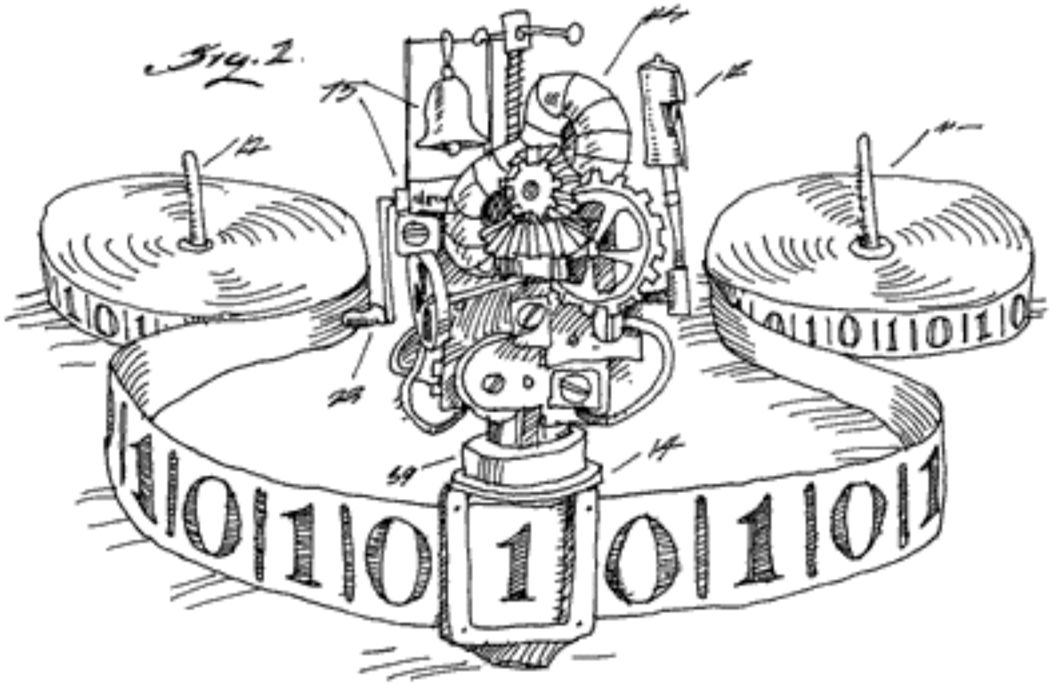
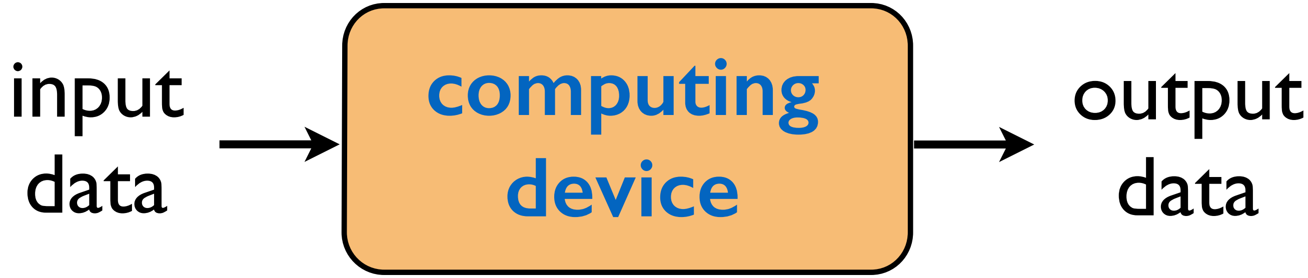
15-251
Great Theoretical Ideas in Computer Science

Lecture 4:
Turing's Legacy



September 10th, 2015

This Week



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.

+ | Primality | Sorting | DFA $|x|$ even

Identify these with the corresponding algorithms.

We only have machines to solve decision problems.

Definitions from last time

Σ^* = the set of all finite length strings over Σ

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

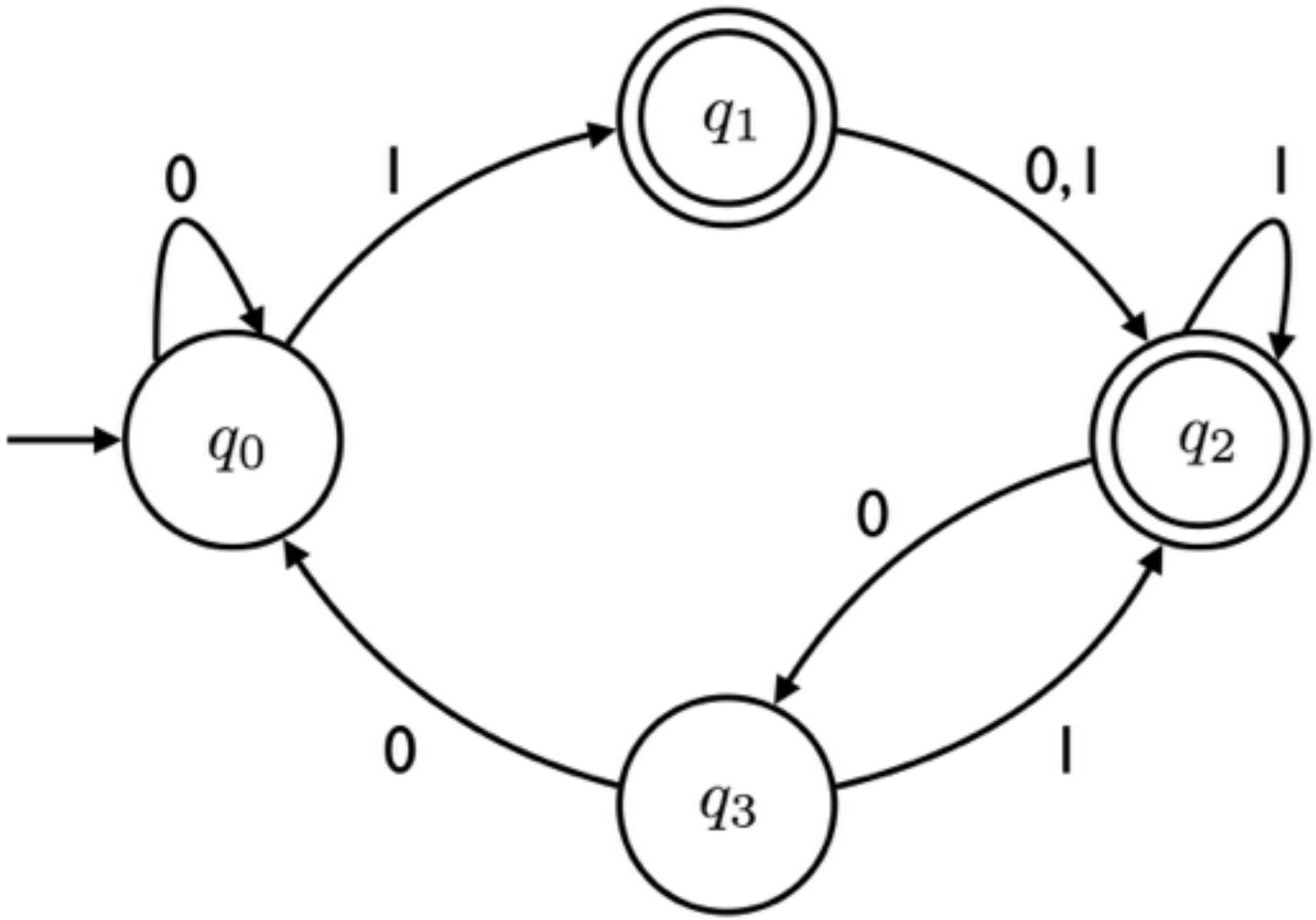
A *computational problem* is a function $f : \Sigma^* \to \Sigma^*$.

