15-251
Great Theoretical Ideas in Computer Science

Lecture 4:
Turing’s Legacy

September 8th, 2016
What is computation?

What is an algorithm?

How can we mathematically define them?
Let's assume two things about our world

No “universal” machines exist.

We only have machines to solve decision problems.
DFA: state diagram + input tape
DFA: state diagram + input tape
DFA: state diagram + input tape
DFA: state diagram + input tape

1 0 1 1 1 0 1 □ □ □ □ □ □ □ ...

\[
\begin{align*}
q_0 & \rightarrow 0 \rightarrow q_1 \\
q_1 & \rightarrow 1 \rightarrow q_2 \\
q_2 & \rightarrow 0,1 \rightarrow q_3 \\
q_3 & \rightarrow 0 \rightarrow q_0
\end{align*}
\]
DFA: state diagram + input tape
DFA: state diagram + input tape

[Diagram of a DFA with states q0, q1, q2, q3 and input tape 1011101]
DFA: state diagram + input tape
DFA: state diagram + input tape

Input tape: 1 0 1 1 1 0 1 0

State diagram:
- $q_0$: 0, 1
- $q_1$: 0, 1
- $q_2$: 0, 1
- $q_3$: 0, 1

Transitions:
- $q_0$ to $q_1$ on 0
- $q_1$ to $q_2$ on 0, 1
- $q_2$ to $q_3$ on 1
- $q_3$ to $q_0$ on 0
DFA: state diagram + input tape

Decision: Accept
def foo(input):
    i = 0;

    STATE 0:
    if (i == input.length): return False;
    letter = input[i];
    i++;
    switch(letter):
        case '0': go to STATE 0;
        case '1': go to STATE 1;

    STATE 1:
    if (i == input.length): return True;
    letter = input[i];
    i++;
    switch(letter):
        case '0': go to STATE 2;
        case '1': go to STATE 2;

    ...
What is computation?

What is an algorithm?

How can we mathematically define them?

The properties we want from the definition:

Simplicity! (the simpler the better)

Generality! (general enough to capture all of computation)
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”.

   An algorithm is a finite answer to infinite number of questions.
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 “unlimited memory”. (there is always more space to work on, if needed) Wouldn’t be mathematically natural otherwise.
Regular languages

EvenLength

Factoring

$0^n \mid n$

isPrime

Solvable with any computing device
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
    return True
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 */
}
Solvable with Python?

- Regular languages
  - EvenLength
  - ...
- Factoring
  - $0^n \mid n$
  - isPrime
  - ...

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 \textit{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.
Turing machine description

TM ≈ DFA + infinite tape

Input is written on the tape starting at index 0.

All other cells contain the *blank* symbol □.

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.
Turing machine description

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

TM could have been defined as a sequence of instructions, where the allowed instructions are:

\[ > \text{Move the head left} \]
\[ > \text{Move the head right} \]
\[ > \text{Write a symbol } a \text{ (from the alphabet)} \]
\[ > \text{If head is reading symbol } a, \text{ goto step } j \]
\[ > \text{Halt and accept} \]
\[ > \text{Halt and reject} \]

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

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

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;
Turing machine description

**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 0 and you read a, then write blank and move Right.
Input: aaba

- Turing machine official picture

- Transition diagram:
  - States: $q_0$, $q_a$, $q_{rej}$, $q_{acc}$, $q_b$
  - Transitions:
    - $q_0$: $a \rightarrow \square, R$
    - $q_{rej}$: $a \rightarrow \square, R$
    - $q_{acc}$: $b \rightarrow \square, L$
    - $q_{acc}$: $a \rightarrow \square, L$
    - $q_{acc}$: $b \rightarrow \square, L$
    - $q_a$: $b \rightarrow \square, L$
    - $q_{rej}$: $a \rightarrow \square, L$
    - $q_{rej}$: $b \rightarrow \square, L$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
    - $q_{rej}$: $\square \rightarrow \square, R$
Turing machine simulation example

Input: aaba

Diagram:

- States: $q_0$, $q_a$, $q_{rej}$, $q_{acc}$, $q_b$
- Transitions:
  - $q_0$: $a \rightarrow \square, R$ and $b \rightarrow \square, R$
  - $q_a$: $b \rightarrow \square, L$
  - $q_{rej}$: $a \rightarrow \square, L$
  - $q_{acc}$: $a \rightarrow \square, L$ and $b \rightarrow \square, L$
  - $q_b$: $\square \rightarrow \square, L$
Turing machine simulation example

Input: aaba
Turing machine simulation example

Input: aaba
Input: aaba
Turing machine simulation example

Input: aaba
Turing machine simulation example

Input: aaba

```
Input:  aaba

<table>
<thead>
<tr>
<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>□</td>
<td>□</td>
<td>□</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>
```

```
Graph with states and transitions:

- **q0**: 
  - **q1**: $a \to \square, R$
  - **q2**: $b \to \square, L$

- **qa**: 
  - **q1**: $b \to \square, L$

- **qrej**: 
  - **q1**: $\square \to \square, R$
  - **q2**: $\square \to \square, L$

- **qb**: 
  - **q1**: $a \to \square, L$

- **qacc**: 
  - **q1**: $a \to \square, L$
  - **q2**: $b \to \square, L$
```
Turing machine simulation example

