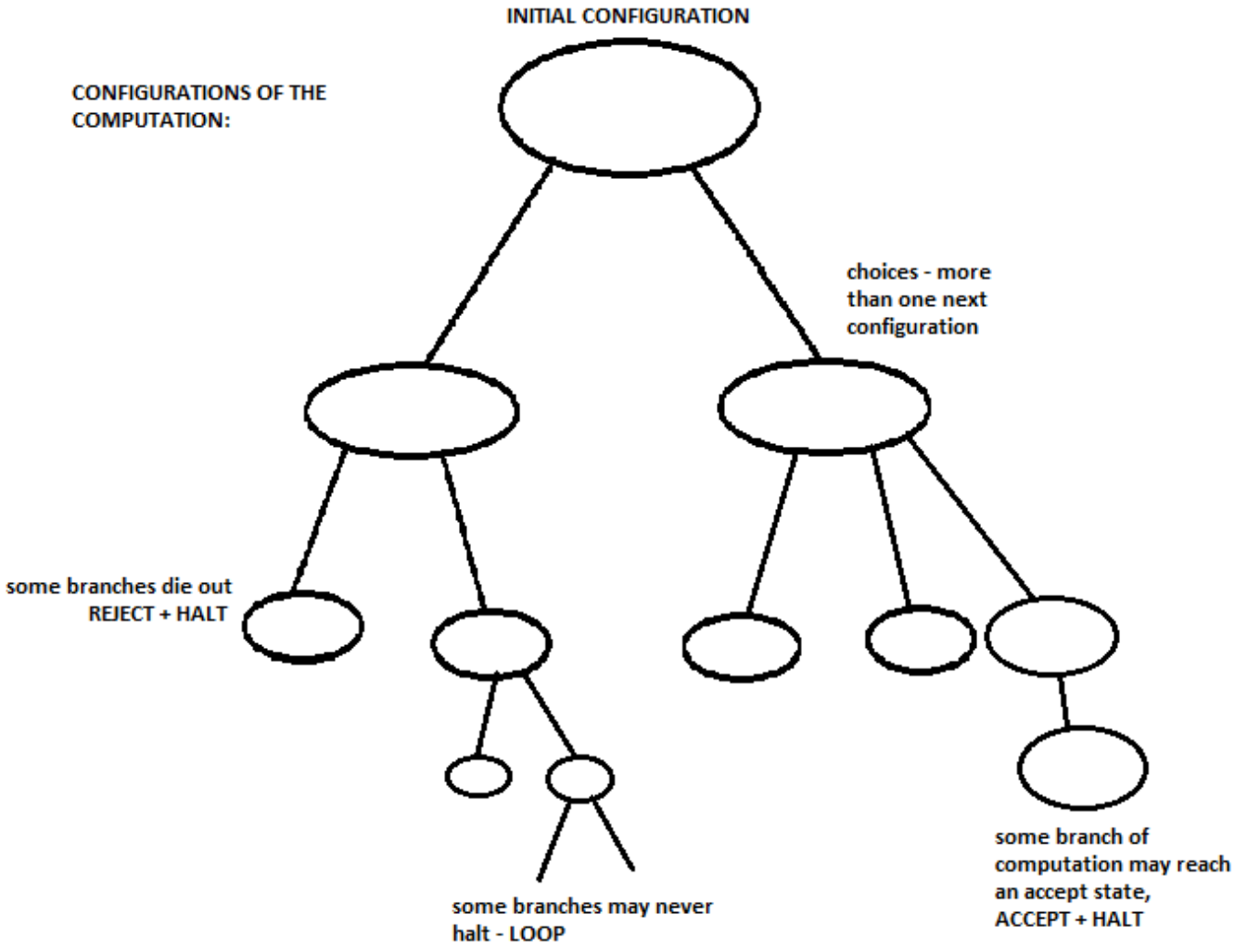
Non-deterministic TM:





Trees:

A tree shows the computation of a non-deterministic TM



Outcomes of non-deterministic computation:

ACCCEPT:

- If any branch of the computation accepts then the non-deterministic TM will ACCEPT.

