Turing machine description

\[ TM \approx DFA + \text{infinite tape} \]

Input is written on the tape starting at index 0.
All other cells contain the blank symbol \( \square \).
There is a tape pointer/head (initially at position 0), can move left or right.
You can read & write to the tape cell pointed to.

At each step, you have to:
- write a new symbol to the cell under the head
- move tape head Left or Right
- go to a new state

So a TM is a sequence of steps (states), each looking like:

**STATE 0:**
**switch** (letter under the head):
- case ‘a’: write ‘b’; move Left; go to STATE 2;
- case ‘b’: write ‘\( \square \)’; move Right; go to STATE 0;
- case ‘\( \square \)’: write ‘b’; move Left; go to STATE 1;
Formal definition: Turing machine

A Turing machine (TM) \( M \) is a 7-tuple
\[
M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})
\]
where
- \( Q \) is a finite set (which we call the set of states);
- \( \Sigma \) is a finite set with \( \square \notin \Sigma \) (which we call the input alphabet);
- \( \Gamma \) is a finite set with \( \square \in \Gamma \) and \( \Sigma \subset \Gamma \) (which we call the tape alphabet);
- \( \delta \) is a function \( \delta : Q \setminus \{ q_{\text{acc}}, q_{\text{rej}} \} \times \Gamma \to Q \times \Gamma \times \{ L, R \} \) (which we call the transition function);
- \( q_0 \in Q \) (which we call the start state);
- \( q_{\text{acc}} \in Q \) (which we call the accept state);
- \( q_{\text{rej}} \in Q \), \( q_{\text{rej}} \neq q_{\text{acc}} \) (which we call the reject state);

Rigorous defn. of TM accepting a string

A bit more involved.

Not too much though.

See course notes.

Language accepted by the TM

The machine accepts a string \( x \) if and only if:
\[
|x| > 1 \text{ and } x[0] = x[1]
\]
DFAs vs TMs
- A DFA does not have access to tape cells that don’t contain the input. (doesn’t have access to unbounded memory)
- A DFA’s tape head can only move right.
- A DFA can’t write to the tape.
- A DFA can have more than one accepting state.
- A DFA always halts once all the input symbols are read. A TM might loop forever!!

Definition: decidable/computable languages
Let $M$ be a Turing machine.
We let $L(M)$ denote the set of strings that $M$ accepts.
So, $L(M) = \{ x \in \Sigma^* : M(x) \text{ accepts.} \}$

What is the analog of regular languages in this setting?

Definition: A TM is called a decider if it halts on all inputs.

Definition: A language $L$ is called decidable (computable) if $L = L(M)$ for some decider TM $M$. 

regular languages = decidable languages
Turing machine that decides $0^n \mid n$

$\Sigma = \{0, 1\}$
$\Gamma = \{0, 1, \#, \square\}$

Input: 00001011
Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011
Turing machine that decides $0^n 1^n$

Input: 0001011

Turing machine that decides $0^n 1^n$

Input: 0001011

Turing machine that decides $0^n 1^n$

Input: 0001011

Turing machine that decides $0^n 1^n$

Input: 0001011
Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011
Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011
Turing machine that decides $0^n1^n$

Input: 00001011
Turing machine that decides $0^n1^n$

Input: 0001011

Turing machine that decides $0^n1^n$

Input: 0001011

Turing machine that decides $0^n1^n$

Input: 0001011

Turing machine that decides $0^n1^n$

Input: 0001011
Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011

Turing machine that decides $0^n1^n$

Input: 00001011
Turing machine that decides $0^n 1^n$

Input: 00001011

Turing machine that decides $0^n 1^n$

Input: 00001011

Turing machine that decides $0^n 1^n$

Input: 00001011

Turing machine that decides $0^n 1^n$

Input: 00001011

Turing machine that decides $0^n 1^n$

Input: 00001011

Turing machine that decides $0^n 1^n$

Input: 00001011

Turing machine that decides $0^n 1^n$

Input: 00001011
Turing machine that decides $0^n1^n$

Input: 00001011
Poll

Suppose we change the definition of a TM so that the head stays put instead of moving left. That is, the transition function becomes

\[ \delta : Q \setminus \{ q_{\text{acc}}, q_{\text{rej}} \} \times \Gamma \rightarrow Q \times \Gamma \times \{ S, R \} \]

What is power of the resulting model compared to normal TMs (in terms of class of languages decided)?

- More powerful
- Less powerful
- Equal in power
- Beats me

Programming with a TM is tiresome.

Every computer scientist should spend some time doing it at least once in their life.

Unfortunately for you, that time is now! (but only very briefly…)

A note

You could describe a TM in 3 ways:

Low level description
   Describe all states and transitions.

Medium level description
   Careful English description of how the TM operates
   Should be ‘clear’ this can be translated into low level
   description with enough pain.