Input: aaba

Diagram:

- States: q0, qa, qrej, qb, qacc
- Transitions:
  - a → □, R
  - b → □, R
  - □ → □, R
  - b → □, L
  - □ → □, L
  - □ → □, L
  - □ → □, L
  - □ → □, L
  - □ → □, L
  - □ → □, L
  - □ → □, L
  - □ → □, L
  - a → □, L
  - b → □, L

Initial state: q0
Final states: qa, qb, qacc
Turing machine simulation example

Input: aaba
Turing machine simulation example

Input: aaba
Turing machine simulation example

Input: aaba

Decision: Accept
Turing machine simulation example

Input: baaaaaa
Input: baaaaaa
Turing machine simulation example

Input: baaaaaa
Turing machine simulation example

Input: baaaaaa
Turing machine simulation example

Input: baaaaaa
Turing machine simulation example

Input: baaaaaa
Turing machine simulation example

Input: baaaaa
Input: baaaaa
Turing machine simulation example

Input: baaaaaa
Turing machine simulation example

Input: baaaaaa
Turing machine simulation example

Input: baaaaa

Decision: Reject
```python
def M(input):
    i = 0  # tape head position

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

---

### State Transition Diagram

```
q0 -> qa:
   a <-> □, R
   b <-> □, R

qa -> q6:
   b <-> □, L

qa -> qrej:
   □ <-> □, L

q6 -> qrej:
   □ <-> □, L

qrej -> qrej:
   □ <-> □, L

qrej -> qa:
   a <-> □, L

qa -> acc:
   a <-> □, L

acc -> q6:
   b <-> □, L
```

---

Note: The state diagram shows the transitions between states based on input symbols and tape head movements.
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;

    
    : :
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;

    \[\vdots\]
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;

..
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;
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;
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;

    :
The machine accepts a string $x$ if and only if:

- $|x| = 2$ and $x[0] = x[1]$
- $x$ has at least two a’s or two b’s.
- $x[0] \neq x[1]$
- $|x| > 1$ and $x[0] = x[1]$ None of these.
- $|x| < 2$ or $x[0] = x[1]$ Beats me.
Let $\Sigma = \{a, b\}$.

Draw the state diagram of a TM that accepts a string iff it starts and ends with an $a$. 
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 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);
- $q_{\text{acc}} \in Q$ (which we call the accept state);
- $q_{\text{rej}} \in Q$, $q_{\text{rej}} \neq q_{\text{acc}}$ (which we call the reject state);
A bit more involved to define rigorously.

Not too much though.

See Homework 2.
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 *decider* if it halts on all inputs.

Definition: A language $L$ is called *decidable* (or *computable*) if $L = L(M)$ for some decider TM $M$. 
regular languages $\equiv$ decidable languages
A Turing machine that decides $0^n1^n$.

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

$\Gamma = \{0, 1, \#, \sqcup\}$

(Omitted information defined arbitrarily. Missing transitions go to the reject state.)
Turing machine that decides $0^n 1^n$

Input: 00001011
Turing machine that decides $0^n1^n$
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
Turing machine that decides $0^n 1^n$

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 $0^n1^n$

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

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 $0^n1^n$

**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 $0^n1^n$

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 $0^n1^n$

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

Input: 00001011
Turing machine that decides $0^n 1^n$
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
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 $0^n 1^n$
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 $0^n1^n$

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!
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

\[ x_1 x_2 x_3 \ldots x_n \quad \text{to} \quad \square x_1 \square x_2 \square x_3 \ldots \square x_n \]

- 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.
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
A note

You could describe a TM in 3 ways:

Low level description
   State diagram

Medium level description
   Description of the movement and the behavior of the tape head.

High level description
   Pseudocode or algorithm
Important Question

Is TM the right definition?

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

EvenLength

Factoring

$0^n \mid n$

isPrime

Solvable with any computing device?
The intuitive notion of “computable” is captured by functions computable by a Turing Machine.

This is **not** a theorem!

Is it …

* an observation?
* a definition?
* a hypothesis?
* a law of nature/physics?
* a philosophical statement?
How did Turing think about all this?

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

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

Turing justified TMs by arguing that it can do anything a human could.
What else did Turing do in his paper?

Universal Machine
(one machine to rule them all)

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)

We could use:

```
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;
```
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.

![Diagram](image.png)
What else did Turing do in his paper?

There are languages that cannot be computed!
Solvable with any computing device

= TM-decidable

- Regular languages
- EvenLength
- Factoring
- \(0^n | n\)
- Primality

\(?\)
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!
How do you show a problem is **undecidable**?

Well, of course, you assume it is decidable, and reach a contradiction.

Next week’s topic!