

Turing's Legacy



What is **computation**?
What is an **algorithm**?

How can we mathematically define them?

How does the definition affect which computational problems are solvable?

Problem: A mapping of *instances* to *solutions*.

$$f : \Sigma^* \rightarrow \Sigma^*$$

E.g.: **MULTIPLICATION**, **SORTING**

Decision Problem: The solution is Yes/No.

E.g.: **PRIMALITY**

PALINDROME

$$L = \{0^n 1^n : n \in \mathbb{N}\}$$

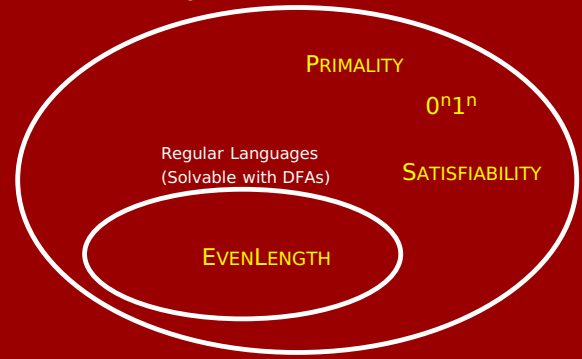
(recall that Languages \equiv Decision Problems)

$$\text{EVENLENGTH} = \{x \in \Sigma^* : \text{length}(x) \text{ is even}\}$$

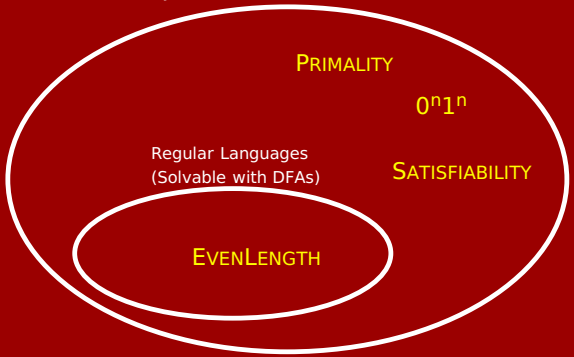
SATISFIABILITY

(*instance*: propositional formula S ;
solution: is S satisfiable?)

Solvable with "algorithms" (?)



Solvable with Python



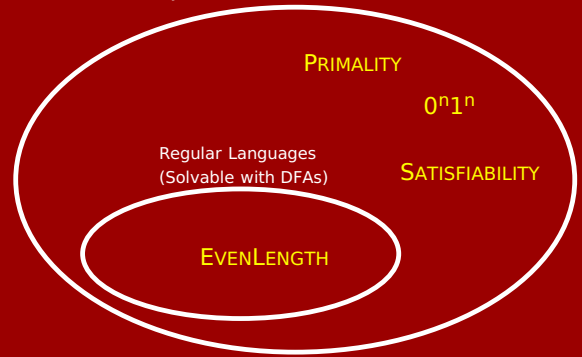
Solving $0^n 1^n$ with Python

```
# Determines if string S is of form 0^n 1^n
def Solution( S ):
    i = 0
    j = len(S)-1
    while j >= i:
        if S[i] != '0' or S[j] != '1':
            return False
        i = i + 1
        j = j - 1
    return True
```

Solving 0^n1^n with C

```
/* Determines if string S is of form 0^n 1^n */
int Solution(char S[])
{
    int i = 0, j;
    while (S[j] != NULL) /* NULL is end-of-string char */
        j++;
    j--;
    while (j >= i)
    {
        if (S[i] != '0' || S[j] != '1')
            return 0; /* Reject */
        i++;
        j--;
    }
    return 1; /* Accept */
}
```

Solvable with Python



Question:

Should we just define “algorithm” to mean a function written in Python?

Before we answer this, a **caveat**:

- Problem instances are strings of **any** length.
- These instances are stored in ‘memory’.
- So we have to **assume unlimited memory**.
- Wouldn't be mathematically natural otherwise!

“640k ought to be enough for anybody.”

APOCRYPHAL



Bill Gates, 1981

Question:

Should we just define “algorithm” to mean a function written in Python?

(allowed access to unlimited memory)

Answer:

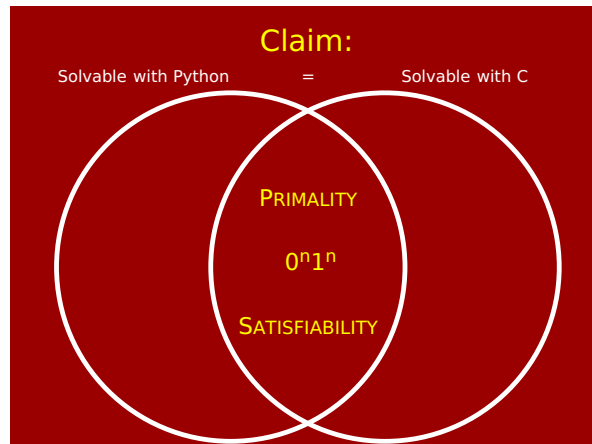
Actually, we'll eventually argue that this would be OK!

Downsides as a formal definition:

- Why choose Python?
Why not C, or Java, or SML, or...?
- Extremely complicated to rigorously define. E.g., official 2011 ISO definition of C requires a 701-page PDF file!

Downsides as a formal definition:

- Why choose Python?
Why not C, or Java, or SML, or...?
- Extremely complicated to rigorously define.
E.g., official 2011 ISO definition of C requires a 701-page PDF file!



Claim:

Solvable with Python = Solvable with C

Proof intuition:

Our shared experience with programming.

“Proof:”

1. Solvable with Python \subseteq Solvable with C.
The standard Python interpreter is written in C.
2. Solvable with C \subseteq Solvable with Python.
Well, it's pretty clear you can write a C interpreter in Python.

Interpreters

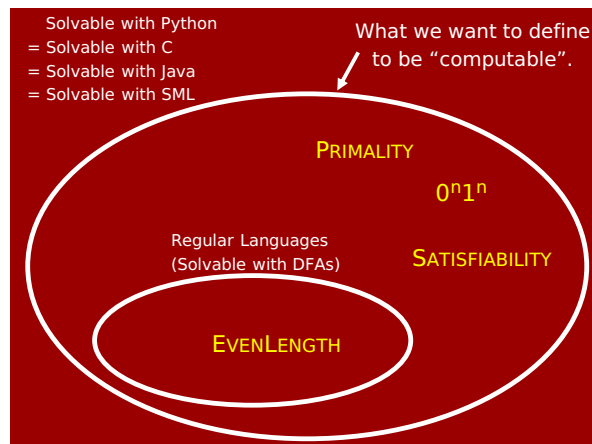
A Python function is (representable by) a string.

A Python interpreter is an algorithm (written in some programming language) that takes two inputs: P , a Python function; x , a string; and step-by-step simulates $P(x)$.

Interpreters

You can write a Python interpreter in C.
You can write a C interpreter in Python.
You can write a Python interpreter in Java.
You can write a Java interpreter in Python.
You can write a Python interpreter in SML.
You can write an SML interpreter in Python.
You can write a Python interpreter in Python!!

The last one is called a “Universal Python Program”



Downsides as a formal definition:

- Why choose Python?
Why not C, or Java, or SML, or...?
- Extremely complicated to rigorously define.
E.g., official 2011 ISO definition of C requires a 701-page PDF file!

Downsides as a formal definition:

- Why choose Python?
Why not C, or Java, or SML, or...?
- **Extremely complicated to rigorously define.**
E.g., official 2011 ISO definition of C requires a 701-page PDF file!
- We could go down to bytecode.
Though still takes 14 pages to define `C0VM`.

Would be nice to have a **totally minimal** ("TM") programming language such that:

- a) it can simulate simple bytecode like `C0VM`, and hence also C, Python, Java, SML, etc.;
- b) it's simple enough to reason about completely mathematically rigorously.

Let's try to invent one.

Inventing TM programming language

TM will be a ridiculously simple programming language for manipulating strings.

Through examples, we'll show TM is **more powerful** than you might first expect.

Eventually, we'll claim you can **simulate simple bytecode** (e.g. `C0VM`) in TM.

We don't have the time to prove this rigorously. We'll just appeal to your **programming intuition**.

By the way:

TM actually stands for **Turing Machines**



Alan Turing invented them in a 1936 paper he wrote while a PhD student.

Defining Turing Machines

As with DFAs, a Turing Machine **M** is like a piece of code. Unlike DFAs, Turing Machines have access to **memory**: abstracted as a two-way infinite "**tape**" of cells.

-2 -1 0 1 2 3 ← Tape cell position numbers



The input from Σ^* is written starting at cell 0.

Defining Turing Machines

As with DFAs, a Turing Machine M is like a piece of code.
 Unlike DFAs, Turing Machines have access to **memory**:
 abstracted as a two-way infinite “**tape**” of cells.

-2 -1 0 1 2 3 ← Tape cell position numbers



The input from Σ^* is written starting at cell 0.
 All other cells contain \sqcup (blank).

Defining Turing Machines

As with DFAs, a Turing Machine M is like a piece of code.
 Unlike DFAs, Turing Machines have access to **memory**:
 abstracted as a two-way infinite “**tape**” of cells.

-2 -1 0 1 2 3 ← Tape cell position numbers



The input from Σ^* is written starting at cell 0.
 All other cells contain \sqcup (blank).

Defining Turing Machines

As with DFAs, a Turing Machine M is like a piece of code.
 Unlike DFAs, Turing Machines have access to **memory**:
 abstracted as a two-way infinite “**tape**” of cells.

-2 -1 0 1 2 3 ← Tape cell position numbers



The input from Σ^* is written starting at cell 0.
 All other cells contain \sqcup (blank).

In general, cells contain symbols from a **tape alphabet** Γ ,
 which must contain Σ , \sqcup , and may have other symbols.
 There's a tape pointer (“**head**”), initially at position 0.

Defining Turing Machines



There's a tape pointer (“**head**”), initially at position 0.

TM could've been defined as sequence of n instructions
 where the allowed instructions are...

- Move the head left
- Move the head right
- Write symbol a (for any $a \in \Gamma$)
- If head is reading symbol a , GOTO step j (for any $a \in \Gamma$, $1 \leq j \leq n$)
- Halt & accept
- Halt & reject

Defining Turing Machines



There's a tape pointer (“**head**”), initially at position 0.

TM could've been defined as sequence of n instructions

But instead they're traditionally defined a little differently.

Defining Turing Machines

- May as well move the head at each step.
 (If you want to stay put, just do a Left then a Right.)
- May as well write a symbol at each step.
 (Can just rewrite the symbol the head is reading.)
- May as well do a GOTO at each step.
 (Can always just GOTO the next step if you want.)
- Instead of IF checking for a particular symbol,
 may as well do a CASE statement on all symbols.

Defining Turing Machines

Suppose, e.g., tape alphabet is $\Gamma = \{a, b, c, \sqcup\}$.

In TM definition, a **machine** (piece of code) **could** be thought of as a sequence of steps S_1, S_2, \dots, S_k , where each looks something like this:

```

S13: Switch(symbol under the head)
  case a: write c, move Left, goto S10
  case b: write b, move Right, goto S2
  case c: write  $\sqcup$ , move Left, goto S13
  case  $\sqcup$ : write  $\sqcup$ , move Left, goto S14
    
```

Also, one S_j is "Halt & accept", another is "Halt & reject".

Defining Turing Machines

Since there's a GOTO in each step, they're not really "ordered" like "steps".