REJECT:

- If all branches of the computation half and reject (i.e. no branches accept, but all computation halts) then the non-deterministic TM REJECTS.

LOOP:

- Computation continues but ACCEPT is never encountered
- Some branches in computation history are infinite

Theorem:

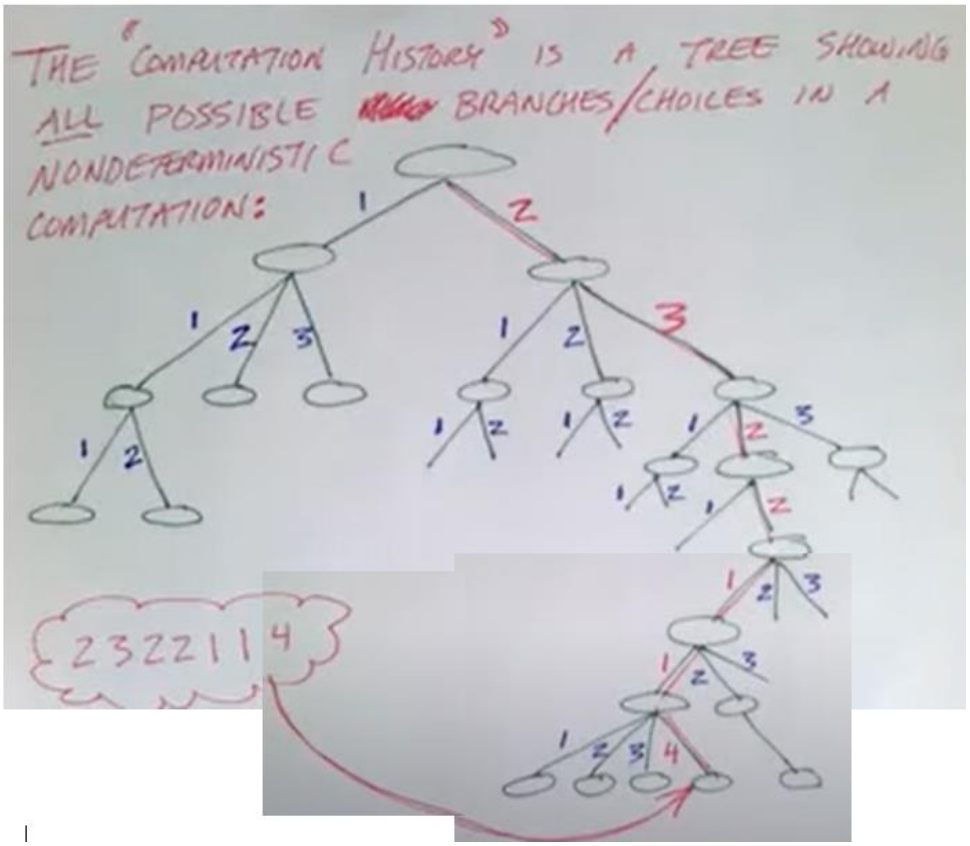
Every non-deterministic TM has an equivalent deterministic TM

Proof:

- Given a non-deterministic TM, **N**, show how to construct an equivalent deterministic TM, **D**
- If N accepts (on any branch) then D will accept
- If N halts on every branch without any ACCEPTS then D will halt and reject.

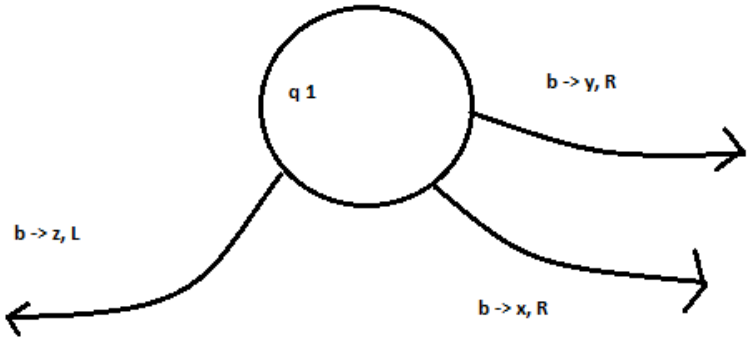
How do we approach this?