High level description
   Pseudocode or algorithm.
   Skip ‘standard’ bookkeeping details

Some TM subroutines and tricks

- Move right (or left) until first ¶ encountered
- Shift entire input string one cell to the right
- Convert input from

\[ x_1 x_2 x_3 \ldots x_n \] to

\[ \square x_1 \square x_2 \square x_3 \ldots \square x_n \]

- Simulate a big \( \Gamma \) by just \( \{0, 1, \square\} \)
- “Mark” cells. If \( \Gamma = \{0, 1, \square\} \), extend it to

\[ \Gamma = \{0, 1, 0, 1, \square\} \]

- Copy a stretch of tape between two marked cells
  into another marked section of the tape

Some TM subroutines and tricks

- Implement basic string and arithmetic operations
- Simulate a TM with multiple tapes and heads
- Implement basic data structures
- Simulate “random access memory”

\[ \vdots \]

- Simulate assembly language

You could prove all this rigorously if you wanted to.

\{0^n : n \in \mathbb{N}\} is decidable

Medium Level description:

1. Sweep from left to right across the tape,
   overwriting a # over top of every other 0.
2. If you saw one 0 on the sweep, accept.
3. If you saw an odd number of 0’s, reject.
4. Move back to the leftmost square.
   (Say you write \( \sqcup \) on the leftmost square at the
   very beginning so that you can recognize it later.)
5. Go back to step 1.
QUESTIONS

1. Is TM the right definition?

2. Are there languages that are not decidable?

3. What was Turing’s and others’ motivation?

2 of Hilbert’s Problems

Hilbert’s 10th problem (1900)
Is there a finitary procedure to determine if a given multivariate polynomial with integral coefficients has an integral solution?

\[ 5x^2yz^3 + 2xy + y - 99xyz^4 = 0 \]

Entscheidungsproblem (1928)
Is there a finitary procedure to determine the validity of a given logical expression?

\[ \neg \exists x, y, z \in \mathbb{N} : (n \geq 3) \land (x^n + y^n = z^n) \]

(Mechanization of mathematics)

What was Hilbert really asking for?

“computers” in early 20th century

The quest for the right definition

“Alright, let’s define this thing mathematically.”
Turing’s thinking

- A (human) computer writes symbols on paper.
  (can view the paper as a sequence of squares)
- No upper bound on the number of squares.
- Human can reliably distinguish finitely many shapes.
- Human observes one square at a time.
- Human has finitely many mental states.
- Human can change symbol,
  can change focus to neighboring square,
  based on its state and the symbol it observes
- Human acts deterministically.
- ...

Turing’s legacy

The beauty of his definition:

1. simplicity

2. “clearly” captures what a human does given a set of instructions.

Simplicity

I. simplicity

a reasonable definiton of computation

strong enough to capture computation the way TMs do.

(Anyone who attempted to define computation could accidentally hit a correct definition)

Generality

2. “clearly” captured what a human does given a set of instructions.

Church-Turing Thesis

The intuitive notion of “computable” is captured by functions computable by a Turing Machine.
What else did Turing do in his paper?

There are languages that cannot be computed!

**Entscheidungsproblem**
Determining the validity of a given FOL sentence.

*e.g.* $\neg \exists x, y, z, n \in \mathbb{N} : (n \geq 3) \land (x^n + y^n = z^n)$

**Not decidable!**

**Halting problem**
Determining if a given TM halts on all inputs.
(i.e. determining if a given TM is a decider.)

**Not decidable!**

What else did Turing do in his paper?

**Universal Machine**
(one machine to rule them all)

+ isPrime Sorting DFA “|x| even?”

Do we really need a separate machine for each task?

What else did Turing do in his paper?

**Universal Machine**
(one machine to rule them all)

input data \( \rightarrow \) Universal Machine \( \rightarrow \) output data

A human is a universal machine:

input data \( \rightarrow \) output data
What else did Turing do in his paper?

**Universal Machine**
(one machine to rule them all)

All can be encoded!
(e.g. think source code)

Code is data!

Universal Machine
(one machine to rule them all)

Example:

```
def foo(input):
    i = 0
    STATE 0:
    letter = input[i];
    switch(letter):
        case 'a': input[i] = ' '; i++; go to STATE a;
        case 'b': input[i] = ' '; i++; go to STATE b;
        case ' ': input[i] = ' '; i++; go to STATE rej;
    STATE a:
    letter = input[i];
    switch(letter):
        case 'a': input[i] = ' '; i--; go to STATE acc;
        case 'b': input[i] = ' '; i--; go to STATE rej;
        case ' ': input[i] = ' '; i--; go to STATE rej;
```

Could you write a Python function that does this? Yes!
Then there is a TM that does this as well.
**Code is data!**

**The positive side**
- Universal TM
- Stored program computer

**The negative side**
- Self-referencing
  
  (can feed a machine its own code as input.)

→ Undecidability

---

**Physical Church-Turing Thesis**

**Strong Physical Church-Turing Thesis**

- What can be computed efficiently in this universe, by any physical process or device, can be computed efficiently by a TM.

