Formal definition: DFA

A deterministic finite automaton (DFA) $M$ is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where
- $Q$ is a finite, non-empty set (which we call the set of states);
- $\Sigma$ is a finite, non-empty set (which we call the alphabet);
- $\delta$ is a function of the form $\delta : Q \times \Sigma \rightarrow Q$ (which we call the transition function);
- $q_0 \in Q$ is an element of $Q$ (which we call the start state);
- $F \subseteq Q$ is a subset of $Q$ (which we call the set of accepting states).

Formal definition: DFA accepting a string

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. $\delta : Q \times \Sigma \rightarrow Q$ can be extended to $\delta^* : Q \times \Sigma^* \rightarrow Q$ as follows:

for $q \in Q, w \in \Sigma^*$, 
$\delta^*(q, w) =$ state we end up in when we start at $q$ and read $w$

In fact, even OK to drop * from the notation.

$M$ accepts $w$ if $\delta(q_0, w) \in F$.
Otherwise $M$ rejects $w$.

Definition: Regular languages

Definition: A language $L$ is called regular if $L = L(M)$ for some DFA $M$. 
Regular languages

Closed properties of regular languages

**Proposition:**
Let $\Sigma$ be some alphabet.
If $L \subseteq \Sigma^*$ is regular, then so is $\overline{L} = \Sigma^* \setminus L$.

**Theorem:**
Let $\Sigma$ be some alphabet.
If $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ are regular, then so is $L_1 \cup L_2$.

**Corollary:**
Let $\Sigma$ be some finite alphabet.
If $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ are regular, then so is $L_1 \cap L_2$.

Closed under intersection

Closure properties can be used to show languages are not regular.

**Example:**
Let $L \subseteq \{0, 1\}^*$ be the language consisting of all words with an equal number of 0's and 1's.

We claim $L$ is not regular. Suppose it was regular.

\[
\{0^n1^m : n, m \in \mathbb{N}\} \cap L = \{0^n1^n : n \in \mathbb{N}\}
\]

\[\text{regular} \quad \text{regular} \quad \text{regular} \quad \text{contradiction} \]

Closed under concatenation (more tricky)

\[
L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}
\]

**Theorem:**
Let $\Sigma$ be some alphabet.
If $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ are regular, then so is $L_1L_2$.

Similarly, if $L \subseteq \Sigma^*$ is regular, then so is $L^*$.
The mindset

Imagine yourself as a DFA.

Rules:
1) Can only scan the input once, from left to right.
2) Can only remember “constant” amount of information. should not change based on input length.

Given \( w \in \Sigma^* \), we need to decide if \( w = uv \) for \( u \in L_1 \), \( v \in L_2 \).

Problem: Don’t know where \( u \) ends, \( v \) begins.

When do you stop simulating \( M_1 \) and start simulating \( M_2 \)?

Suppose God tells you \( u \) ends at \( w_3 \).

\[
\begin{array}{cccccccccc}
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
\hline
w_1 & w_2 & w_3 & w_4 & w_5 & w_6 & w_7 & w_8 & w_9 & w_{10} \\
q_0 & q_1 & q_1 & q_3 & q_0 & q_2 & q_2 & q_1 & q_2 & q_1 \hline
q_0 & q_1 & q_1 & q_3 & q_0 & q_2 & q_2 & q_1 & q_2 & q_1
\end{array}
\]

thread: a simulation of \( M_1 \) and then \( M_2 \) that corresponds to breaking up input \( w \) as \( uv \) where \( u \in L_1 \).
Regular languages, recursively

Define a class of languages inductively as follows:

- $\emptyset$ is regular.
- For every $a \in \Sigma$, \{a\} is regular.
- $L_1, L_2$ regular $\implies L_1 \cup L_2$ regular.
- $L_1, L_2$ regular $\implies L_1 \cdot L_2$ regular.
- $L$ regular $\implies L^*$ regular.

**Theorem:** Above precisely defines regular languages

Note: we saw one direction, that any language thus defined is regular. Other direction skipped.
Regular languages and expressions

Define a class of languages inductively as follows:

- \( \emptyset \) is regular.
- For every \( a \in \Sigma \), \( \{a\} \) is regular.
- \( L_1, L_2 \) regular \( \implies L_1 \cup L_2 \) regular.
- \( L_1, L_2 \) regular \( \implies L_1 \cdot L_2 \) regular.
- \( L \) regular \( \implies L^* \) regular.