A *decision problem* is a function $f : \Sigma^* \to \{0, 1\}$.

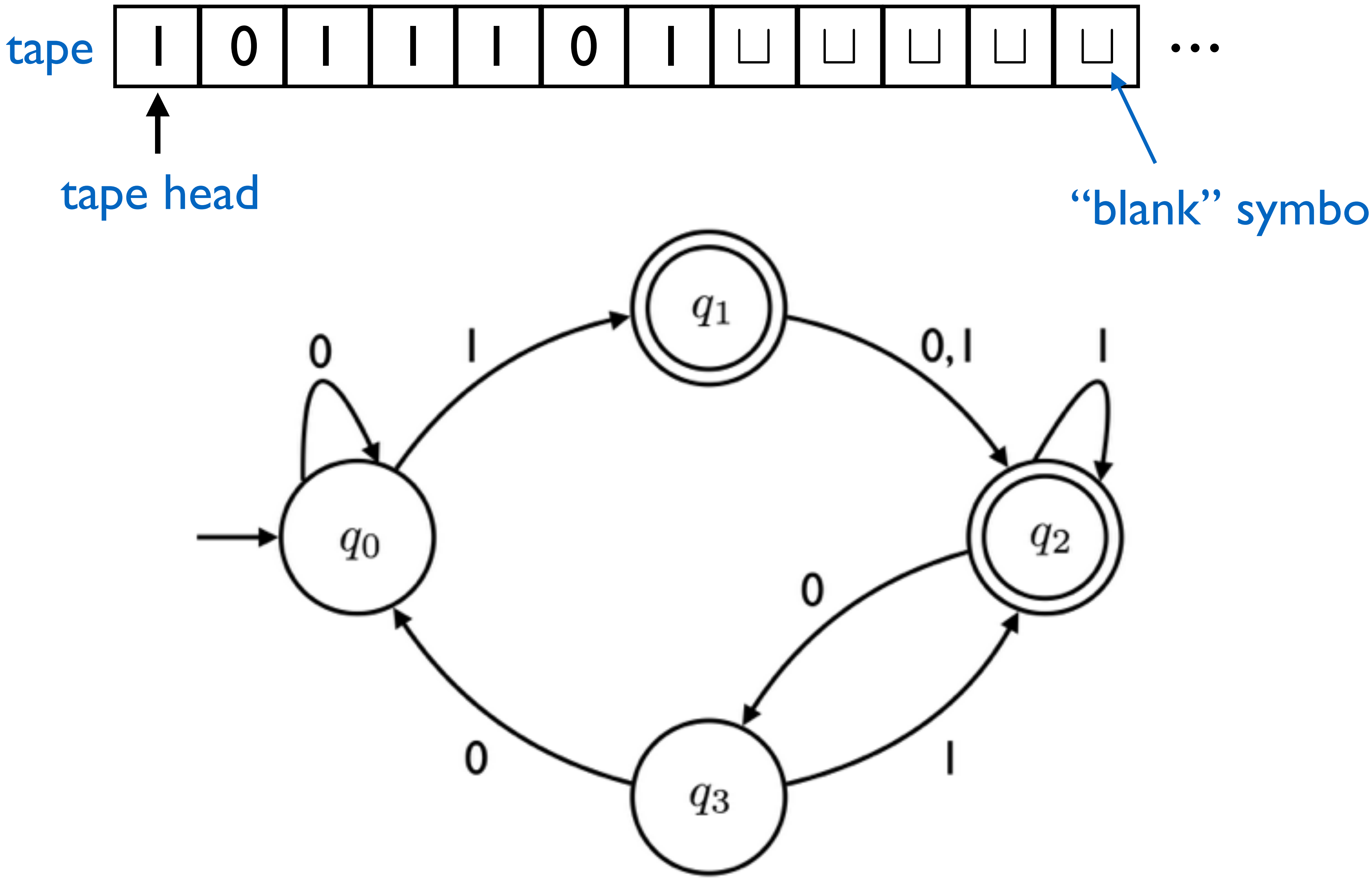
There is a one-to-one correspondence between decision problems and languages.