- Simulate N; simulate on all branches of computation; search for any way N can ACCEPT
- The path to any node is denoted by a number
- Search the tree looking for the ACCEPT using a bredth first seach
- Perform the entire computation from scratch
- The path numbers tell which of the many non-deterministic choices to make



Further example:

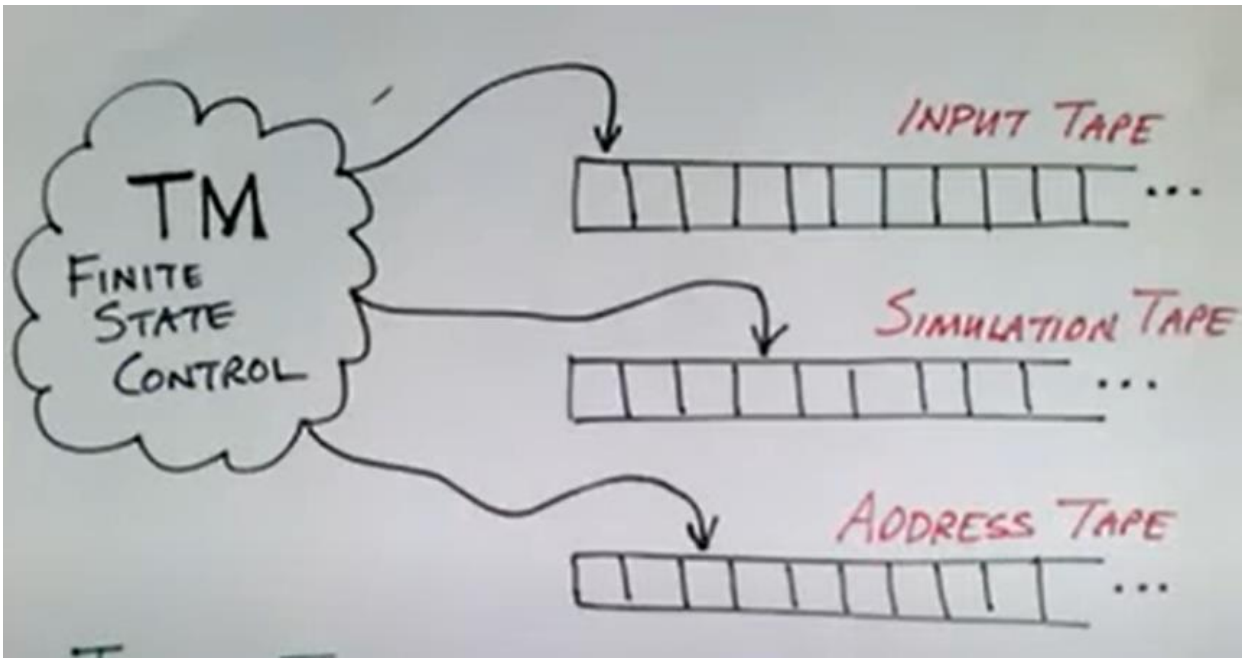
- How many choices at each step in the computation?
- Examine the Non-deterministic machine; there will be some maximum.



3

For this example:

- Any particular computation will have in it fewer than 3 choices at most of the computation steps.
- Some points will have zero choices; these branches of the ND HALT AND REJECT.