Any such language described by a regular expression, such as:
\[ a(a \cup b)^* a \cup b(a \cup b)^* b \cup a \cup b \]

An application of DFAs

String Searching Problem

Input: string \( T \) of length \( n \). string \( w \) of length \( k \).
Output: Yes/No. Does \( w \) occur in \( T \)?

Naive algorithm:

About \( nk \) steps.
Can we do better?

Automaton solution:

The language \( \Sigma^* w \Sigma^* \) is regular.

So there is some DFA \( M_w \) that accepts it.

Build \( M_w \) and feed it \( T \). Running time: \( \sim n \) steps.

Time to build \( M_w \)?
Simple alg: \( \sim k^3 \) steps.
Knuth-Morris-Pratt 1977: \( \sim k \) steps to build \( M_w \).
How can we mathematically define the general notion of an algorithm?

The properties we want from the definition:

- Generality! (general enough to capture all of computation)
- Simplicity! (the simpler the better)

Goal of remainder this lecture:
Define *Turing machines*.
Understand how they work.

Goal of next lecture:
Explore physical, philosophical, historical questions surrounding Turing machines, and their motivation and amazing implications.

Goal is to reach the definition of a Turing machine.
2 important observations:

1. The device should be a “finite object”.
   An algorithm should be a “finite object”.

2. The device should be able to use “unbounded memory”.
   (memory should scale with input size; there is always more space to work on, if needed)

An algorithm is a finite answer to infinite number of questions.

Solving $0^n \mid n$ in Python

```python
def foo(input):
    i = 0
    j = len(input) - 1
    while (j >= i):
        if (input[i] != '0' or input[j] != '1'):
            return False
        i = i + 1
        j = j - 1
    return True
```
Solving $0^n \mid n$ in C

```c
int foo(char input[]) {
    int i = 0, j;
    while(input[j] != NULL) /* NULL is end-of-string character */
        j++;
    j--;
    while(j >= i) {
        if(input[i] != '0' || input[j] != '1')
            return 0; /* Reject */
        i++;
        j--;
    }
    return 1; /* Accept */
}
```

Should we define **computable** to mean what is computable by a Python function/program?

Downsides as a formal definition?

- Why choose Python, why not C, Java, SML,...?
  Are these equivalent?
  solvable in Python = solvable in C?

- Extremely complicated to define rigorously.
  (even bytecode)
Should we define *computable* to mean what is computable by a Python function/program?

Downsides as a formal definition?

- Why choose Python, why not C, Java, SML, ...?
  Are these equivalent?
  solvable in Python = solvable in C?

- Extremely complicated to define rigorously.
  (even bytecode)

---

**So what we want is:**

A totally minimal (TM) programming language such that

- it can simulate simple bytecode
  (and therefore Python, C, Java, SML, etc...)

- it is simple to define and reason about completely mathematically rigorously

---

Actually TM™ stands for Turing machine.

Defined by Alan Turing in a paper he wrote in 1936 while he was a PhD student.
How did Turing think about all this?

1936: On Computable Numbers, with an Application to the Entscheidungsproblem

Any notion of “computation” must be able to be carried out by a “computer”.

At the time of writing, “computer” meant a person, trained in calculation.

Turing justified TMs by arguing that it can do anything a human could.

Turing’s mathematical abstraction of a computer

- A (human) computer writes symbols on paper
- WLOG, the paper is a sequence of squares
- No upper bound on the number of squares
- At most finitely many kinds of symbols
- Human observes one square at a time
- Human has only finitely many mental states
- Human can change symbols and change focus to a neighboring square, but only based on its state and the symbol it observes
- Human acts deterministically

Turing machine description

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

Input is written on the tape starting at index 0.

All other cells contain the blank symbol $\Box$.

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.

But, we want to keep the definition as simple as possible.
Turing machine description

**TM ≈ DFA + infinite tape**

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** ‘ ’; **move** Right; **go to** STATE 0;
  - case ‘ ’: **write** ‘b’; **move** Left; **go to** STATE 1;

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

Don’t want to change the symbol: **write** the same symbol.
Want to stay put: **move** Left then Right.
Don’t want to change state: **go to** the same state.