May as well call them "states".

In each state, TM takes a different action depending on symbol the tape head is reading.

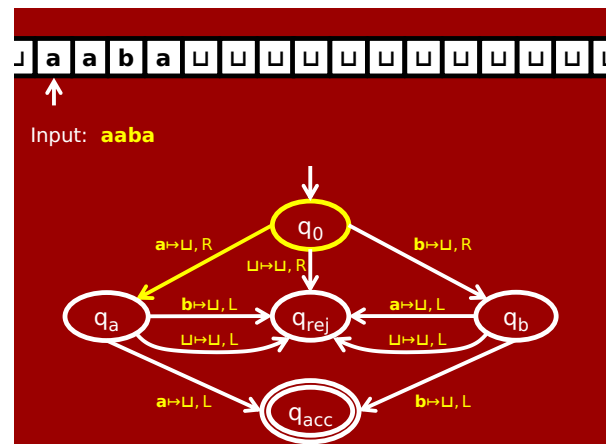
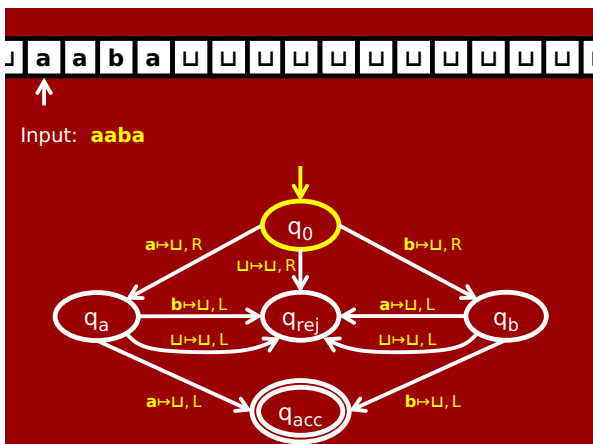
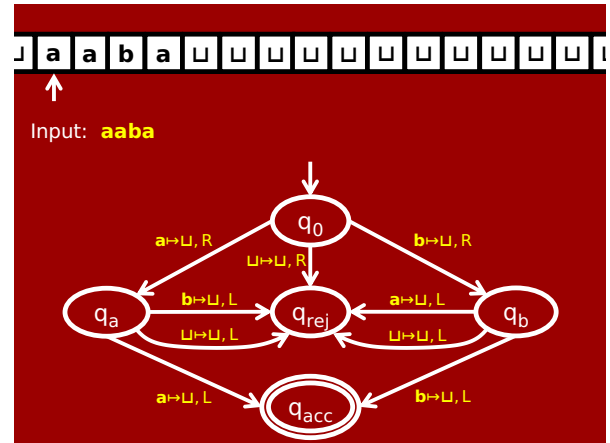
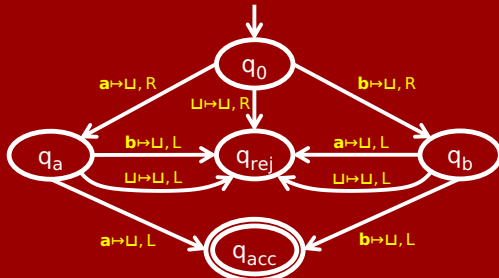
Quite similar to DFAs...

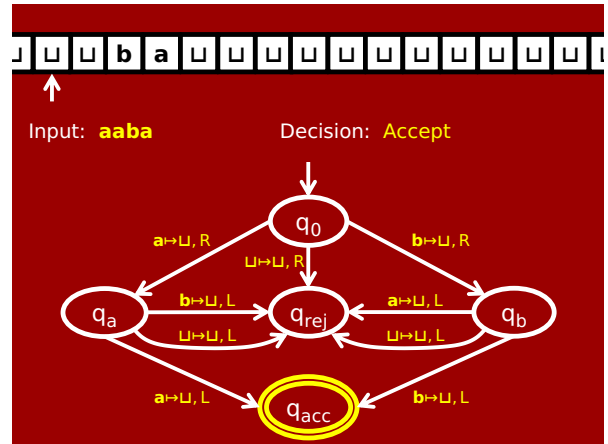
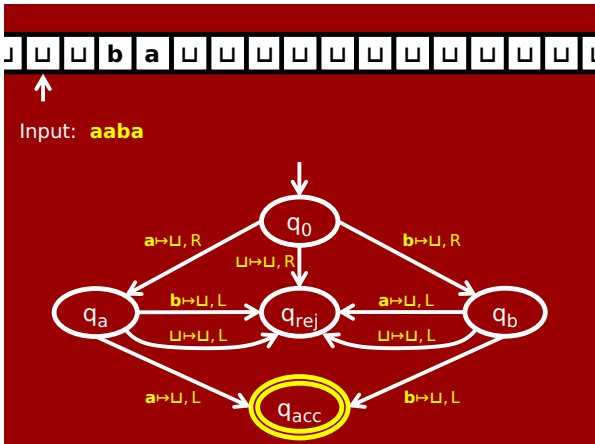
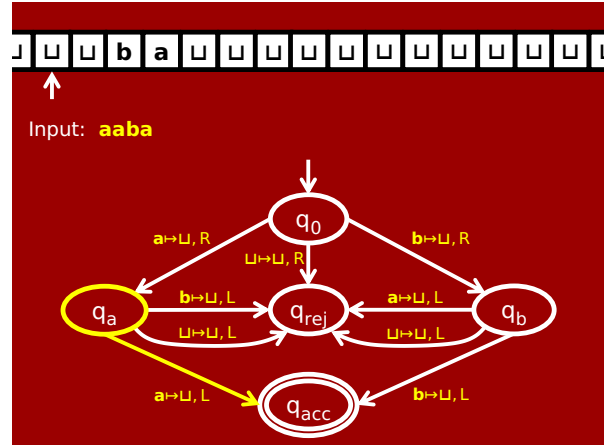
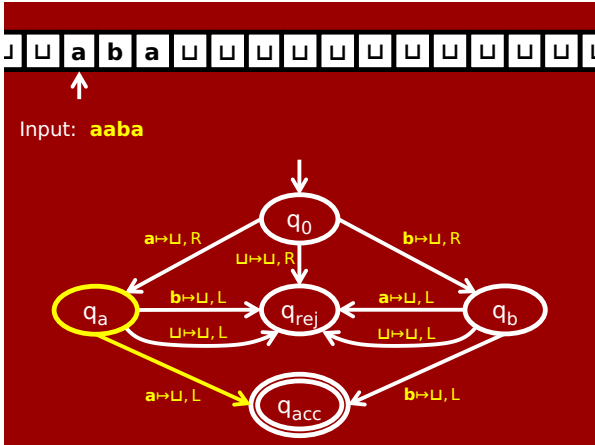
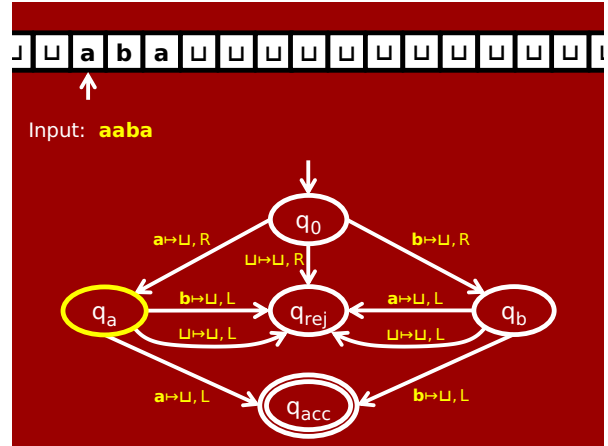
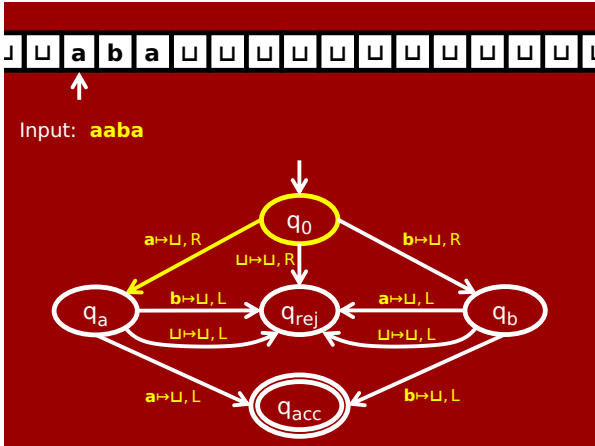
Indeed, we usually draw them...

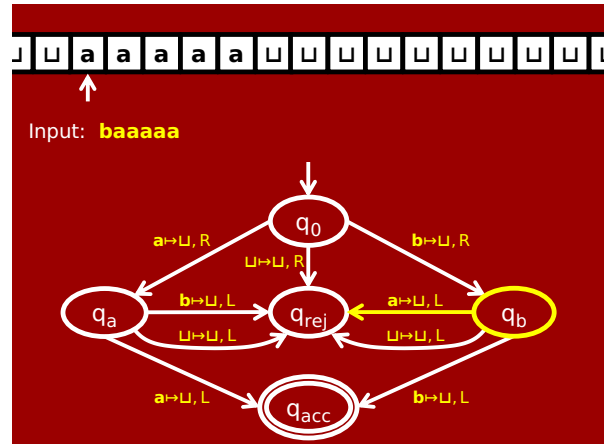
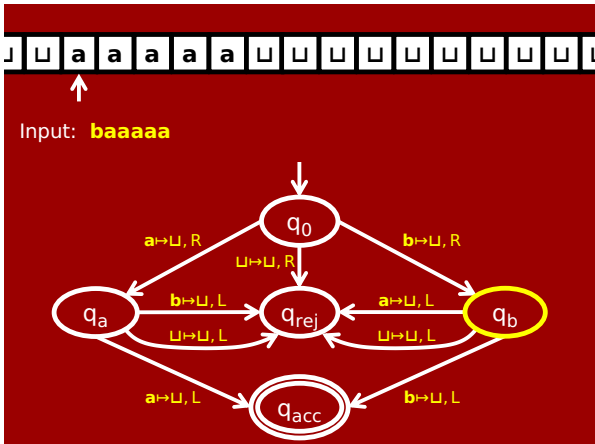
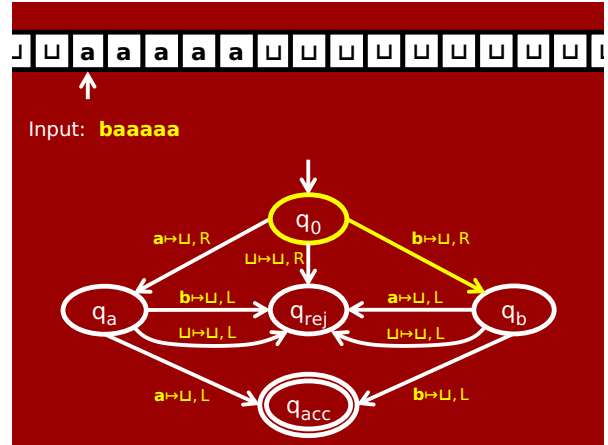
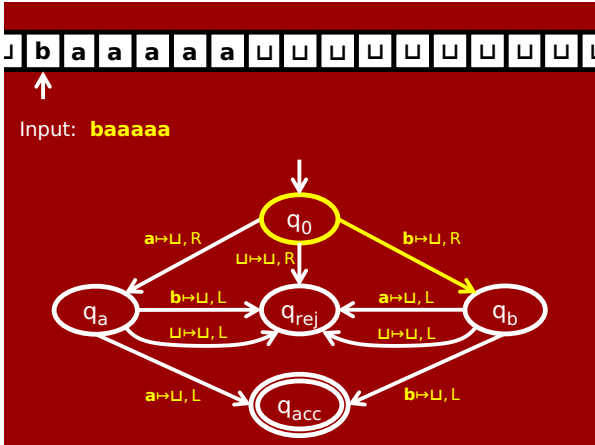
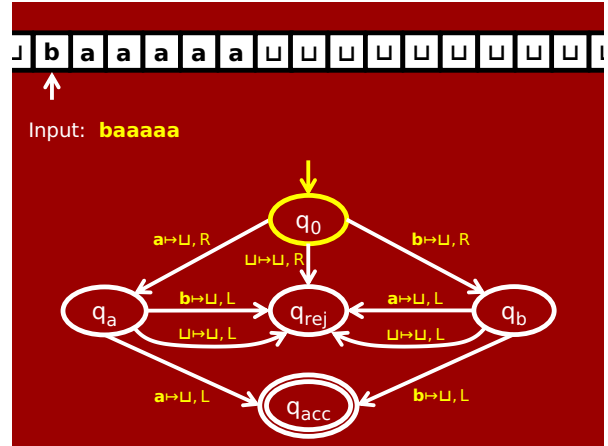
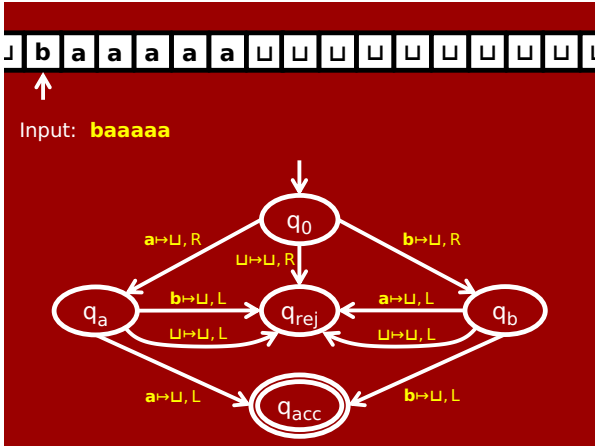
OFFICIAL PICTURE of a Turing Machine