Input tape:

- Initial input; never modified

Simulation tape:

- Used like the tape of a Deterministic TM to perform the simulation

Address tape:

- Used to control the breadth-first search.
- Tells us which choice to make during a simulation

Algorithm:

Initially:

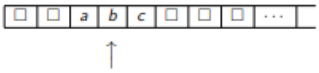
- TAPE 1 contain the input
- TAPE 2 and 3 are empty

1. Copy tape 1 to tape 2
2. Run the simulation
   a. Use tape 2 as 'the tape'
   b. When choices occur (i.e. when ND branch points are encountered) consult tape 3
   c. Tape 3 contains a 'path'. Each number is a choice to make
3. Run through the simulation all the way down the branch as far as the address/path goes (or the computation dies out)
4. Try the next branch by incrementing the address on tape 3
5. REPEEAT ALL STEPS

- If an ACCEPT is encountered, HALT + ACCEPT
- If all branches REJECT or die out, then HALT + REJECT

Turing machines with semi-infinite tape

A common variation of the standard TM is one where the tape unbounded in just one direction.



The TM with semi-infinite tape is identical to the standard model, except the tape head cannot move L when it is already at the tape boundary.

THE2 Lecture 19: Further Turing machines

Equivalence of standard and semi-infinite tape TMs

These classes of automata are equivalent, because any standard TM can be simulated by a semi-infinite tape TM:

- A standard (infinite) tape is simulated by a semi-infinite tape that has two **tracks**: upper track corresponds to characters to the right of some reference point, while lower track corresponds to those left of the reference point, in reverse order.
- The state space is partitioned into two parts: one part indicates the upper track should be worked on, and the other for the lower track. Special markers are put at the boundary of the tape to help with the transition from one track to the other.

DIAGRAM

How would we simulate $\delta(q_1, b, L) = (q_2, d)$?

THE2 Lecture 19: Further Turing machines

The off-line Turing machine

Our original description of an automaton showed an input file as well as temporary storage, but our TM definition merged the two, with input initially stored on the tape. We now show that this is a valid "simplification", with no effect on the expressive power of the TM. A TM with an additional read-only input file is known as an **off-line Turing machine**. Transitions are governed by current state, input and what's under the read-write head.

DIAGRAM

THE2 Lecture 19: Further Turing machines

Equivalence of standard and off-line TMs

A standard TM can be simulated by an off-line TM: the simulating machine starts by copying the input file to the tape, and then proceeds as the standard TM.

In the other direction, we need to show how an off-line TM can be simulated by a standard TM. For this we use a 4-track machine:

DIAGRAM

The first two tracks represent the input file's contents and position, and the other two represent the tape's contents and position.

A move in the off-line TM: first, from a standard position, it searches track 2 to find the read position in the input file. Then it stores that symbol by entering a specific state. Track 4 is searched to find the position of the read-write head. Using the stored input symbol and the symbol from track 3, the machine has the information to make its move; again this is stored in a state. Then tracks 2-4 are updated to reflect this move and the read-write head returns to its standard position ready for the next move.

THE2 Lecture 19: Further Turing machines

Multi-dimensional Turing machines

It's also possible to think of a tape extending in more than one dimension.

DIAGRAM of 2-D TM

Then the transition function for a 2-D TM could take the form

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, U, D\}$$

where U and D represent the tape head moving up or down.

Turing Machines as Problem Solvers

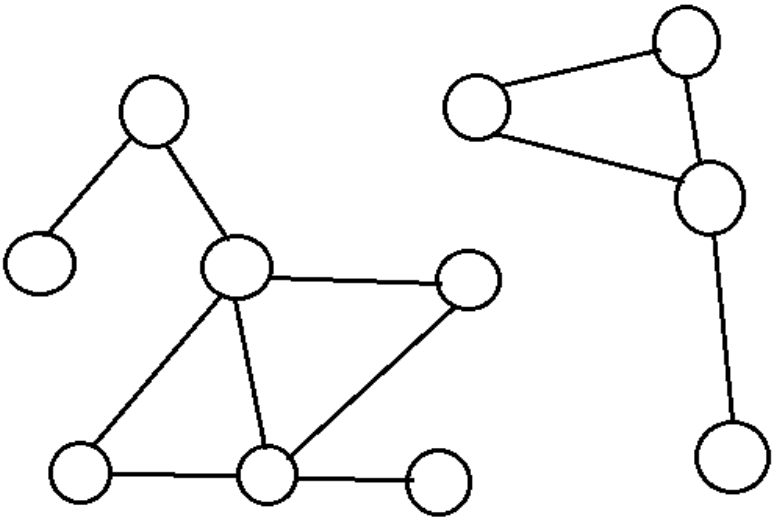
Any arbitrary problem can be expressed as a language

