Goal of next lecture:
Explore physical, philosophical, historical questions surrounding Turing machines.
Let's assume two things about our world

1. No “universal” machines exist.

2. We only have machines to solve decision problems.

**DFA: state diagram + input tape**

Decision: Accept
DFA as a programming language

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;
    ...

machine ≈ algorithm describing it

input data → a DFA → output data

algorithm

What is computation?
What is an algorithm?
How can we mathematically define them?

The properties we want from the definition:
2 important observations:

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

Solvable with any computing device
Solving $0^n1^n$ 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 $0^n1^n$ in C

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

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

Downsides as a formal definition?

**So what we want is:**

A totally minimal (TM) programming language such that:

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**

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 definition 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 'a'; 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:
Want to stay put:
Don’t want to change state:

**Turing machine official picture**

```
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   | a | a | b | a |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+
Input: aaba
```

**TM as a programming language**

```python
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;
```

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

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

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$$

where

- $Q$ is a finite set (which we call the set of states);
- $\Sigma$ is a finite set with $\cup \notin \Sigma$ (which we call the input alphabet);
- $\Gamma$ is a finite set with $\cup \in \Gamma$ and $\Sigma \subseteq \Gamma$ (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);
Formal definition: TM accepting a string

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

DFAs vs TMs

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?
regular languages = decidable languages

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

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

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

Input: 00001011
Turing machine that decides $0^n \mid n$

Input: 00001011          Decision: reject

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 \( \text{\text{\textbackslash I}} x_1 \text{\text{\textbackslash I}} x_2 \text{\text{\textbackslash I}} x_3 \ldots \text{\text{\textbackslash I}} x_n \)
- 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”

- 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**
- **Medium level description**
- **High level description**

---

**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

Factoring

$0^n \mid n$

isPrime

Regular languages

EvenLength

: 

: 

: 

: 