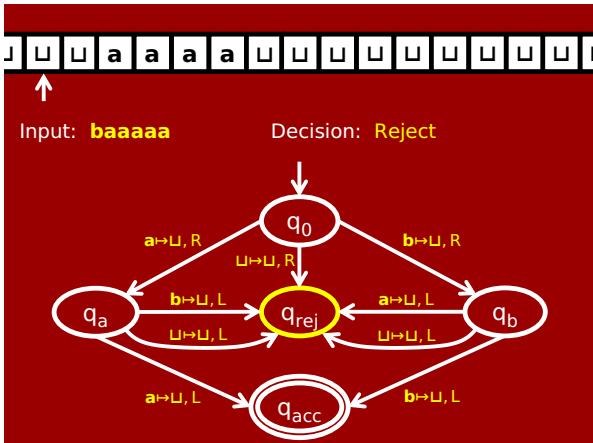
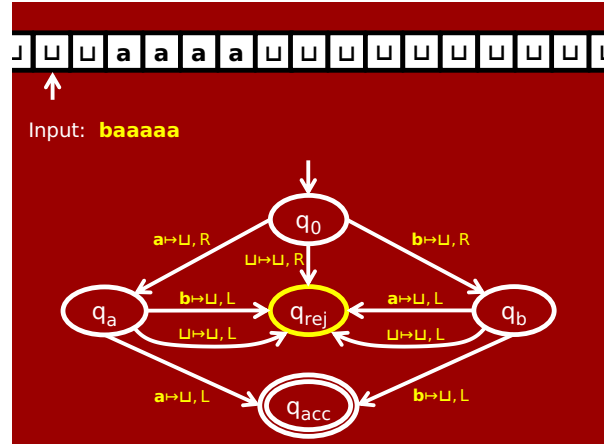
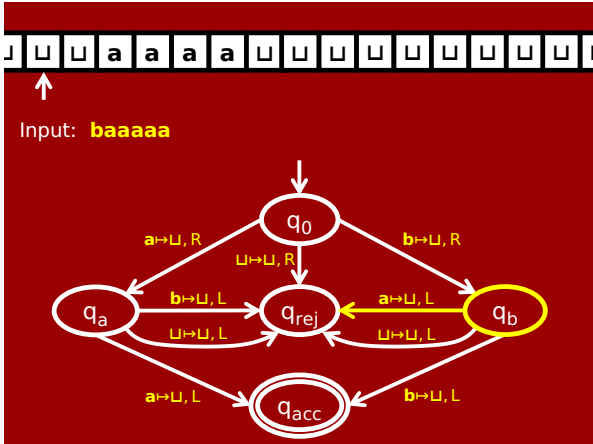
Input alphabet: $\Sigma = \{a, b\}$ Tape alphabet: $\Gamma = \{a, b, \sqcup\}$

$a \rightarrow \sqcup, R$ means "if reading **a**, write **\sqcup** & move head Right"









Let's call that Turing Machine M .
 What does M do on input $x \in \{a,b\}^*$?

If x has length at least 2,
 and first two symbols same, $M(x)$ accepts.

If x has length less than 2,
 or first two symbols different, $M(x)$ rejects.

(Side effects: Tape cells 0, 1 always end up \sqcup .
 Head ends up at position 0,
 unless $x = \epsilon$, in which case position 1.)

OFFICIAL DEFINITION of Turing Machines

A Turing Machine is a 7-tuple
 $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$:

Q is a finite set of **states**,
 Σ is a finite **input alphabet** (with $\sqcup \notin \Sigma$),
 Γ is a finite **tape alphabet** (with $\sqcup \in \Gamma, \Sigma \subseteq \Gamma$)
 $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$ is **transition function**,
 $q_0 \in Q$ is the **start state**,
 $q_{\text{accept}} \in Q$ is the **accept state**,
 $q_{\text{reject}} \in Q$ is the **reject state**, $q_{\text{reject}} \neq q_{\text{accept}}$.

We won't write a formal definition of
 how computation proceeds,
 but it's just what you expect.

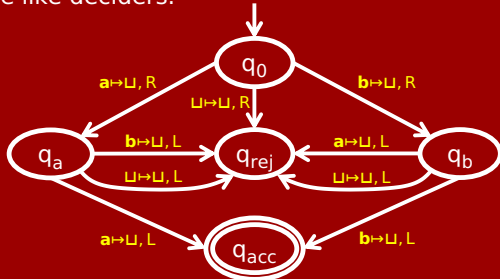
One **CRUCIAL** difference compared to DFAs:
 A Turing Machine might **never halt**.
 Just the same as with "usual" algorithms!

Python example:

```
x=1
while x != 0:
    x=x+1
```

M , our example Turing Machine, **halts on all inputs**.
 A Turing Machine with this property is called a **decider**.

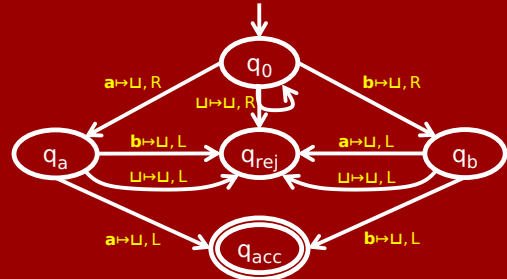
We like deciders.



This slight variant **does not halt** on input ϵ .

If $M(x)$ never halts, we say $M(x)$ "loops".

Here, $M(\epsilon)$ loops. So M is **not a decider**.



Decidable languages

Definition:

A language $R \subseteq \Sigma^*$ is **decidable** (or **computable**) if there is a **decider** Turing Machine M such that:

1. For all $x \in R$, $M(x)$ accepts.
2. For all $x \notin R$, $M(x)$ rejects.

In this case we write $R = L(M)$.

NOTE: If M is not a **decider**, then let's say that M **does not** compute/decide anything.

The language $\{0^n 1^n : n \in \mathbb{N}\}$ is not regular:
 no DFA decides it.

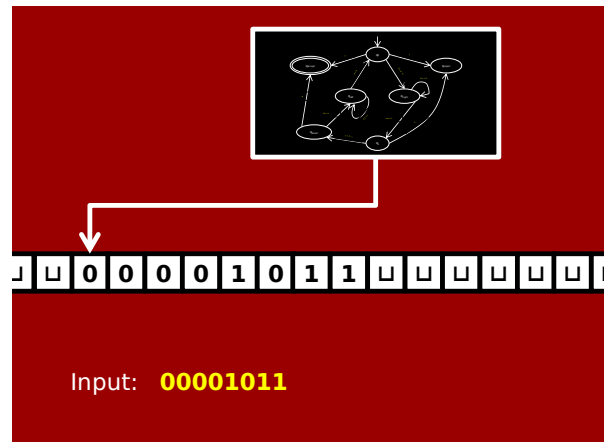
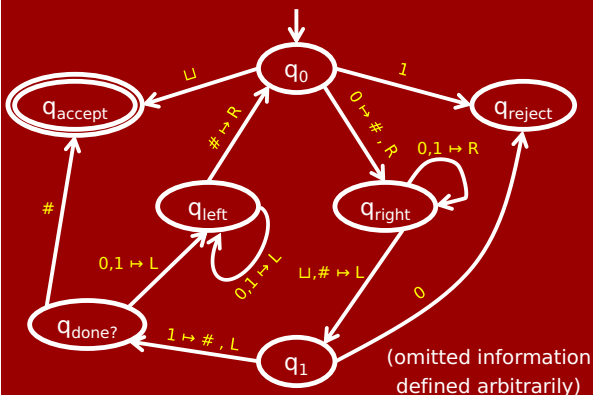
Is it **decidable**?