---

**What else did Turing do in his paper?**

**Universal Machine**
(one machine to rule them all)

This is exactly what an **interpreter** does.

---

**Perhaps Turing and others weren’t ambitious enough!**

- Solvable by any physical process
- Solvable by a TM

- universe and the laws of physics

- Self-referencing
  
  (can feed a machine its own code as input.)

→ Undecidability

---

**Python Interpreter**

- output of the Python program on input x
Physical Church-Turing Thesis

What can be computed in this universe, by any physical process or device, can be computed by a (rand.) TM.

Why should we expect this to be true?

Strong Physical Church-Turing Thesis

What can be computed efficiently in this universe, by any physical process or device, can be computed efficiently by a QTM.

This is the grand unification/simplification of computation!!

Complex things can be explained by simple rules.
- physics: try to find the simple rules that give rise to the universe
- evolution: complex life forms emerge from simple beginnings and rules
- math: complex proofs arise by combining very simple deductive rules
- programming: everything boils down to very simple instructions

Implications

1. Studying the power and limits of TMs
   Studying the power and limits of computation in our universe

2. Computation in its full generality is everywhere.
   Even in extremely simple systems!

3. The universe may be a simulation. (a philosophical musing)
Conway's Game of Life (A cellular automaton)

Imagine an infinite 2D grid.

Each cell can be dead or alive.

**Evolution Laws**

Loneliness: live cell with fewer than 2 neighbors dies.
Overcrowding: live cell with more than 3 neighbors dies.
Procreation: dead cell with exactly 3 neighbors gets born.

Conway's Game of Life

**Some Patterns**

- Stable
- Periodic
- Moving

Conway’s Game of Life

Can a TM simulate any instance of Game of Life?
Can Game of Life simulate any TM?
Can Game of Life simulate Game of Life?

That's all for the Church-Turing Thesis.
Let’s talk decidability.
Languages involving encodings of machines

*Code is data!*

There are many interesting problems where the input data is code.

---

**Working as a TA for 15-112**

We need to write an autograder for `isPrime`

Student submission

the correct program

`isPrime`

Do they return True on exactly the same inputs?

---

**Working as a TA for 15-112**

We need to write an autograder for `isPrime`

Kosbie’s version

returns True on exactly same inputs?

True or False

Student submission

Does such a program exist?

i.e., can we solve/decide the following?

\[
EQ = \{ (M_1, M_2) : M_1 \text{ and } M_2 \text{ are TMs s.t. } L(M_1) = L(M_2) \}
\]
Working as a TA for 15-112

Similar but simpler looking languages:

ACCEP'TS = \{ (M, x) : M \text{ is a TM and } x \in \Sigma^* \text{ s.t. } x \in L(M) \}

EMPTY = \{ (M) : M \text{ is a TM s.t. } L(M) = \emptyset \}

Grading DFAs

ACCEP'TS_{DFA} = \{ (D, x) : D \text{ is a DFA and } x \in \Sigma^* \text{ s.t. } x \in L(D) \}

EMPTY_{DFA} = \{ (D) : D \text{ is a DFA s.t. } L(D) = \emptyset \}

EQ_{DFA} = \{ (D_1, D_2) : D_1 \text{ and } D_2 \text{ are DFAs s.t. } L(D_1) = L(D_2) \}

ACCEP'TS_{DFA}

ACCEP'TS_{DFA} is decidable.

A TM can simulate a DFA.
(special case of Universal TM)

EMPTY_{DFA}

EMPTY_{DFA} is decidable.

- Build a directed graph from D
- Run a graph search algo. starting from the start state of D.
- If ever visit an accept state of D, reject;
  else when no new states are visited, accept.
**EQ\text{DFA}**

\[ \text{EQ\text{DFA}} = \{ (D_1, D_2) : D_1 \text{ and } D_2 \text{ are DFAs s.t. } L(D_1) = L(D_2) \} \]

is decidable.

\[ L(D_1) = L(D_2) \text{ if and only if } (L(D_1) \cap \overline{L(D_2)}) \cup (\overline{L(D_1)} \cap L(D_2)) = \emptyset \]

Build a DFA for latter language, and check if it accepts the empty language.

**Reduction**

Solving \text{EQ\text{DFA}} reduces to solving \text{EMPTY\text{DFA}}.

\[ \text{EQ\text{DFA}} \text{ reduces to } \text{EMPTY\text{DFA}} \]

\[ \text{EQ\text{DFA} } \leq \text{ EMPTY\text{DFA}} \]

\[ \text{EMPTY\text{DFA}} \text{ decidable } \implies \text{EQ\text{DFA}} \text{ decidable.} \]

We write \( L \leq K \), if we can solve \( L \) using a decider for \( K \) as a helper function.

Reductions expand the landscape of decidable languages.

**COMING NEXT**

Turing’s Legacy Continues

Undecidability