Any 'instance' of the problem is encoded into a string

- The string is in the language ≡ the answer is yes
- The string is not in the language ≡ the answer is no

Example – undirected graphs:

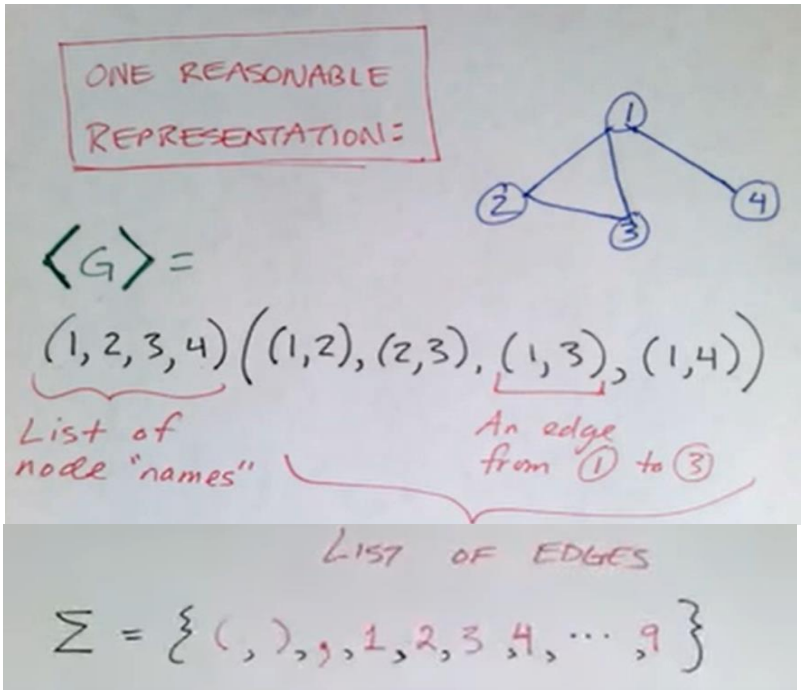
- Is this graph connected? We must encode the problem into a language!



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

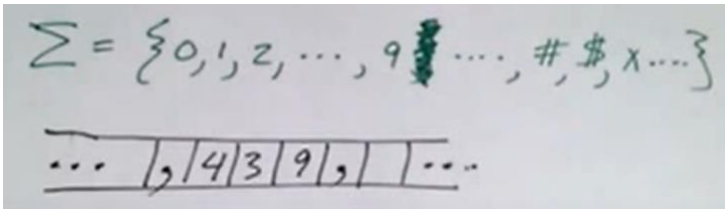
We must find a TM to DECIDE this language:

- ACCEPT = "YES", this is a connected graph
- REJECT = "NO", this is not a connected graph or this is not a valid representation of a graph
- LOOP = this problem is decidable, out TM will always halt.