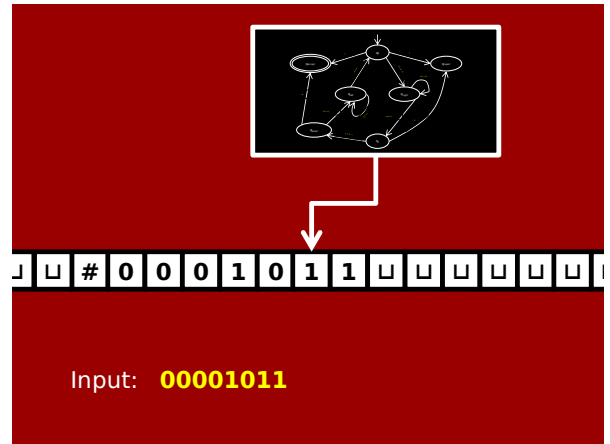
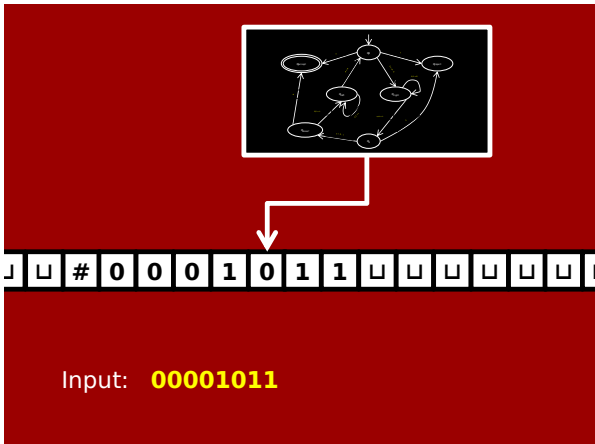
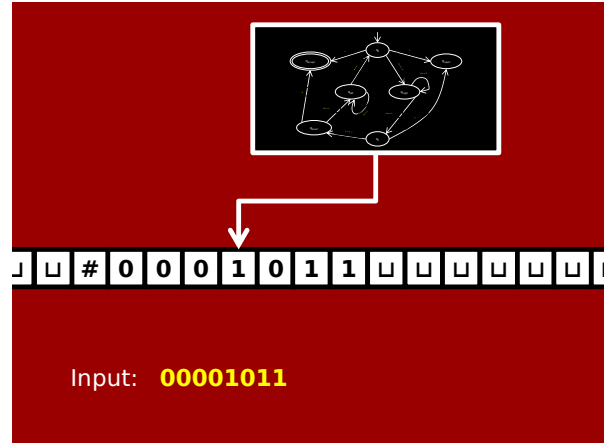
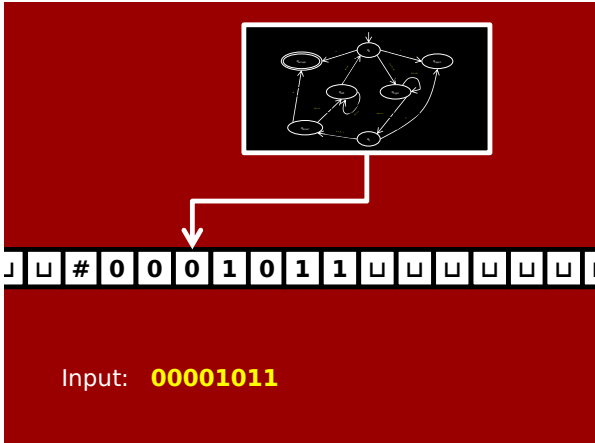
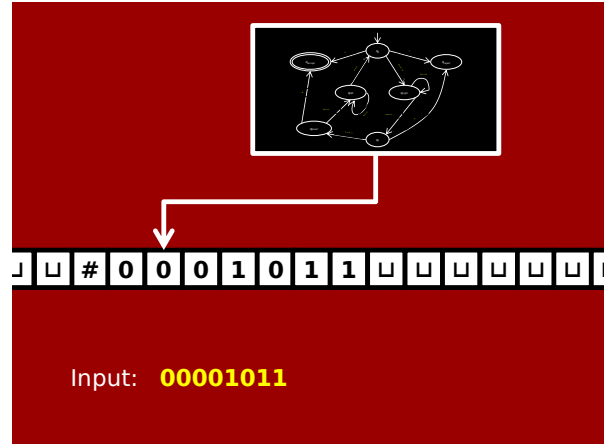
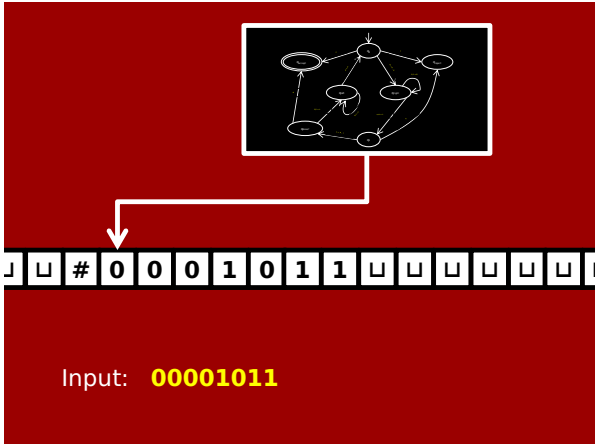
I.e., is there a TM deciding it?

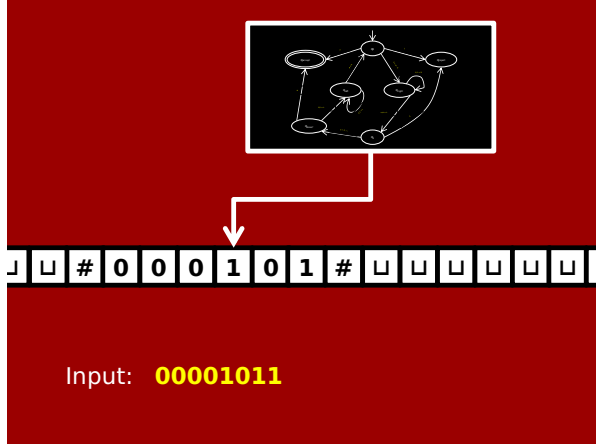
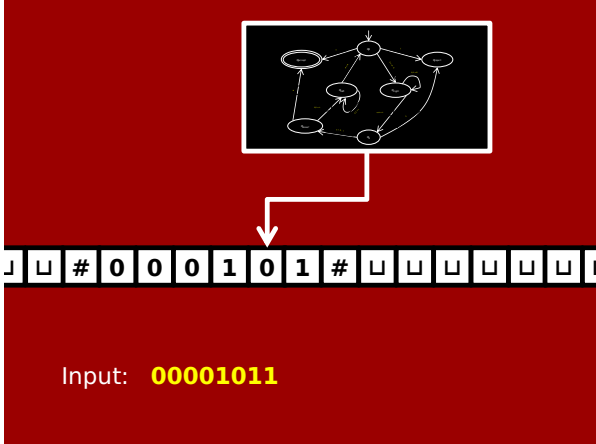
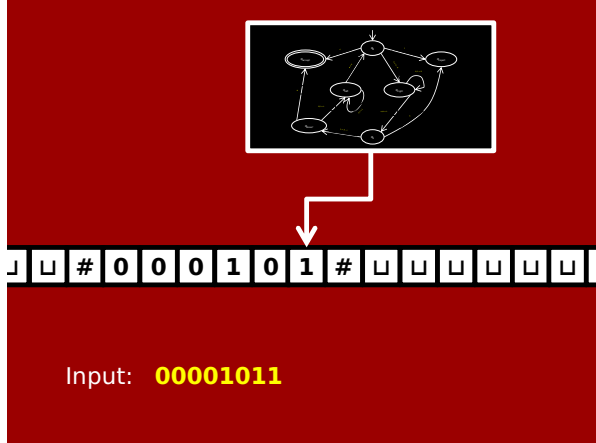
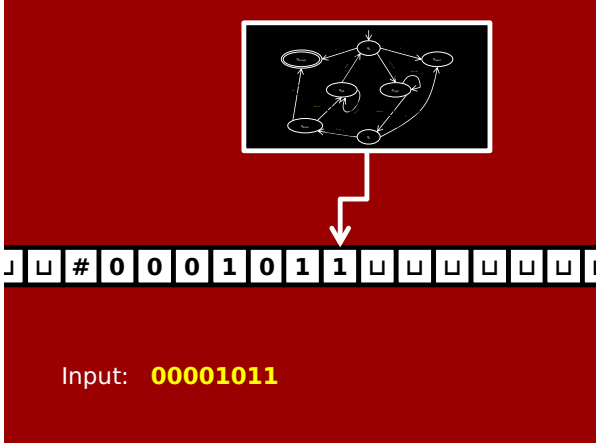
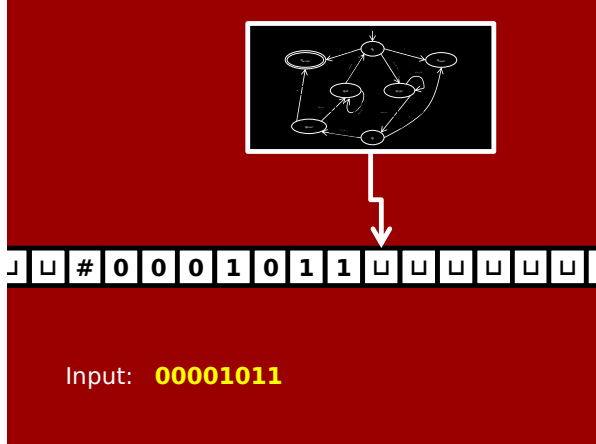
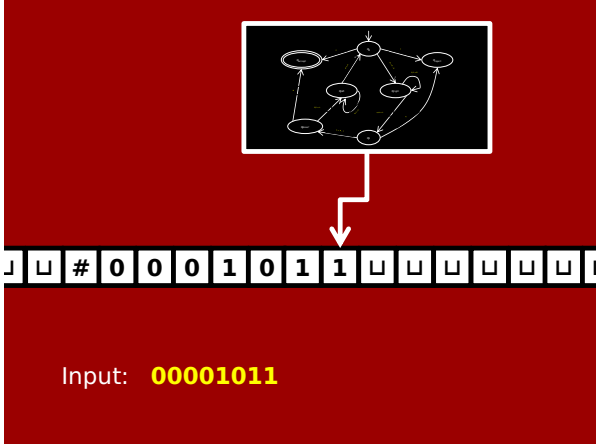
Yes! Let's see a decider TM M such that