DFA state diagram

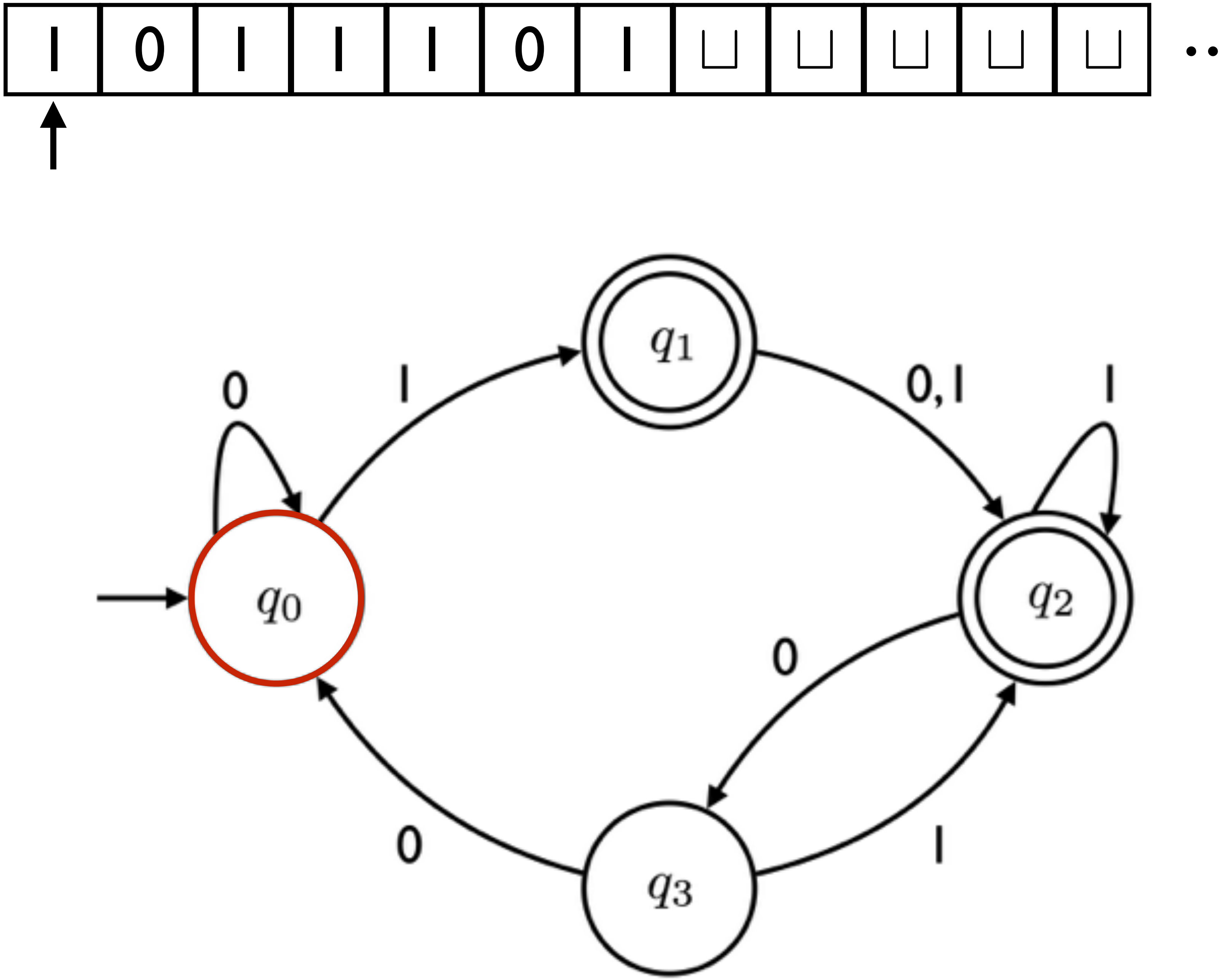
Input: 01111



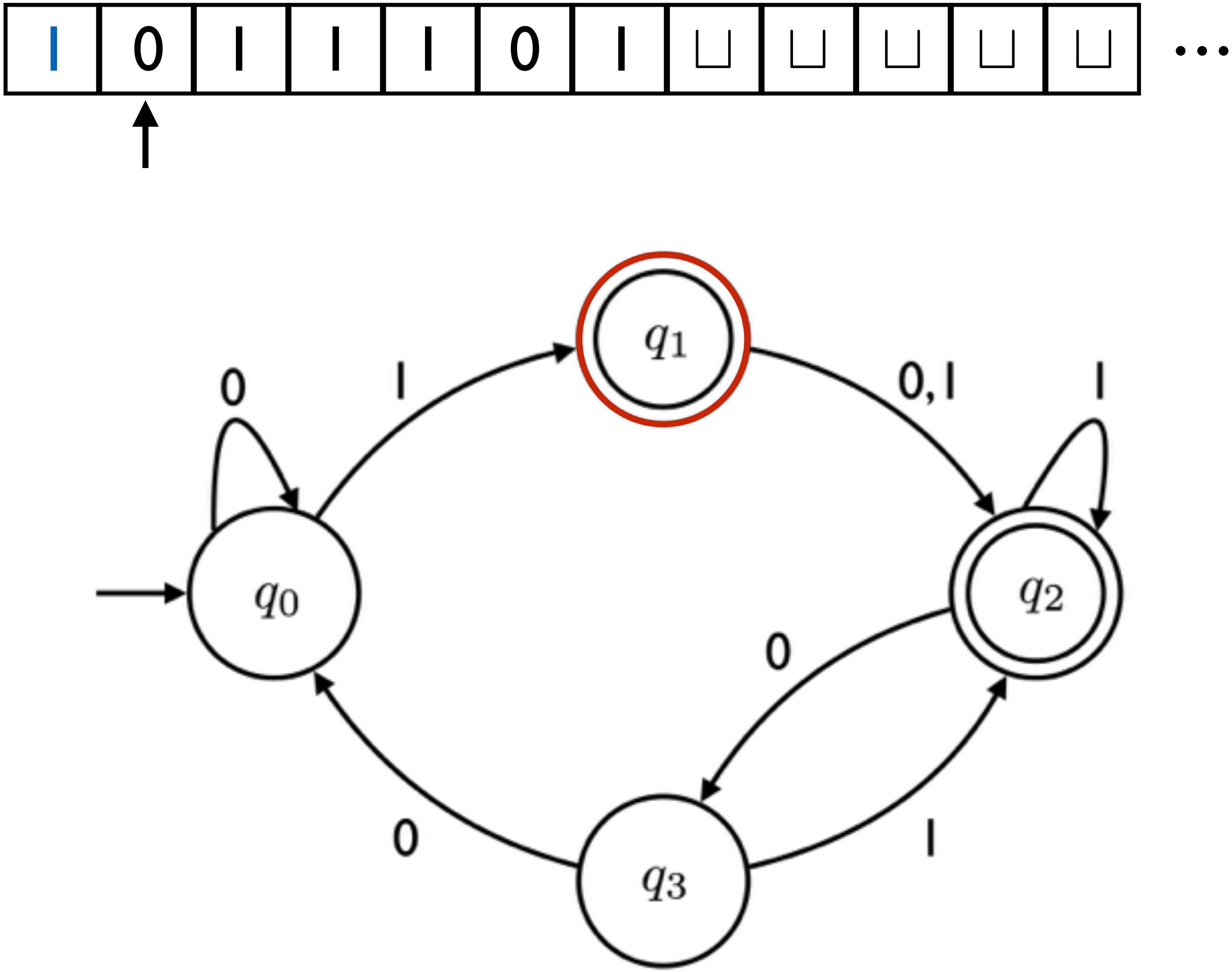
DFA: state diagram + input tape



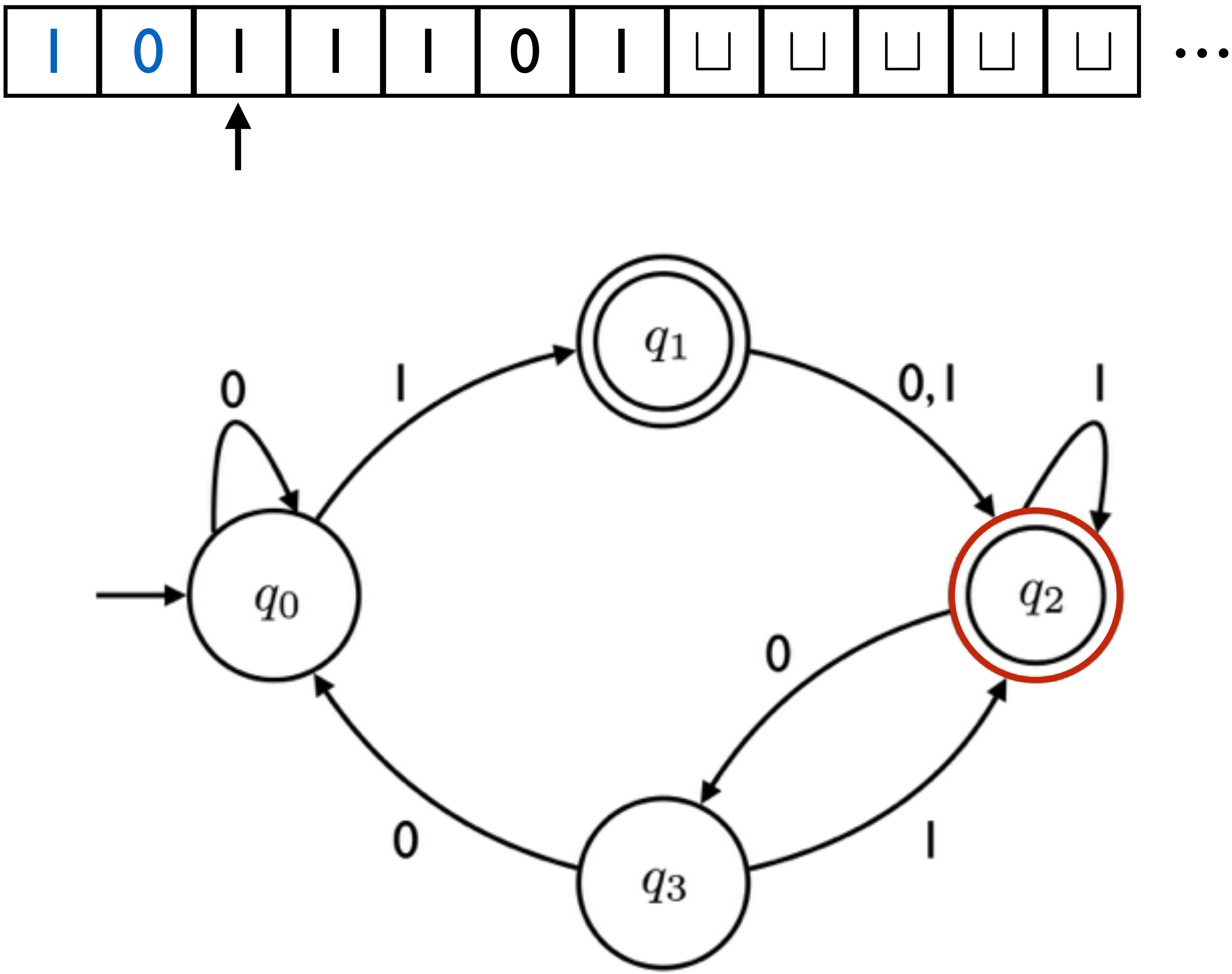
DFA: state diagram + input tape



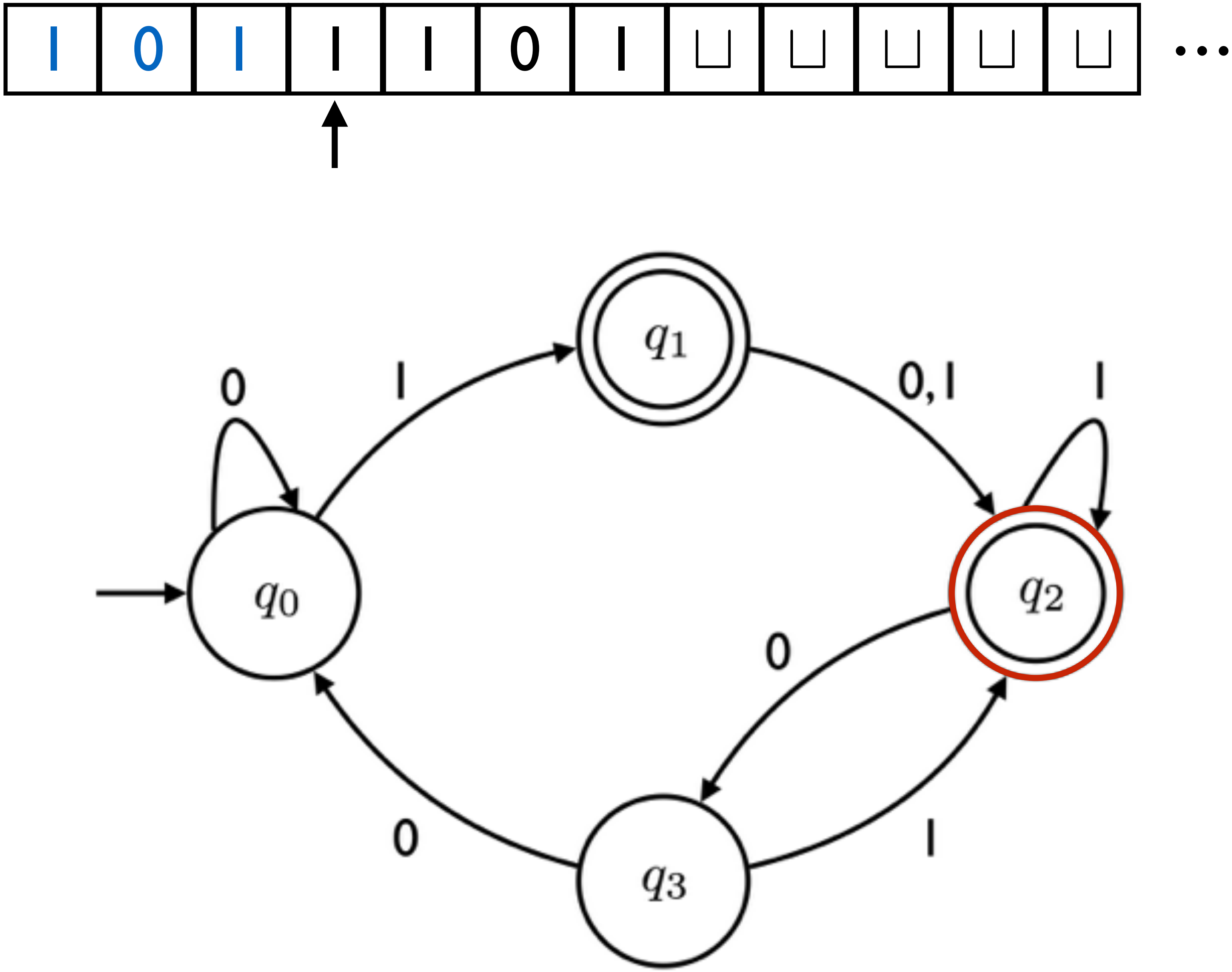
DFA: state diagram + input tape



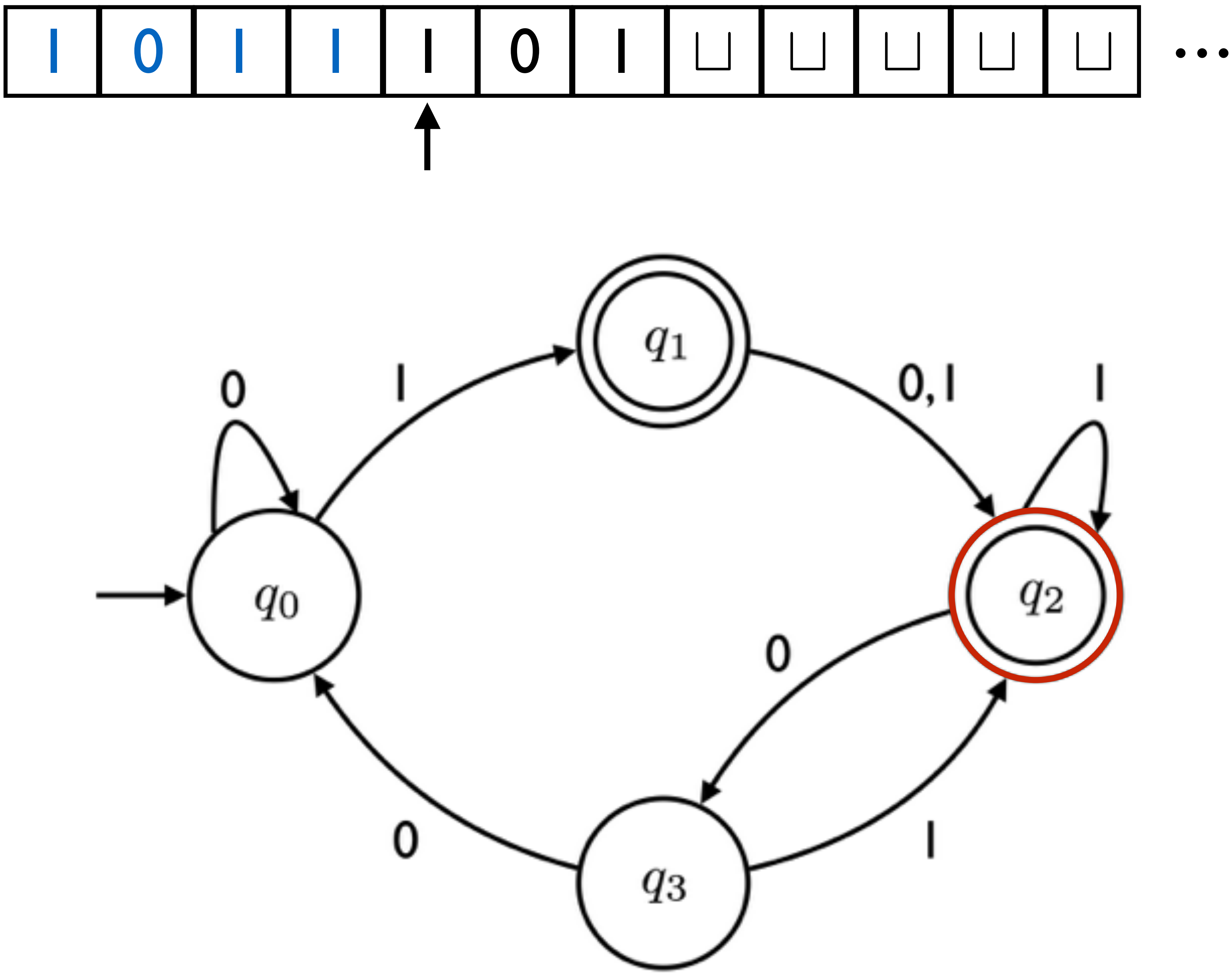