Input: aaba if you are in state q₀ and you read a, then write blank and move Right

Input: aaba
Turing machine simulation example

Input: aaba
Turing machine simulation example

**Input:** aaba

```
q0
a ↦ ⊥, R
b ↦ ⊥, R

qa
b ↦ ⊥, L
a ↦ ⊥, L

qb

qrej
⊕ ↦ ⊥, L
⊕ ↦ ⊥, L

qacc
a ↦ ⊥, L
b ↦ ⊥, L
```

**Decision:** Accept

Turing machine simulation example

**Input:** baaaaa

```
q0
a ↦ ⊥, R
b ↦ ⊥, R

qa
b ↦ ⊥, L
a ↦ ⊥, L

qb

qrej
⊕ ↦ ⊥, L
⊕ ↦ ⊥, L

qacc
a ↦ ⊥, L
b ↦ ⊥, L
```

Turing machine simulation example

**Input:** baaaaa

```
q0
a ↦ ⊥, R
b ↦ ⊥, R

qa
b ↦ ⊥, L
a ↦ ⊥, L

qb

qrej
⊕ ↦ ⊥, L
⊕ ↦ ⊥, L

qacc
a ↦ ⊥, L
b ↦ ⊥, L
```
Turing machine simulation example

Input: baaaaa

![Turing machine simulation diagram](image-url)
Turing machine simulation example

Input: baaaaa

- $q_0$: Initial state
- $q_a$: Accept state
- $q_b$: Rejected state

Transitions:
- $a \rightarrow \square, R$
- $b \rightarrow \square, R$
- $\square \rightarrow \square, L$
- $a \rightarrow \square, L$
- $b \rightarrow \square, L$

Transition matrix:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Turing machine simulation example

**Input:** baaaaa  
**Decision:** Reject

```
def M(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;
    ...;
```
def M(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;

STATE rej:
    i = i + 1;

TM as a programming language
def M(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;
    ...
TM as a programming language
def M(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;

Poll
The machine accepts a string x if and only if:
|x| > 1 and x[0] = x[1]
x has at least two a’s or two b’s.
x[0] ≠ x[1]
|x| > 1 and x[0] = x[1]                         None of these.
|x| < 2 or x[0] = x[1]                         Beats me.

Exercise
Let \( \Sigma = \{a, b\} \).
Draw the state diagram of a TM that accepts a string iff it starts and ends with an \( a \).

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

Rules of computation:
Tape starts with input $x \in \Sigma^*$, followed by infinite $\omega$'s.
Control starts in state $q_0$, head starts in leftmost square.
If the current state is $q$ and head is reading symbol $s \in \Gamma$,
the machine transitions according to $\delta(q,s)$, which gives:
the next state,
what tape symbol to overwrite the current square with,
and whether the head moves Left or Right.
Technicality: moving left from the leftmost square $\equiv$ staying put.
Continues until either the accept state or reject state reached.
When accept/reject state is reached, M halts.
M might also never halt, in which case we say it loops.

Rigorous defn. of TM accepting a string

A bit more involved.
Not too much though.
See course notes.

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!!

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 \textit{decider} if it halts on all inputs.

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

Regular languages = decidable languages

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

\[ \Sigma = \{0, 1\} \quad \Gamma = \{0, 1, \#, \downarrow\} \]

Input: 00001011

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

Input: 00001011

(Omitted information defined arbitrarily.
Missing transitions go to the reject state.)
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: 0001011

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: 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

Input: 00001011

Input: 00001011

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

Decision: reject
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

A note

Some TM subroutines and tricks

- Move right (or left) until first \(\square\) 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 2 tapes and heads
- Implement basic data structures
- Simulate “random access memory”
  
  You could prove this rigorously if you wanted to.

Important Question

Is TM the right definition?

Is there a reasonable definition of “algorithm” that can compute more languages than TM-decidable ones?

Solvable with any computing device


What else did Turing do in his paper?

Universal Machine

(one machine to rule them all)

+ isPrime Sorting DFA

|\{|x|\} even?

All can be encoded/represented with a string. (e.g. think source code)

Fix some alphabet \(\Sigma\).

We use the \(\langle , \rangle\) notation to denote the encoding of an object as a string in \(\Sigma^*\).

\(\langle M \rangle \in \Sigma^*\) is the encoding of a TM \(M\)
What else did Turing do in his paper?

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

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

What else did Turing do in his paper?

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

This is exactly what an interpreter does.

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

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

**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!**