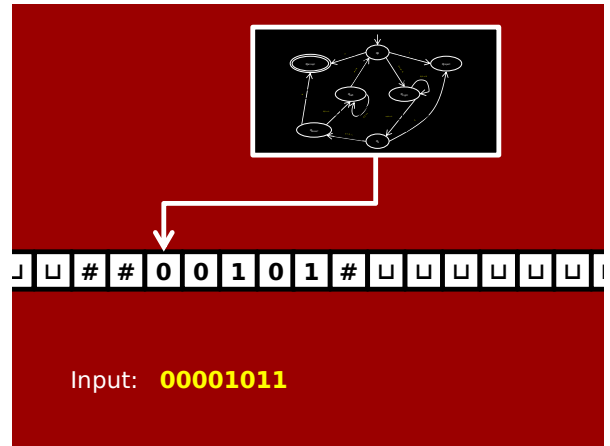
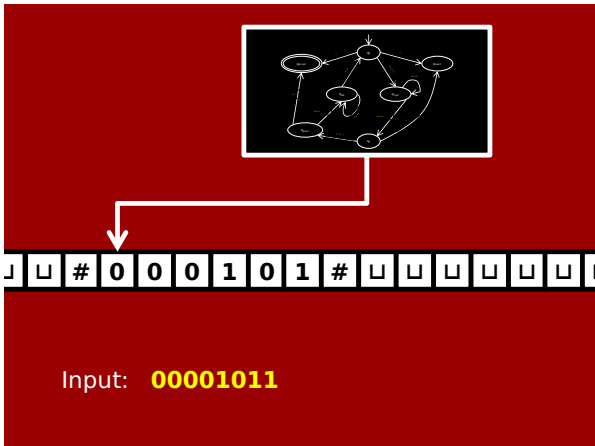
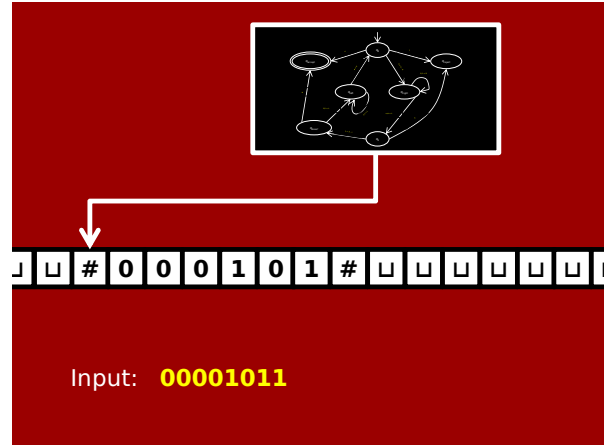
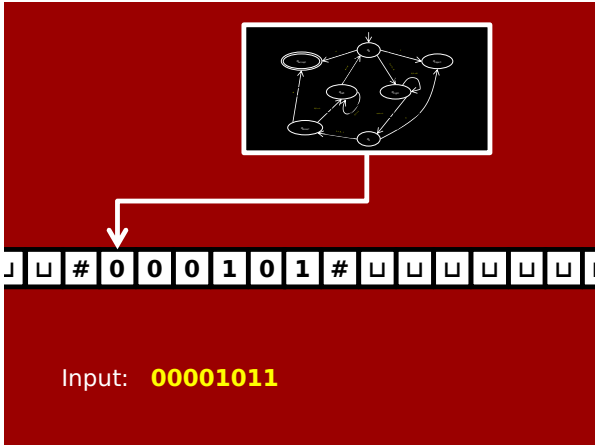
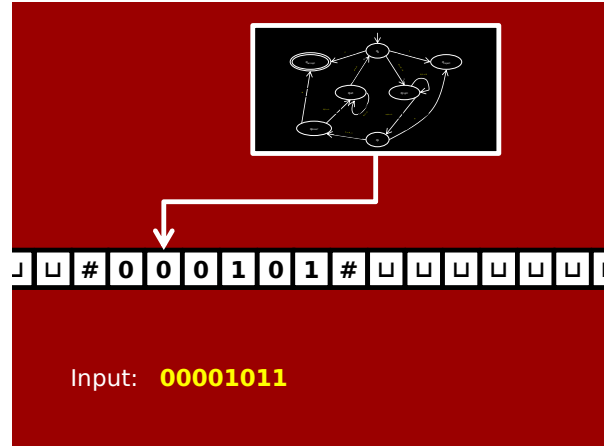
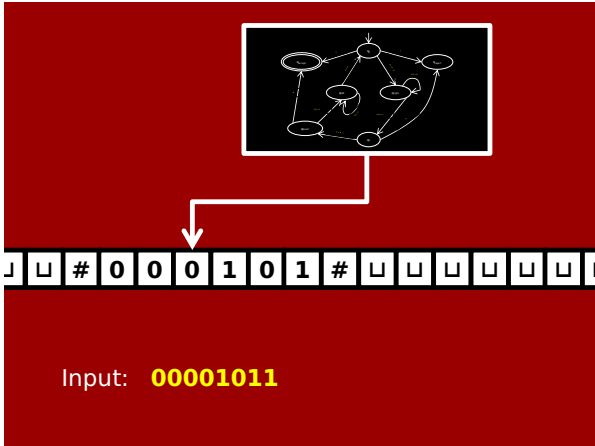
$$L(M) = \{0^n 1^n : n \in \mathbb{N}\}$$

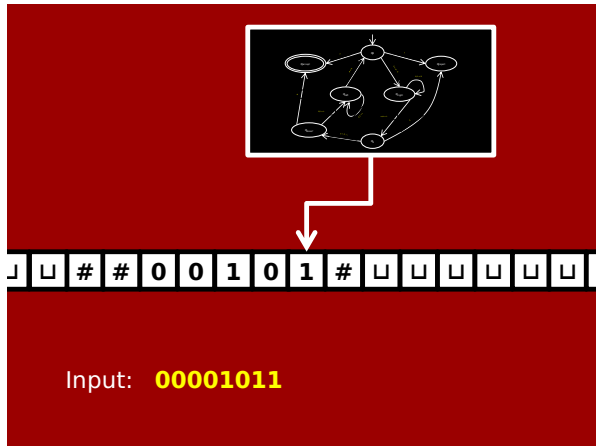
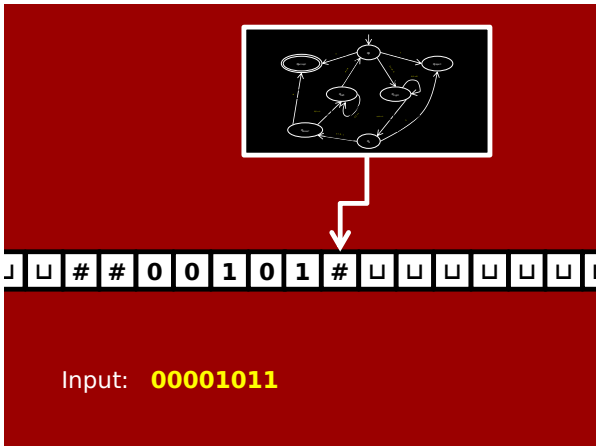
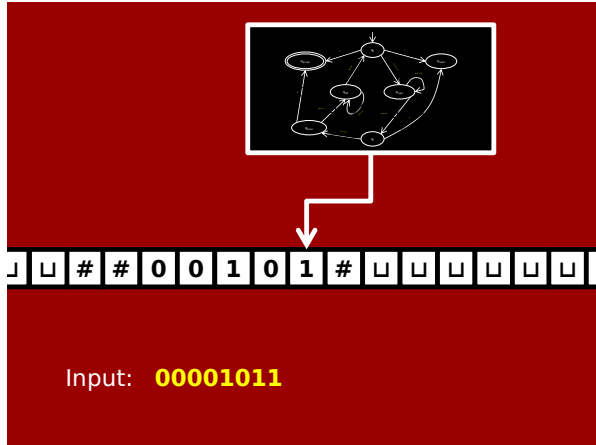
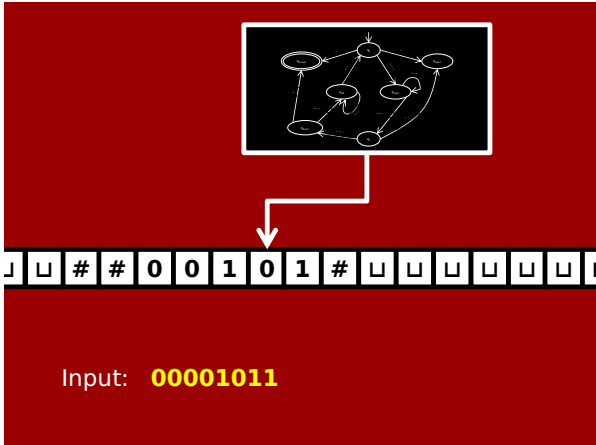
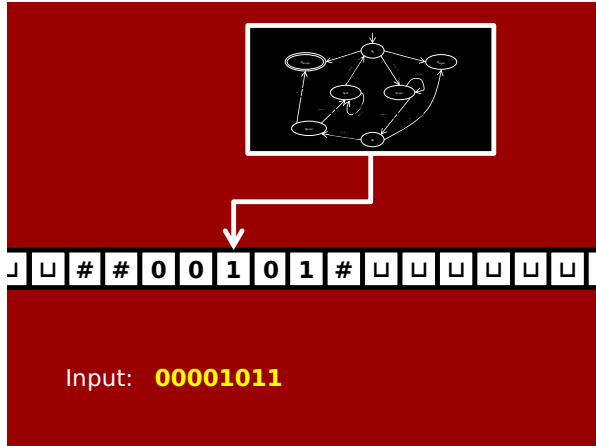
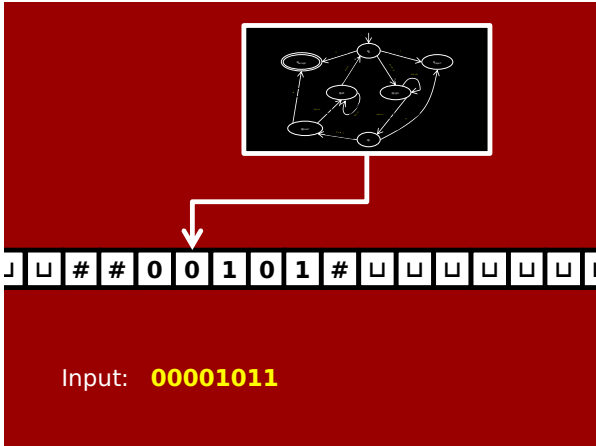
Input alphabet: $\Sigma = \{0, 1\}$ Tape alphabet: $\Gamma = \{0, 1, \#, \sqcup\}$

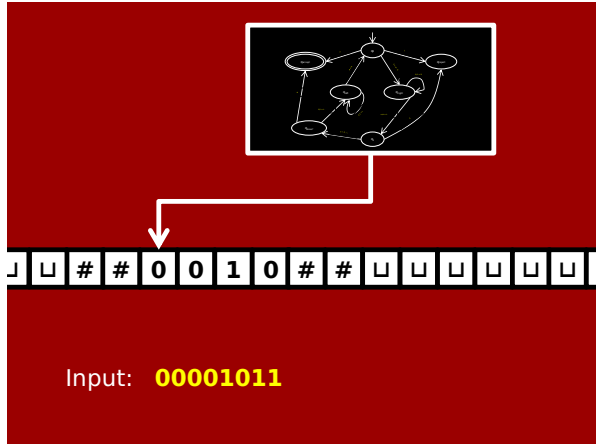
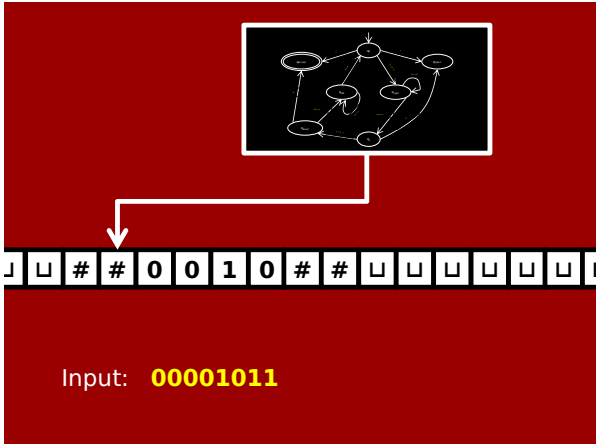
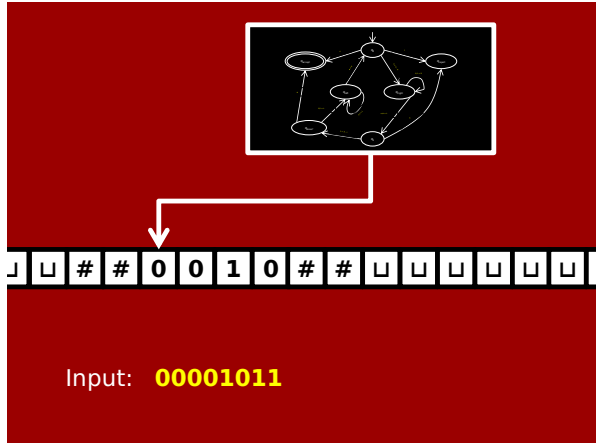
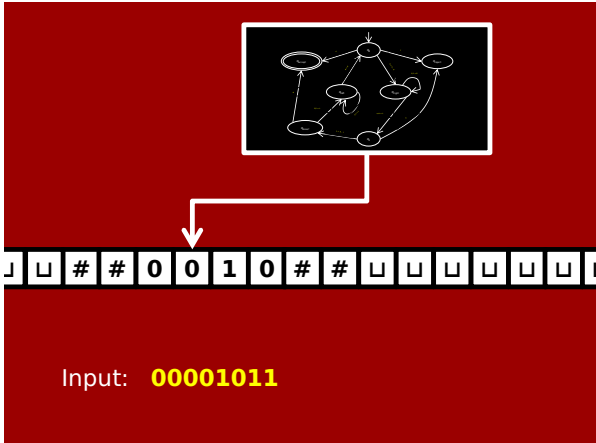
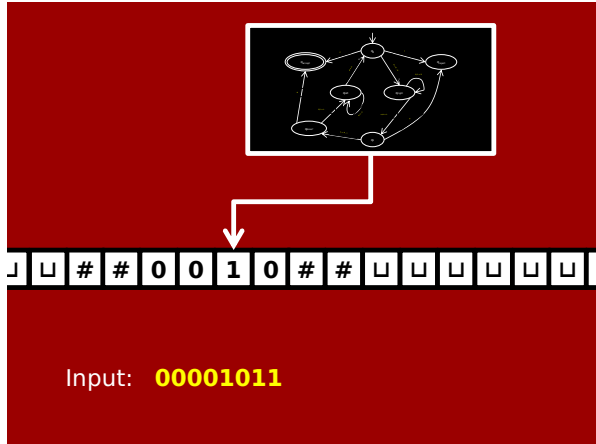
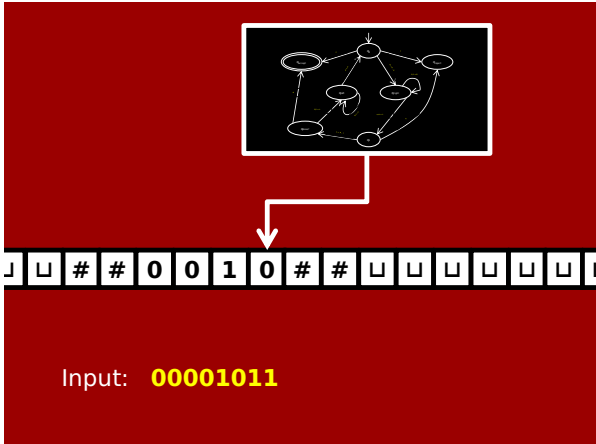