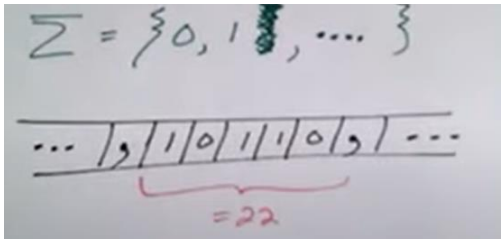


Representing numbers on a tape:

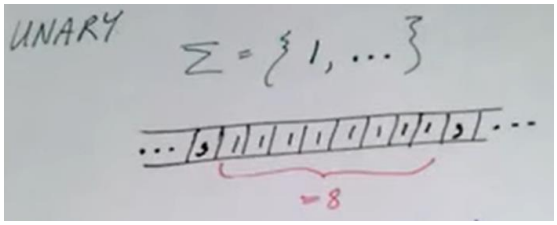
DECIMAL



BINARY

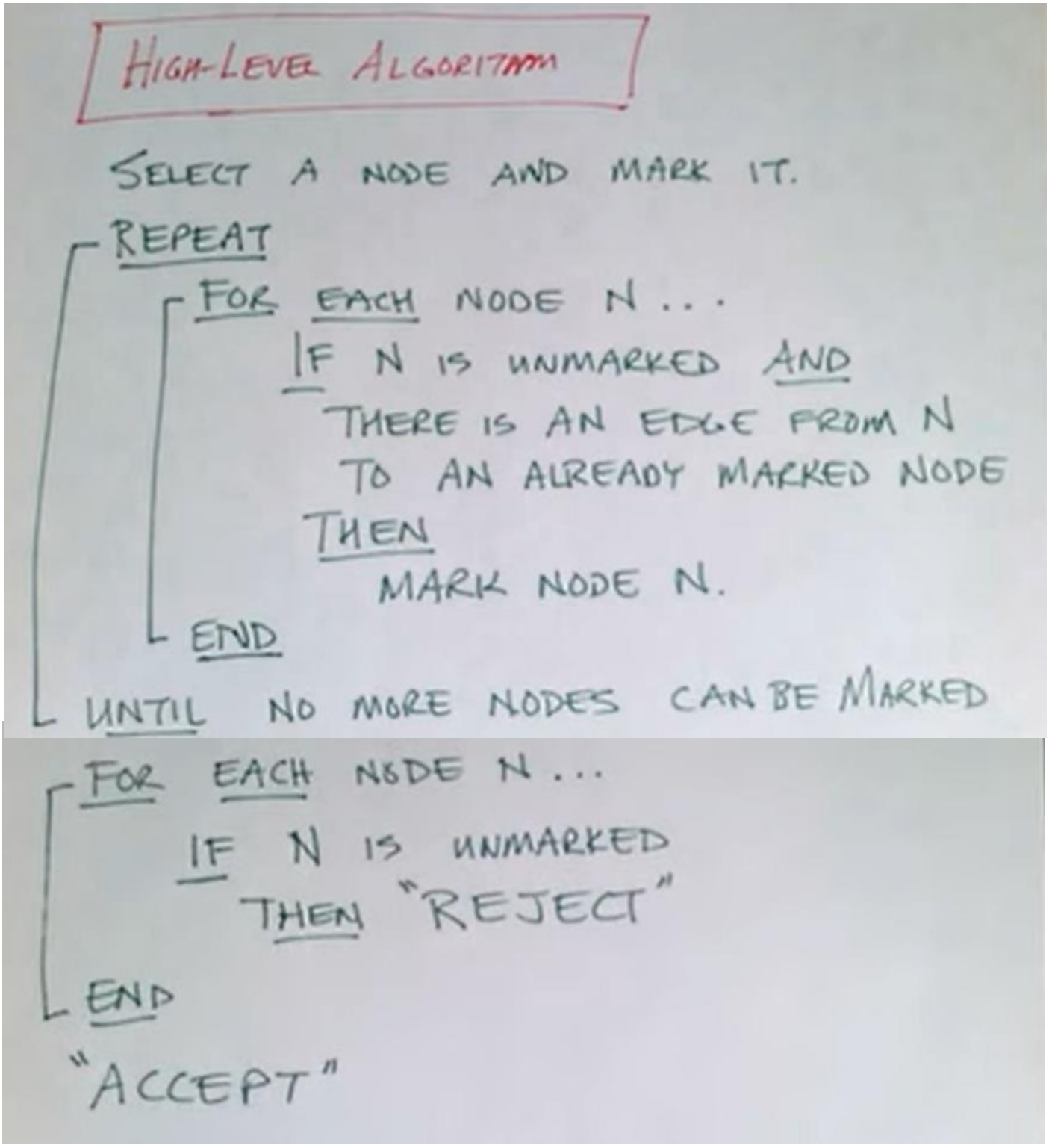


UNARY



Algorithm = Turing Machine

- High level specification
  - Pseudo code
  - Expressed in programming languages
- Implementation level
  - Contents of the tape
  - Data representation
  - Motion of the tape head
  - more detail, but still abstract
- TM specification
  - States
  - Alphabets
  - Transition function
  - Fully detailed (possibly incomprehensible)



implementation program:

- Check that input describes a valid graph
- Check node list
  - Scan C followed by digit,…
  - Check that all nodes are different and there are no repeats

1. Mark all first node
2. Place a dot under the first node in the node list
3. Scan the node list to find a node that is not marked

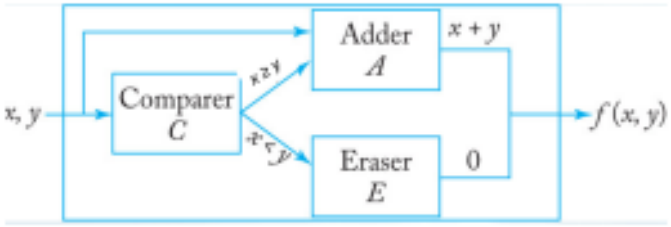
Etc…

Combining Turing Machines

Suppose we want a TM to compute:

$$\begin{aligned} f(x,y) &= x+y \quad \text{if } x \geq y \\ &= 0 \quad \text{if } x < y \end{aligned}$$

where x and y are positive integers (in unary representation, w(x), separated by 0 on the tape). In lecture 18, we outlined a TM to add integers. Can we use this, along with a ''comparison" TM and an ''eraser" TM to compute f ?



Compare TMs:

We want the TM to achieve:

- halt in final state q_f if $x \geq y$
- halt in non-final state q_n if $x < y$

So the computations are:

- $q_0 w(x)0w(y) \vdash^* q_f w(x)0w(y)$ if $x \geq y$
- $q_0 w(x)0w(y) \vdash^* q_n w(x)0w(y)$ if $x < y$