DFA: state diagram + input tape



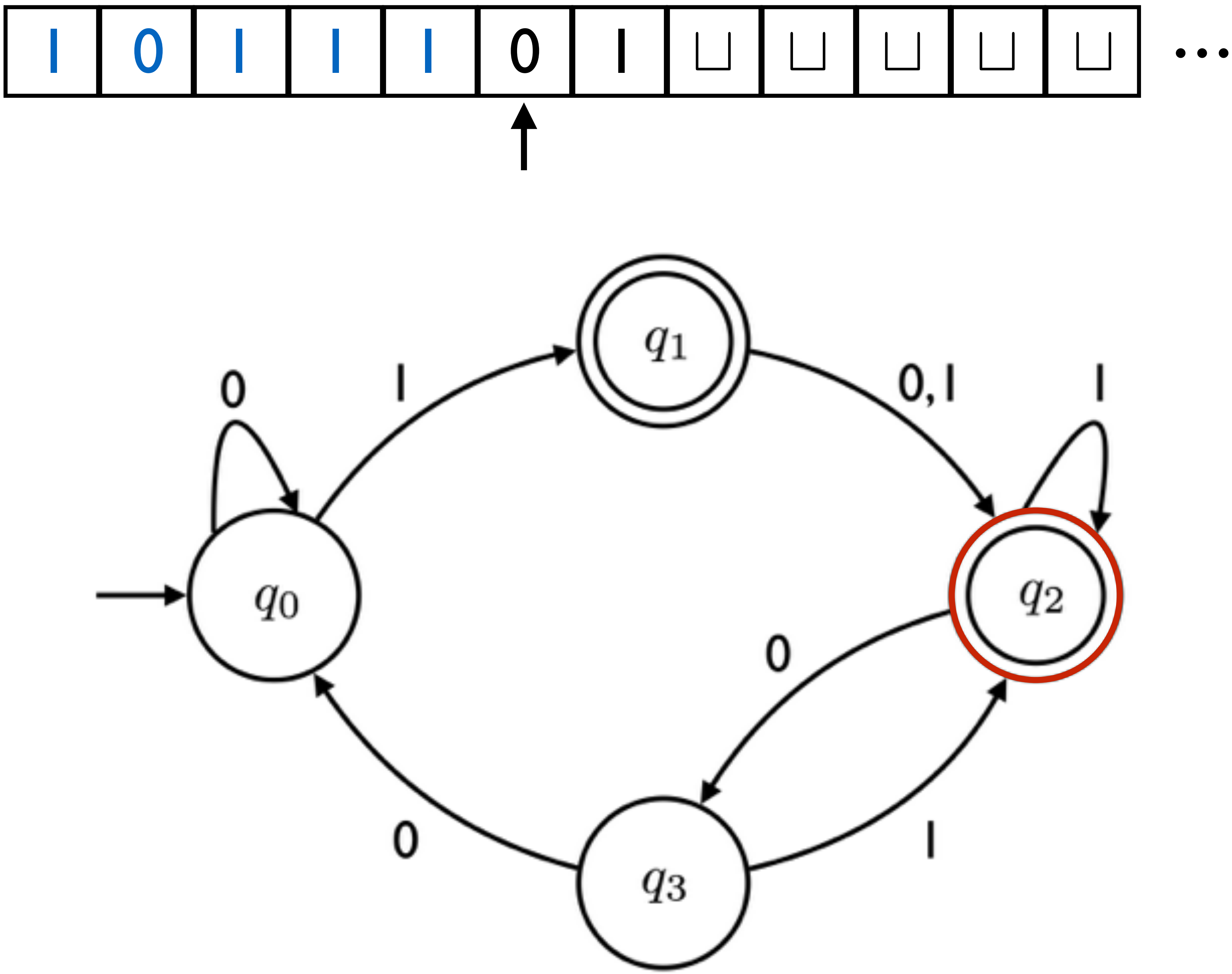
DFA: state diagram + input tape



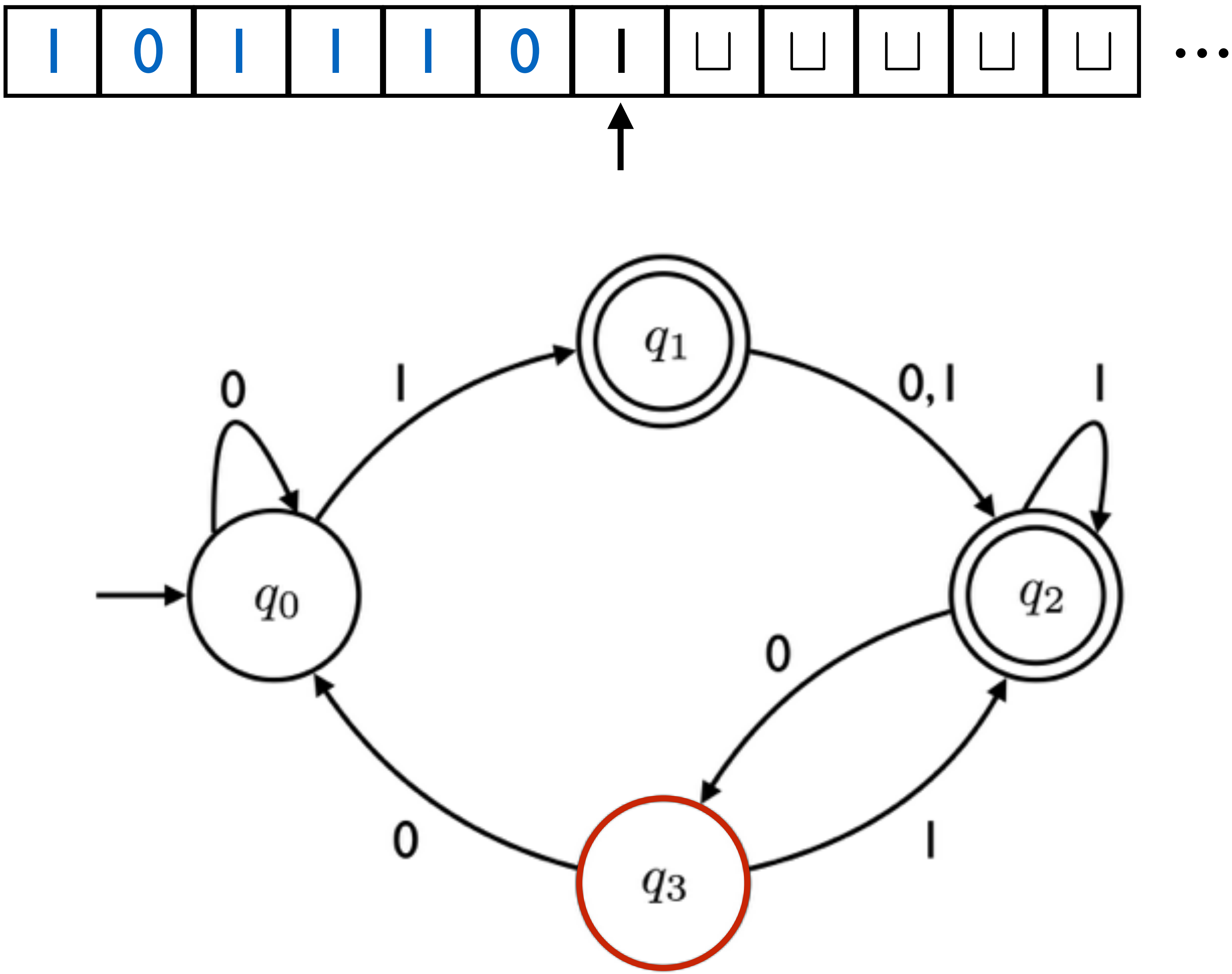
DFA: state diagram + input tape



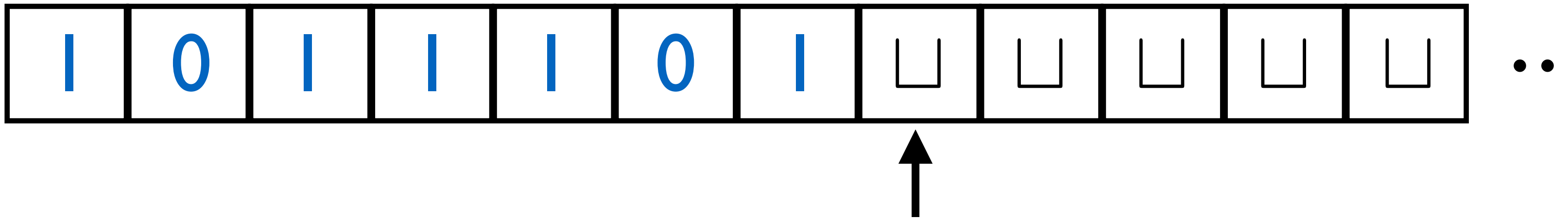
DFA: state diagram + input tape

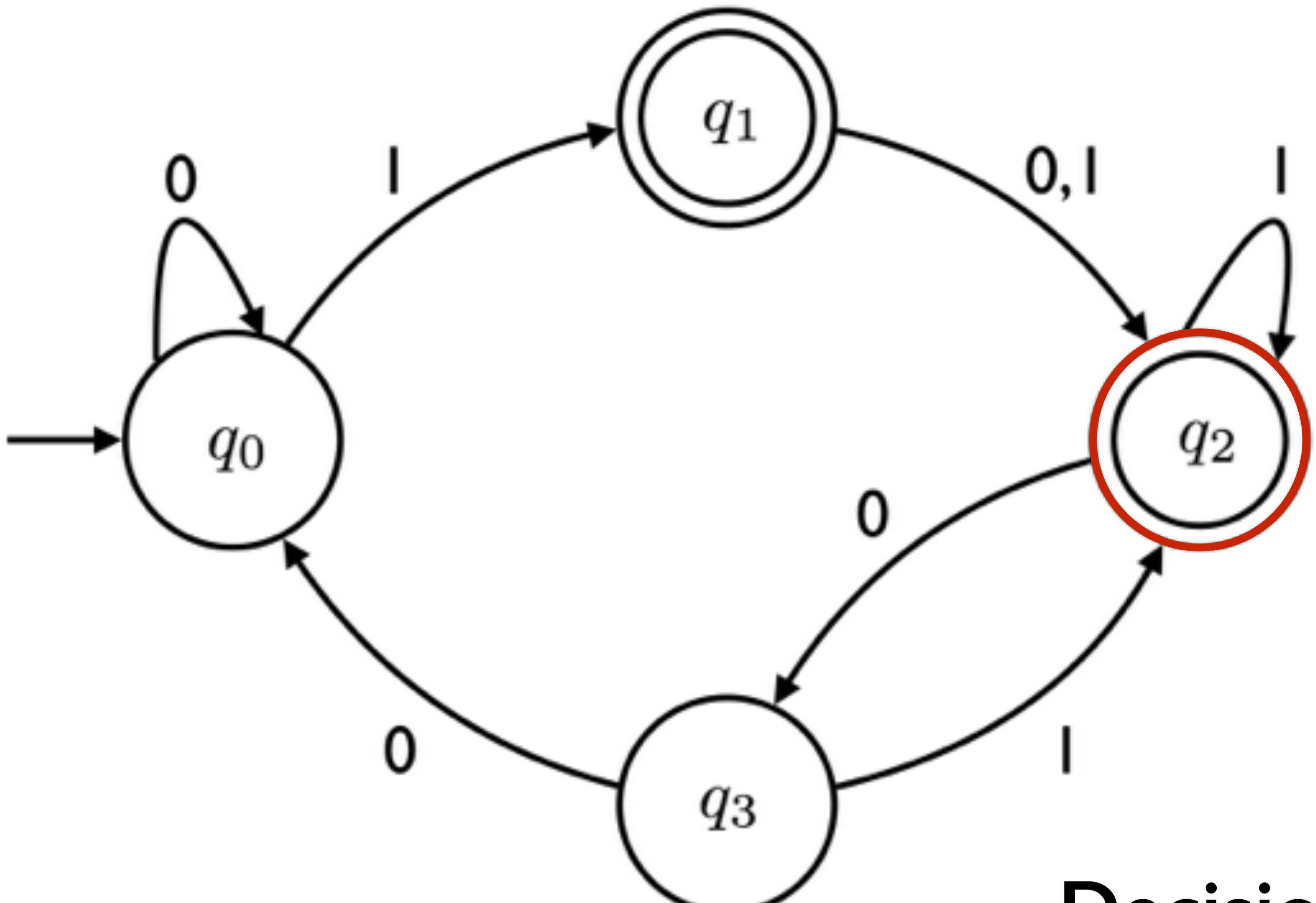