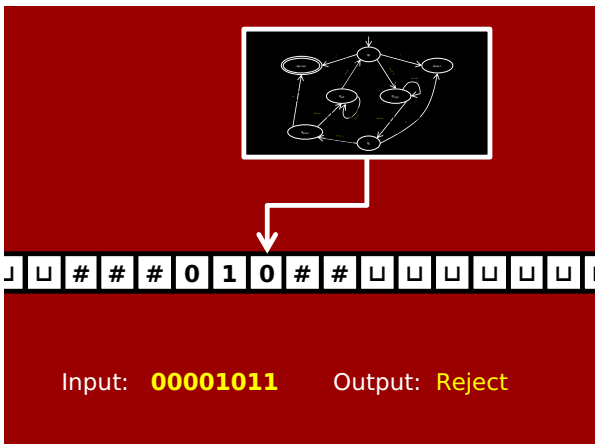
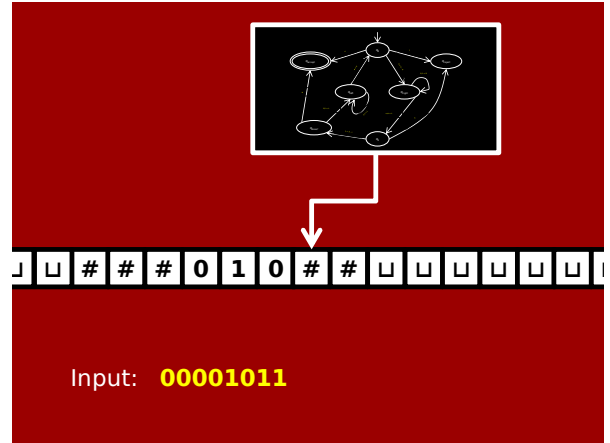
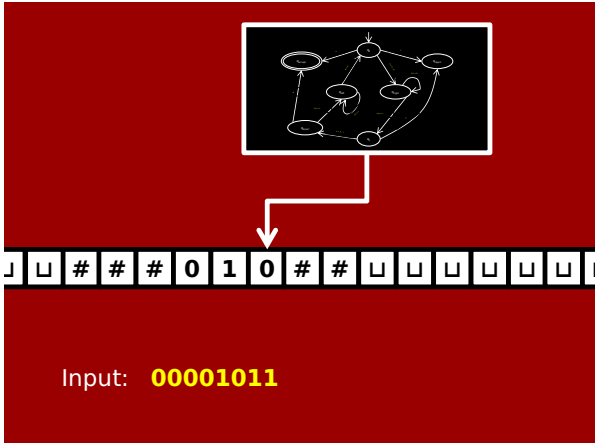
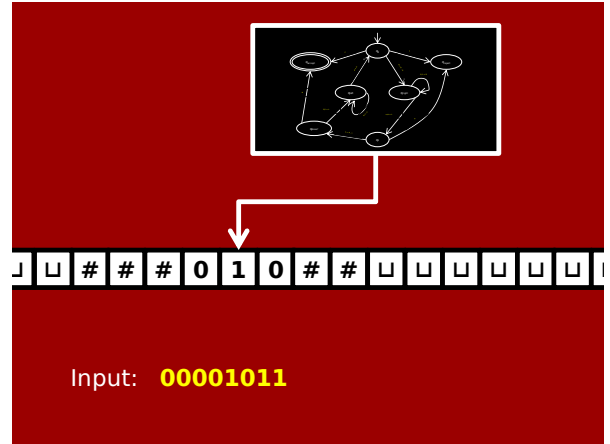
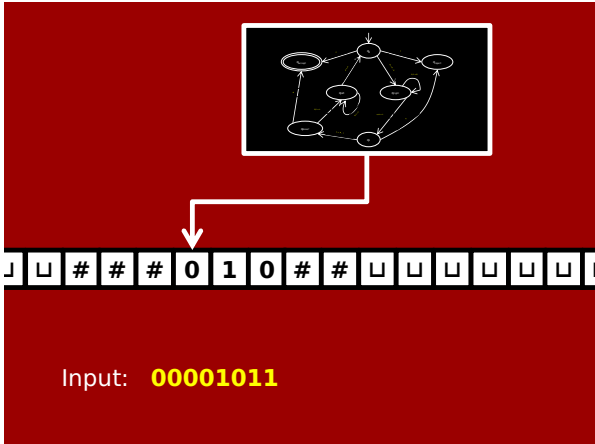












Programming with Turing Machines is tiresome.

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

Unfortunately for you, that time is this month.

Some TM subroutines and tricks

1. Move right (or left) until first \sqcup encountered
2. Shift entire input string one cell to the right
3. Convert input from $x_1x_2x_3\cdots x_n$ to $\sqcup x_1 \sqcup x_2 \sqcup x_3 \sqcup \cdots \sqcup x_n$
4. Simulate a big Γ by just $\{0,1,\sqcup\}$ (or even just $\{1,\sqcup\}$!)
5. "Mark" cells. If your tape alphabet is, say, $\{0,1,\sqcup\}$, extend it (or simulate extending it) to $\{0,1,0^*,1^*,\sqcup\}$. Treat 0^* , 1^* like 0 , 1 , but use marks to "remember" cells.
6. Copy a stretch of tape between two marked cells into another marked section of tape.

Some TM subroutines and tricks

7. Implement basic string & arithmetic operations
8. Simulate a TM with 2 tapes & read/write heads
9. Implement a dictionary (associative array) data structure
10. Simulate "random access memory"
11. ...
12. Simulate **COVM**, or some other simple bytecode.

Honest. It's not too hard to show all this. It just takes a little more time than we have.

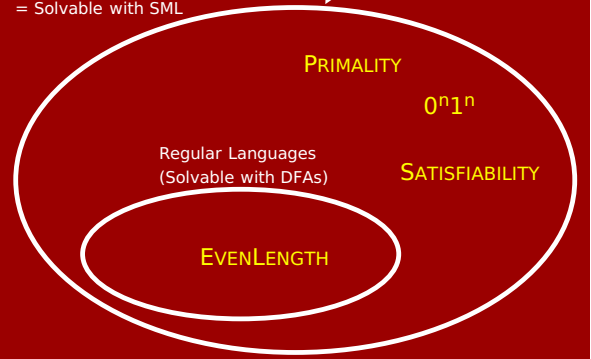
Conclusion:

Any algorithm written in Python, C, Java, SML, etc. can be simulated with a Turing Machine.

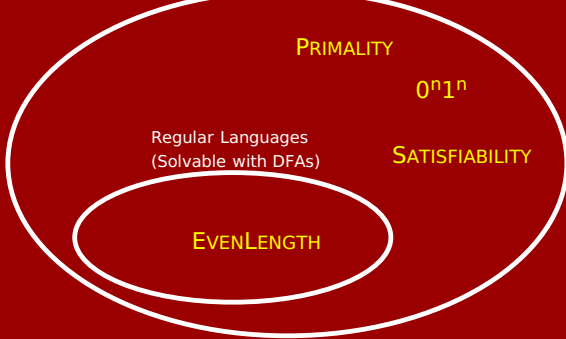
(In fact, even somewhat efficiently!)

Solvable with Python
= Solvable with C
= Solvable with Java
= Solvable with SML

What we want to define to be "computable".



Solvable with Python = Solvable with **TM**
= Solvable with C = "**Decidable**"
= Solvable with Java
= Solvable with SML



In particular:

You can write a TM interpreter with a TM.

I.e., there's a **Universal Turing Machine U**.
(There's even one with fewer than a dozen states!)

This TM **U** takes as input a *pair*, (M,x)

where **M** is a Turing Machine (encoded in some reasonable way as a string)
and **x** is a string...

... and **U** simulates $M(x)$.

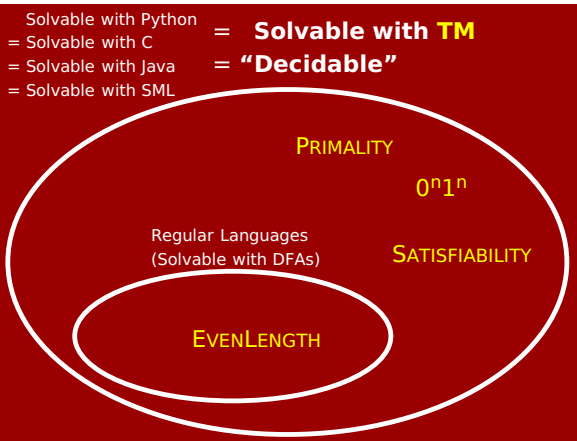
In other words **U(M,x)** accepts if $M(x)$ accepts,
U(M,x) rejects if $M(x)$ rejects,
U(M,x) loops if $M(x)$ loops.

If you don't believe me, you can consult pp.14–17 of Alan Turing's 1936 paper:



PS: at the time of writing, a “computer” meant a **person**, trained in calculation.

Computers in the age of Turing



Question:

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

Answer 1: It’s sort of hard to imagine.

Any new programming language would be originally written in Python / C / etc.

Question:

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

Answer 2: (from Turing’s 1936 paper)

Any notion of “computation” must be able to be carried out by a “computer” (i.e., a computing human!).

Turing justifies the TM by explaining why it can do anything a human could.