We can achieve this by matching digits of x with digits of y, and replacing them with X. We will end up with:

- $XX..110XX..X\square$ if $x \geq y$
- $XX..X0X..X11\square$ if $x < y$

In the first case the TM goes to state q_f, and in the second case to q_n. We also need to put all the Xs back to 1s.

Erase TMs:

- Change each 1 to a blank

Putting TMs together:

- Label the TM states to identify the sub-machines and avoid confusion: $q_{C,0}$, $q_{A,0}$

Then we adjust the submachine computations to reflect this name change, and link the states:

Compare:

- $q_{C,0} w(x)0w(y) \vdash^* q_{A,0} w(x)0w(y)$ if $x \geq y$
- $q_{C,0} w(x)0w(y) \vdash^* q_{E,0} w(x)0w(y)$ if $x < y$

Adder:

- $q_{A,0} w(x)0w(y) \vdash^* q_{A,f} w(x+y)0$

Erase:

- $q_{E,0} w(x)0w(y) \vdash^* q_{E,f} 0$

Enumeration

Recursive and recursively enumerable languages

- A language L is recursively enumerable (Type 0) iff there is a TM that accepts it. (Remember that strings are rejected if the machine doesn't halt, or halts in a non-final state.)
- A language L is recursive iff there is a TM that accepts it which is guaranteed to halt on every valid input string (ie all of $\Sigma*$).
- ie L is recursive iff there is a membership algorithm for it

Non-recursive languages

Theorem: *For any non-empty alphabet Σ, there are languages that are not recursively enumerable.*

Proof: See Linz (or other textbooks) for diagonalization argument, based on the fact that $L \subseteq \Sigma *$

[So there are more languages than Turing machines.]