DFA: state diagram + input tape



DFA: state diagram + input tape

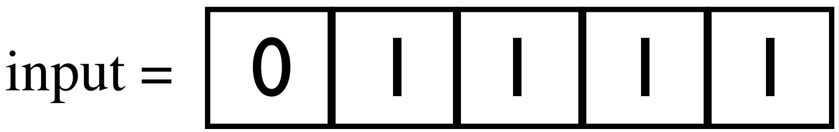


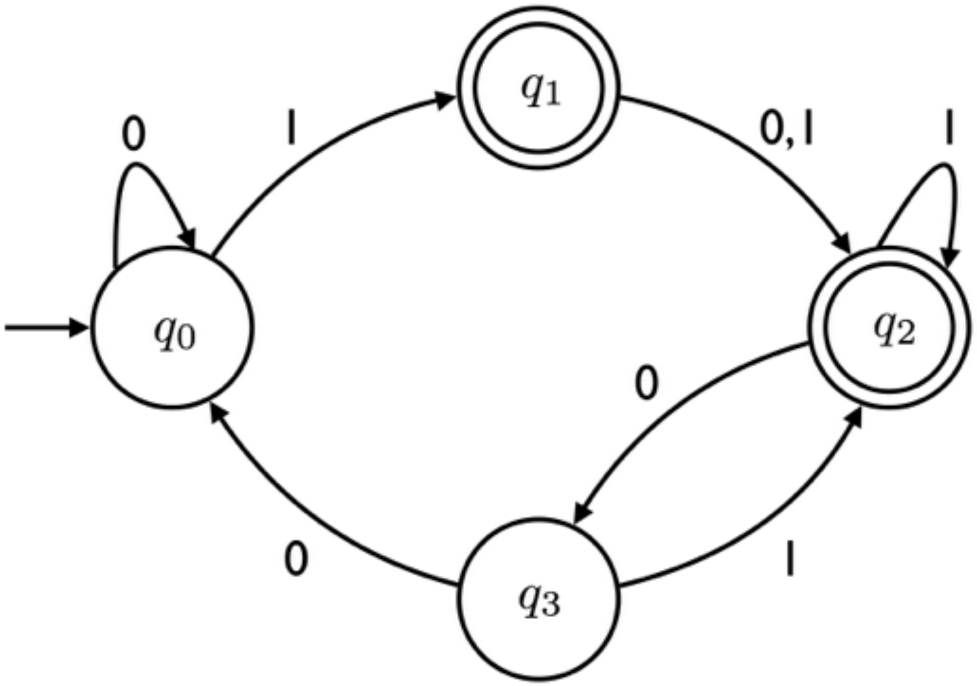


Decision: Accept

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;
    ...
```





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);
- Σ is a finite set (which we call the alphabet);
