15-251
Great Theoretical Ideas in Computer Science

Lecture 4:
Turing’s Legacy

September 10th, 2015
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.

Identify these with the corresponding algorithms.

We only have machines to solve decision problems.
Definitions from last time

\[ \Sigma^* = \text{the set of all finite length strings over } \Sigma \]

A subset \( L \subseteq \Sigma^* \) is called a language.

A computational problem is a function \( f : \Sigma^* \rightarrow \Sigma^* \).

A decision problem is a function \( f : \Sigma^* \rightarrow \{0, 1\} \).

There is a one-to-one correspondence between decision problems and languages.
Input: 01111
DFA: state diagram + input tape

tape

| 1 | 0 | 1 | 1 | 1 | 0 | 1 | □ | □ | □ | □ | □ |

tape head

“blank” symbol
DFA: state diagram + input tape

1 0 1 1 1 0 1 ...
DFA: state diagram + input tape

\[\begin{array}{ccccccccc}
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & \ldots
\end{array}\]
DFA: state diagram + input tape

```
1 0 1 1 1 0 1 □ □ □ □ □ □ ...
```

![DFA Diagram](image)
DFA: state diagram + input tape

```
1 0 1 1 1 0 1
...```

![DFA Diagram](image)
DFA: state diagram + input tape
DFA: state diagram + input tape
DFA: state diagram + input tape
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;
    ...
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 set (which we call the set of states);
- $\Sigma$ is a finite 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 \( w = w_1 w_2 \cdots w_n \) be a string over an alphabet \( \Sigma \).

Let \( M = (Q, \Sigma, \delta, q_0, F) \) be a DFA.

We say that \( M \) accepts the string \( w \) if there exists a sequence of states \( r_0, r_1, \ldots, r_n \in Q \) such that

- \( r_0 = q_0 \);
- \( \delta(r_{i-1}, w_i) = r_i \) for each \( i \in \{1, 2, \ldots, n\} \);
- \( r_n \in F \).

Otherwise we say \( M \) rejects the string \( w \).
Definition: A language $L$ is called regular if $L = L(M)$ for some DFA $M$. 
Theorem:
The language \( L = \{0^n1^n : n \in \mathbb{N}\} \) is not regular.

Theorem:
Let \( \Sigma \) be some finite alphabet.
If \( L_1 \subseteq \Sigma^* \) and \( L_2 \subseteq \Sigma^* \) are regular, then so is \( L_1 \cup L_2 \).
What is **computation**?

What is an **algorithm**?  
A specification that describes how information is transformed.

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.

Stephen Kleene
2 important observations:

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.
Solvable with any computing device

- Regular languages
  - EvenLength
  - ...

- Factoring
  - $0^n \mid n$
  - Primality
  - ...


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
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++;   /* Reject */
        j—;
    }
    return 1; /* Accept */
}
Solvable with Python?

- Regular languages
  - EvenLength

- Factoring
  - $0^n1^n$

- Primality
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.
### Turing Machine Description

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a</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>

- **TM** $\approx$ **DFA** $+$ 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).
  - You can read & write to the tape cell pointed to.
Turing machine description

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

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

- Move the head left
- Move the head right
- Write a symbol a (from the alphabet)
- If head is reading symbol a, GOTO step j
- Halt and accept
- Halt and reject

But, we want to keep the def’n as simple as possible.
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 simulation example

Input: aaba
Input: aaba
Turing machine simulation example

Input: aaba
Turing machine simulation example

Input: aaba
Turing machine simulation example

Input: aaba
Turing machine simulation example

Input: aaba

Diagram with states and transitions:
- States: $q_0$, $q_a$, $q_{rej}$, $q_{acc}$, $q_b$
- Transitions:
  - $a \leftrightarrow \square, R$
  - $b \leftrightarrow \square, R$
  - $\square \leftrightarrow \square, R$
  - $b \leftrightarrow \square, L$
  - $a \leftrightarrow \square, L$
  - $\square \leftrightarrow \square, L$
  - $\square \leftrightarrow \square, L$
  - $a \leftrightarrow \square, L$
  - $b \leftrightarrow \square, L$

Input tape: aaba
Turing machine simulation example

Input: aaba

Diagram:

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

Input tape:

```
3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13
□ □ □ □ □ □ b a □ □ □ □ □ □ □ □ □ □ □
```
Turing machine simulation example

Input: aaba

\[ q_0 \]
- \( a \rightarrow \square, R \)
- \( b \rightarrow \square, R \)
- \( \square \rightarrow \square, R \)

\[ qa \]
- \( b \rightarrow \square, L \)
- \( \square \rightarrow \square, L \)

\[ q_{rej} \]
- \( a \rightarrow \square, L \)
- \( \square \rightarrow \square, L \)

\[ qb \]
- \( \square \rightarrow \square, L \)

\[ q_{acc} \]
- \( a \rightarrow \square, L \)
- \( b \rightarrow \square, L \)
Turing machine simulation example

Input: aaba

<table>
<thead>
<tr>
<th>3</th>
<th>-2</th>
<th>-1</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></td>
<td></td>
<td></td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- **q0**: Start state.
- **qa**: Transition on 'a' to 'q0', move right.
- **qb**: Transition on 'b' to 'q0', move right.
- **qrej**: Transition on 'a' or 'b' to 'qrej', move left or right.
- **qacc**: Accept state.

Transition rules:

- 'a' moves right: \( a \rightarrow \sqcup, R \)
- 'b' moves right: \( b \rightarrow \sqcup, R \)
- Blank moves right: \( \sqcup \rightarrow \sqcup, R \)
- 'a' moves left: \( a \rightarrow \sqcup, L \)
- 'b' moves left: \( b \rightarrow \sqcup, L \)
- Blank moves left: \( \sqcup \rightarrow \sqcup, L \)
- 'a' moves left: \( a \rightarrow \sqcup, L \)
- 'b' moves left: \( b \rightarrow \sqcup, L \)
Turing machine simulation example

Input: aaba

Decision: Accept
Input: baaaaaa
Turing machine simulation example

Input: baaaaaa

Diagram:

States: q0, qa, qrej, qacc, qb

Transitions:
- q0 to qa: a → □, R
- qa to q0: b → □, L
- q0 to qrej: □ → □, R
- qrej to q0: □ → □, L
- q0 to qacc: a → □, L
- q0 to qb: b → □, L
Turing machine simulation example

Input: baaaaaa
Input: baaaaaa
Turing machine simulation example

Input: baaaaaa
Input: baaaaaa

Turing machine simulation example
Turing machine simulation example

Input: baaaaaa
Turing machine simulation example

Input: baaaaa

Diagram:

- Initial state: $q_0$
- Transition rules:
  - $a \mapsto \square, R$
  - $b \mapsto \square, R$
  - $\square \mapsto \square, R$
  - $b \mapsto \square, L$
  - $a \mapsto \square, L$
  - $\square \mapsto \square, L$
  - $\square \mapsto \square, L$

States:
- $q_a$
- $q_{rej}$
- $q_{acc}$
- $q_b$
Turing machine simulation example

Input: baaaaaa
Turing machine simulation example

Input: baaaaaa

```
3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13
|
| a a a a a
|
|
```

**States:**
- $q_0$
- $q_a$
- $q_{rej}$
- $q_{acc}$
- $q_b$

**Transitions:**
- $a \leftrightarrow \square, R$
- $b \leftrightarrow \square, R$
- $\square \leftrightarrow \square, R$
- $b \leftrightarrow \square, L$
- $a \leftrightarrow \square, L$
- $\square \leftrightarrow \square, L$
- $\square \leftrightarrow \square, L$
- $a \leftrightarrow \square, L$
- $b \leftrightarrow \square, L$
Turing machine simulation example

Input: baaaaa

Decision: Reject
def foo(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;

...
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;
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;
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;
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;
    ...
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;

: 
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;

  

TM as a programming language
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;
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;

::
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;

    ...
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;


TM as a programming language
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;

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

- $x[0] = x[1]$ and $|x| = 2$
- $x$ 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.
- $x[0] = x[1]$  
  Beats me.
Let $\Sigma = \{a, b\}$.

Draw the state diagram of a TM that accepts a string iff it starts and ends with an $a$. 
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 $\sqcup \notin \Sigma$ (which we call the input alphabet);
- $\Gamma$ is a finite set with $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$ (which we call the tape alphabet);
- $\delta$ is a function of the form $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ (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 complicated 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
Turing machine that decides $0^n1^n$

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

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

(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 $0^n1^n$

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 $0^n1^n$

**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 $0^n1^n$

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 $0^n1^n$

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

Input: 00001011
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 $0^n1^n$

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 $0^n1^n$

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

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

Input: 00001011
Input: 00001011

Turing machine that decides $0^n1^n$
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 $0^n 1^n$
Turing machine that decides $0^n1^n$

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

Input: 00001011
Turing machine that decides $0^n 1^n$
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 □ 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 \( \Gamma = \{0, 1, 0^\ast, 1^\ast, \square\} \)

- 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 a dictionary data structure
- Simulate “random access memory”
  :
- Simulate assembly language
  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?
Solvable with any computing device

- TM-decidable
  - Regular languages
    - EvenLength
    - \( 0^n \mid n \)
  - Factoring
  - Primality

\textit{Solvable with any computing device}
Church-Turing Thesis

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.
Computers in the age of Turing
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)

+ Primality Sorting DFA $|x|$ even

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

$\langle \text{A TM} \rangle$ this is a TM

$\langle x \rangle$ output of the TM on input $x$
What else did Turing do in his paper?

Universal Machine
(one machine to rule them all)

This is exactly what an interpreter does.
What else did Turing do in his paper?

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

= 

TM-decidable

? 

Regular languages

Factoring

EvenLength

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.

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

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!