Church–Turing Thesis:

“Any natural / reasonable notion of computation can be simulated by a TM.”

This is not a theorem.

Is it... ..an observation?

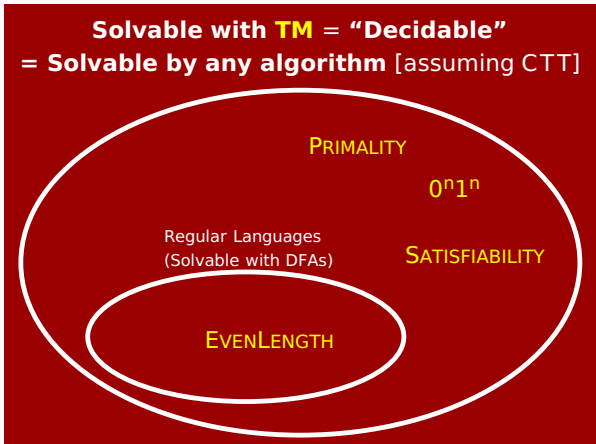
...a definition?

...a hypothesis?

...a law of nature/physics?

...a philosophical statement?

Well, whatever. Essentially everyone believes it.



Solvable with TM = "Decidable"
= Solvable by any algorithm [assuming CTT]

Question:
Is there **any** language which is **not** decidable?

Answer:
Also investigated in Turing's paper; we'll see the answer on Thursday!


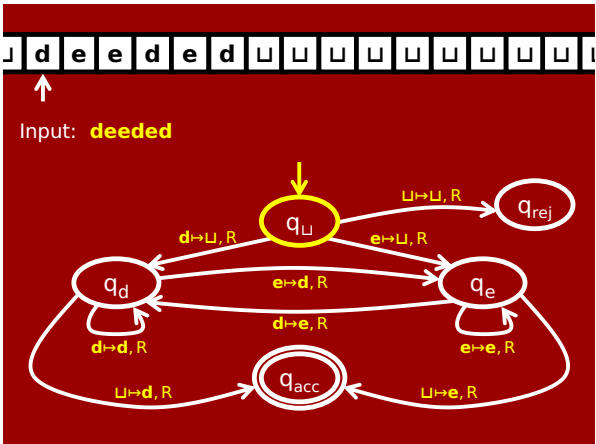
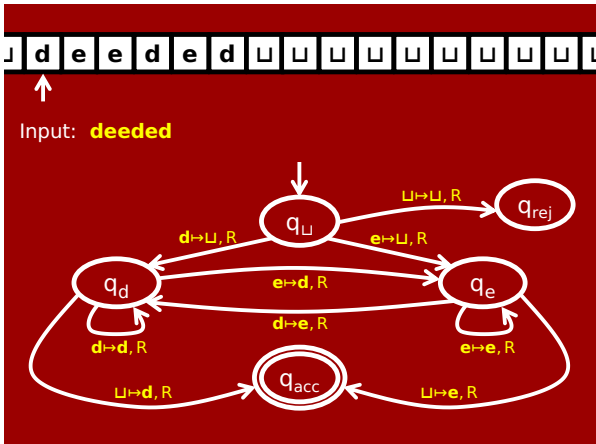
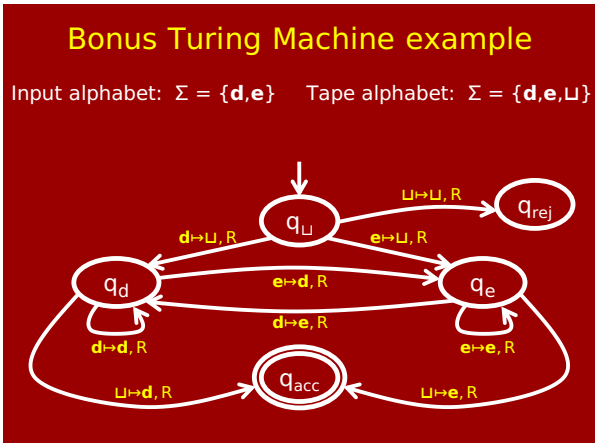
Study Guide

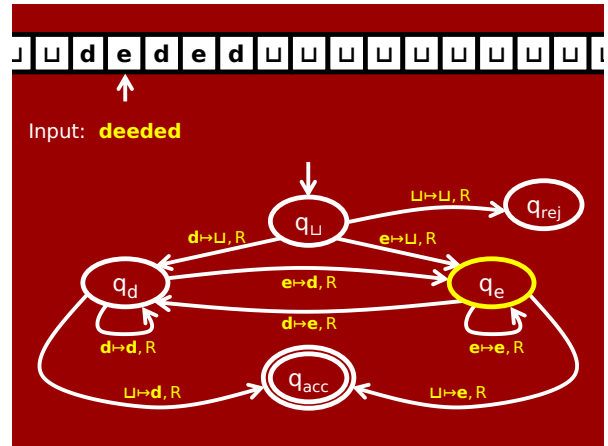
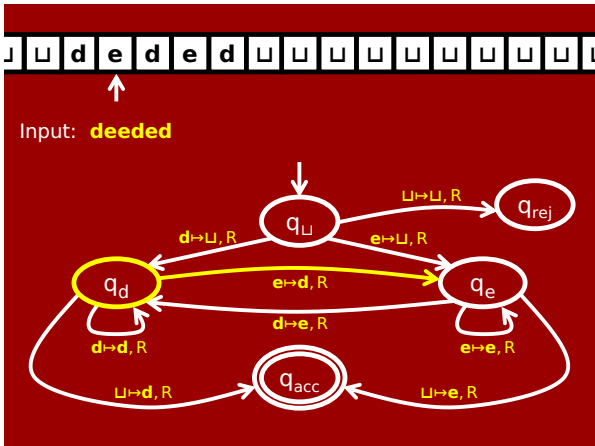
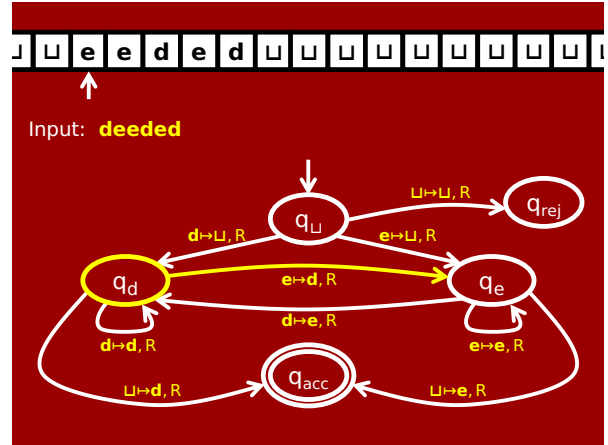
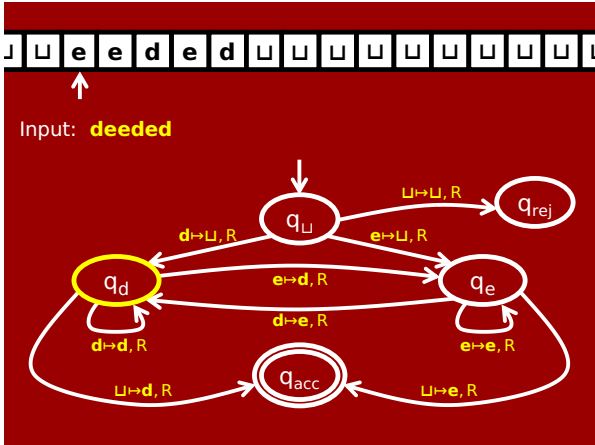
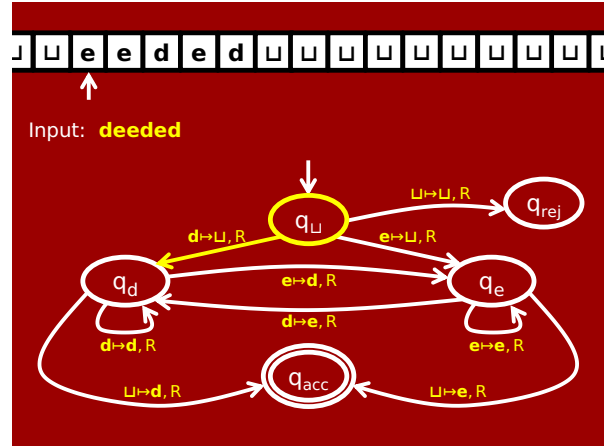
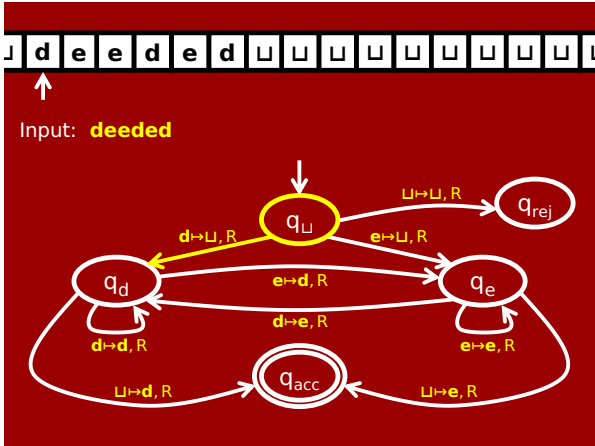
Definitions:

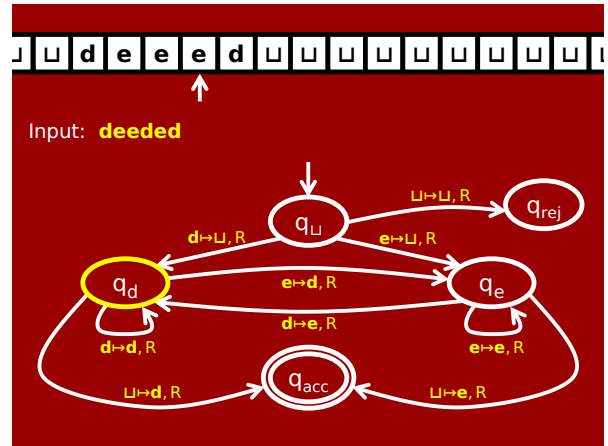
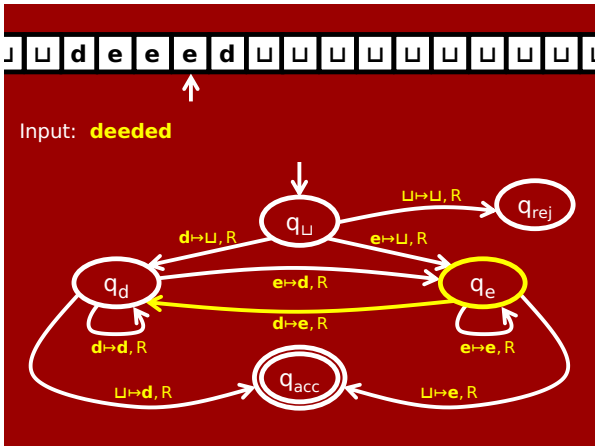
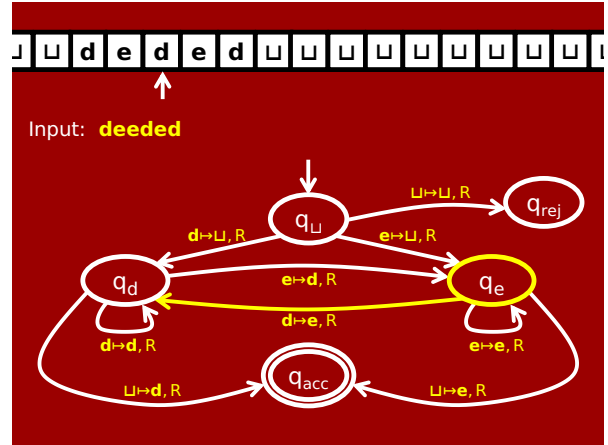
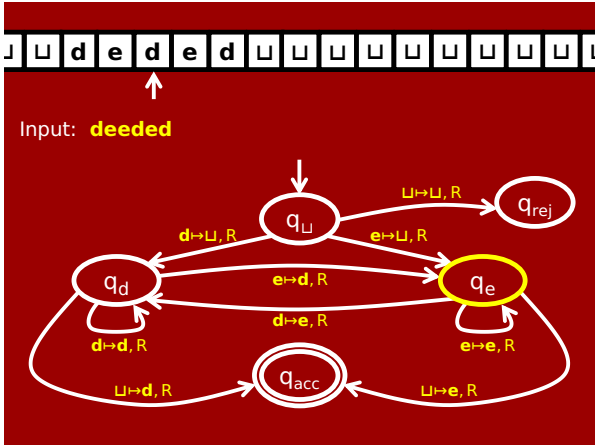
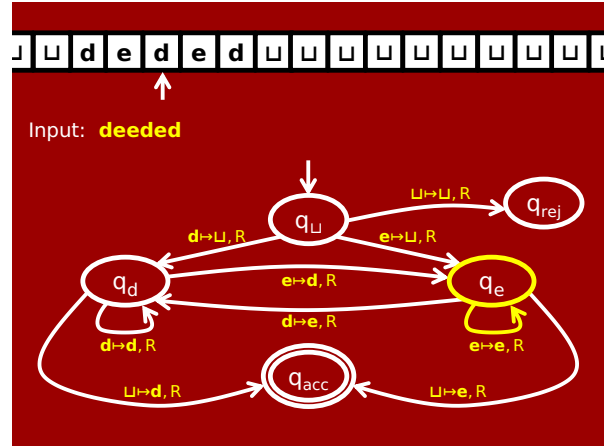
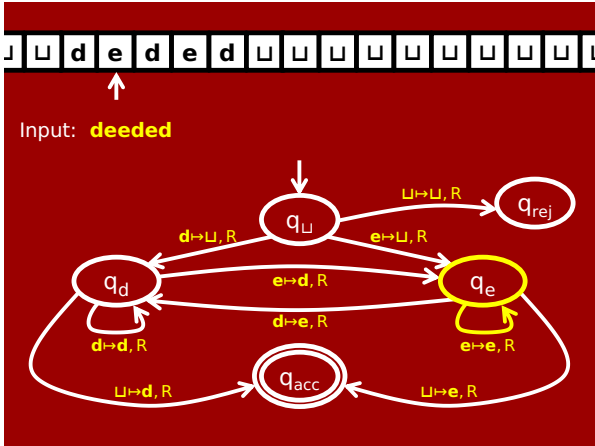
- TMs
- Interpreters & universal machines
- Deciders, decidable langs.
- Church-Turing Thesis

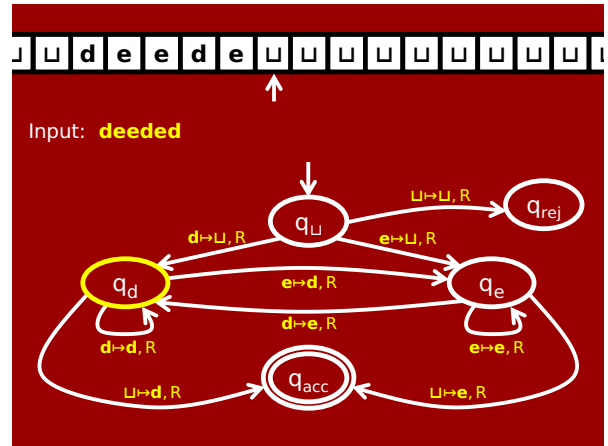
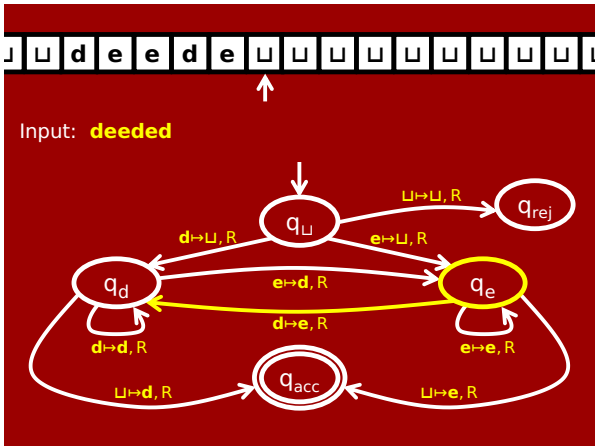
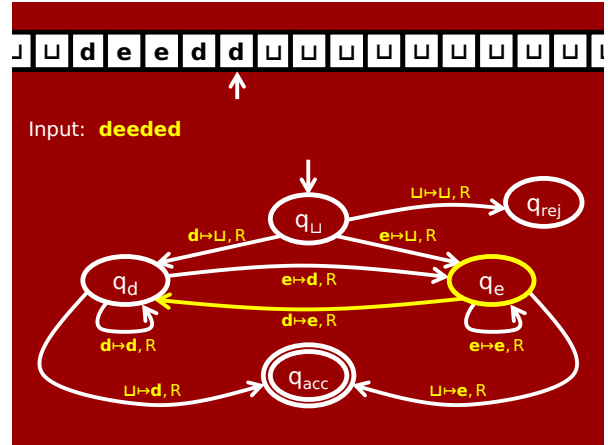
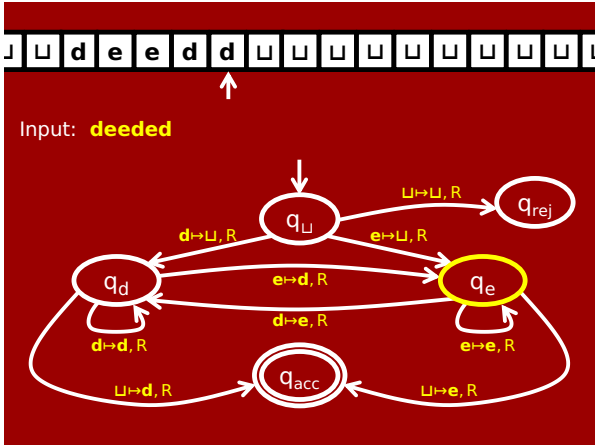
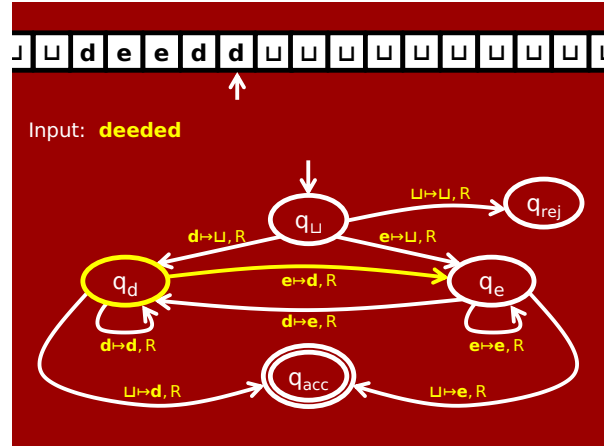
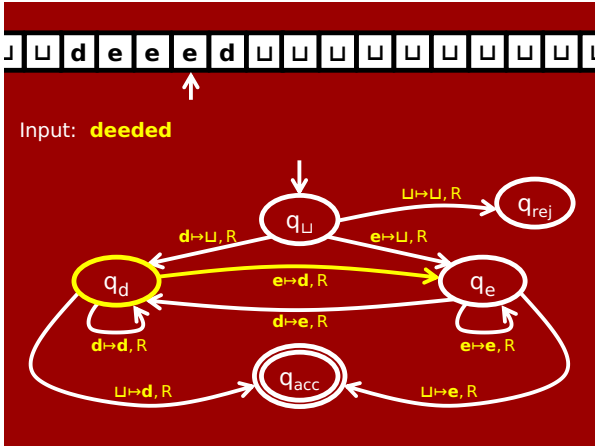
Practice:

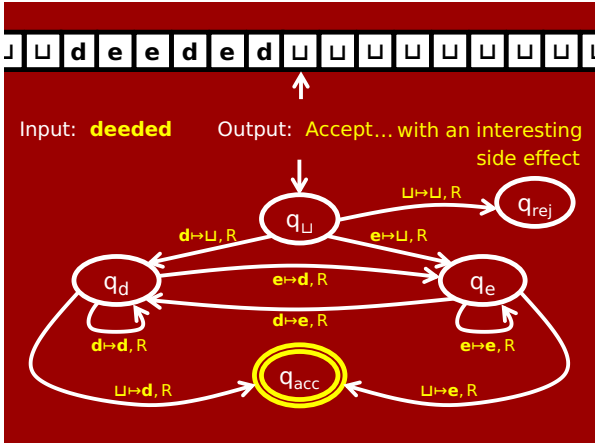
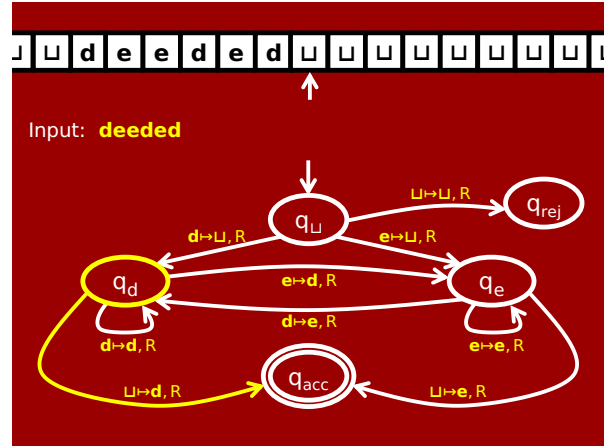
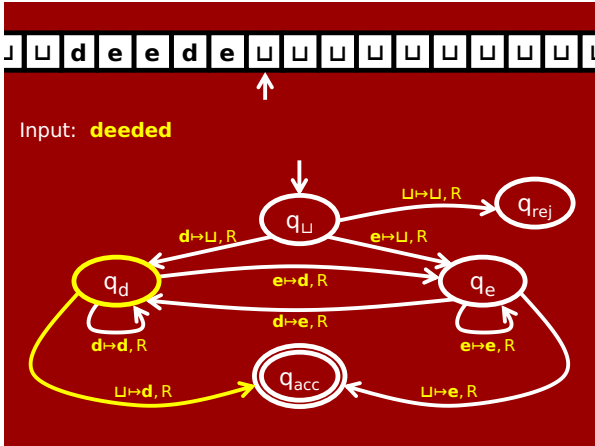
- Programming with TMs









Input: **deeded** Output: Accept... with an interesting side effect

Call this machine M .
 On input $x \in \{d, e\}^*$,

- $M(x)$ rejects if $x = \epsilon$.
- $M(x)$ accepts if $x \neq \epsilon$.

In particular, this M is a decider. $L(M) = \{x : x \neq \epsilon\}$.
 (Side effect: input string shifted right one cell.)