Theorem: *The recursive languages are a strict subset of the recursively enumerable languages*

Unrestricted grammars:

- An unrestricted grammar has no restrictions at all on its productions (except that λ cannot appear as LHS):
- any combination of terminals and non-terminals on LHS any combination of terminals and non-terminals on RHS For example:

$$
\begin{array}{ll}
S & \rightarrow S_1 B \\
S_1 & \rightarrow a S_1 b \\
bB & \rightarrow bbbB \\
aS_1 b & \rightarrow aa \\
B & \rightarrow \lambda
\end{array}
$$

What language is generated?

Unrestricted grammars and recursively enumerable languages

- Every language generated by an unrestricted grammar is recursively enumerable
  - enumerate strings, by length of derivation
- Every recursively enumerable language (accepted by TM) can be generated by an unrestricted grammar
  - complex and lengthy construction, showing how TM allows us to construct grammar (cf NPDA to CFG)

So the unrestricted grammars generate exactly the recursively enumerable languages, which is the biggest family of languages that can be generated or recognized algorithmically.

- non-contracting: the working string of terminals and non-terminals cannot reduce in length at any point
- context-sensitive: we can specify that non-terminals should only be replaced in certain contexts
  - eg A should only be replaced if followed by b or c

Context sensitive languages

- A language L is context-sensitive (Type 1) iff there exists a context-sensitive grammar G such that L = L(G) or L = L(G) ∪ {λ} • λ is included because, by definition, a context-sensitive grammar can never generate a language which includes λ. The family of context-free languages is a (strict) subset of the family of context-sensitive languages

Context sensitive language: example

$L = \{a^n b^n c^n \mid n \geq 1\}$ is a context-sensitive language, because it is generated by the grammar we had earlier:

$$\begin{aligned} S &\to abc \mid aAbc \\ Ab &\to bA \\ Ac &\to Bbcc \\ bB &\to Bb \\ aB &\to aa \mid aaA \end{aligned}$$

For instance:

$$\begin{aligned} \underline{S} &\Longrightarrow a\underline{Ab}c \Longrightarrow ab\underline{Ac} \\ &\Longrightarrow a\underline{bB}bcc \Longrightarrow a\underline{B}bbcc \\ &\Longrightarrow aabbcc \end{aligned}$$

Effectively, A and B acts as messengers, moving through the string as it develops.

Context sensitive languages and Linear bounded automata

A **Linear bounded automaton** (LBA) is similar to a standard Turing machine, but with a restriction that it can only use the part of the tape that is orginally used for the input.
One way to think about it is that the ends of the input are marked with special markers, say [and], which cannot then be moved, passed or overwritten.
For example, consider the TM described previously for $L = \{a^n b^n\}$, where input symbols are examined and overwritten, but no extra tape space is used: this is an LBA. (A similar idea could be used for an LBA to accept $a^n b^n c^n$.)
Theorem: For every context-sensitive language not including λ, there exists a linear bounded automaton that accepts it.
Theorem: If L is accepted by an LBA, then there is a context-sensitive grammar that generates L.
So, the context-sensitive grammars generate exactly the context-sensitive languages, ie those languages accepted by LBAs

Recursive and Context sensitive languages

- Every context-sensitive language is recursive
- Some recursive languages are not context-sensitive
- Ordering of expressive power of automata:
  - LBAs are less powerful than Turing machines LBAs are more powerful that NPDAs

Reminder of the Chomsky hierarchy

Type	Grammar/Language	Grammar Constraints	Machine
0	Recursively enumerable	$\alpha \to \beta$	Turing machines
1	Context-sensitive	$\alpha A \gamma \to \alpha \beta \gamma$	Linear-bounded automata
2	Context-free	$A \to \beta$	Non-deterministic pushdown automata
3	Regular	$A \to aB$	Finite state automata

Remember length constraint for type-1 grammars