- δ is a function of the form $\delta : Q \times \Sigma \to Q$
  (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 Σ.

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: Regular languages

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

2 Theorems

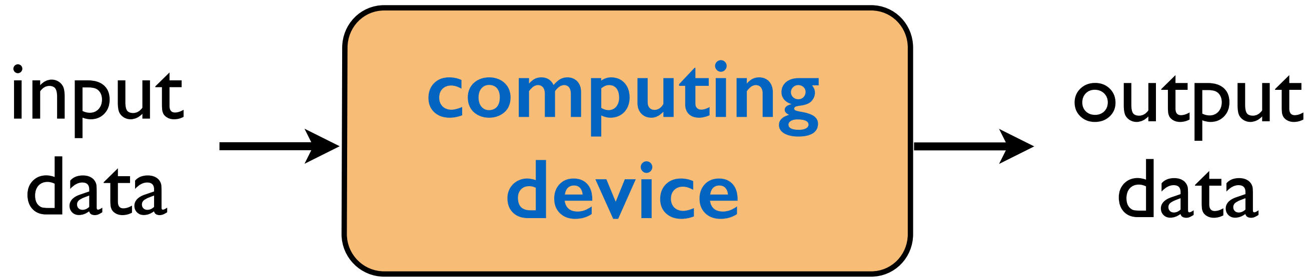
Theorem:

The language $L = \{0^n 1^n : n \in \mathbb{N}\}$ is **not** regular.

Theorem:

Let Σ 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?

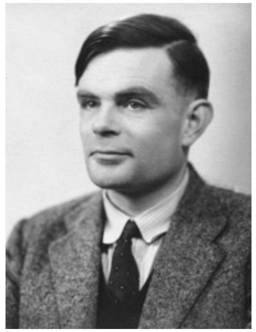
A specification that describes how information is transformed.

How can we mathematically define them?

The properties we want from the definition:

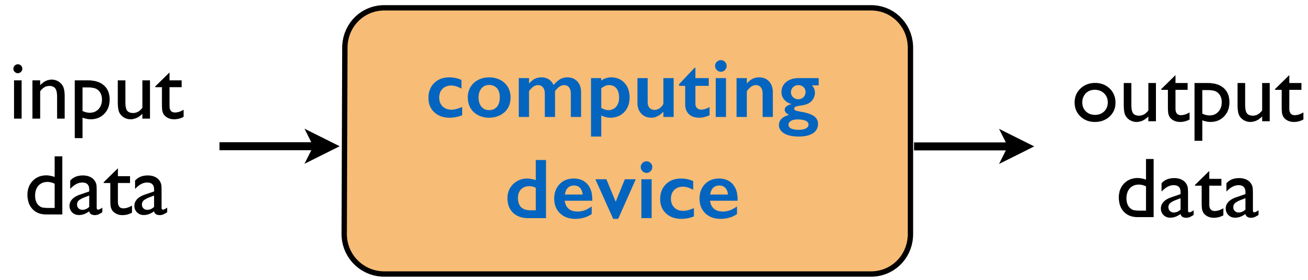
Simplicity! (the simpler the better)

Generality! (general enough to capture all of computation)



1900 1936 2015

Goal is to reach the definition of a Turing machine.



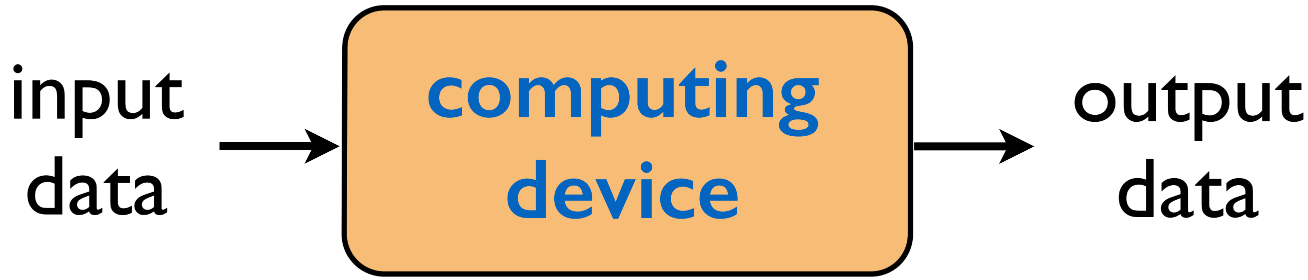
2 important observations:

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

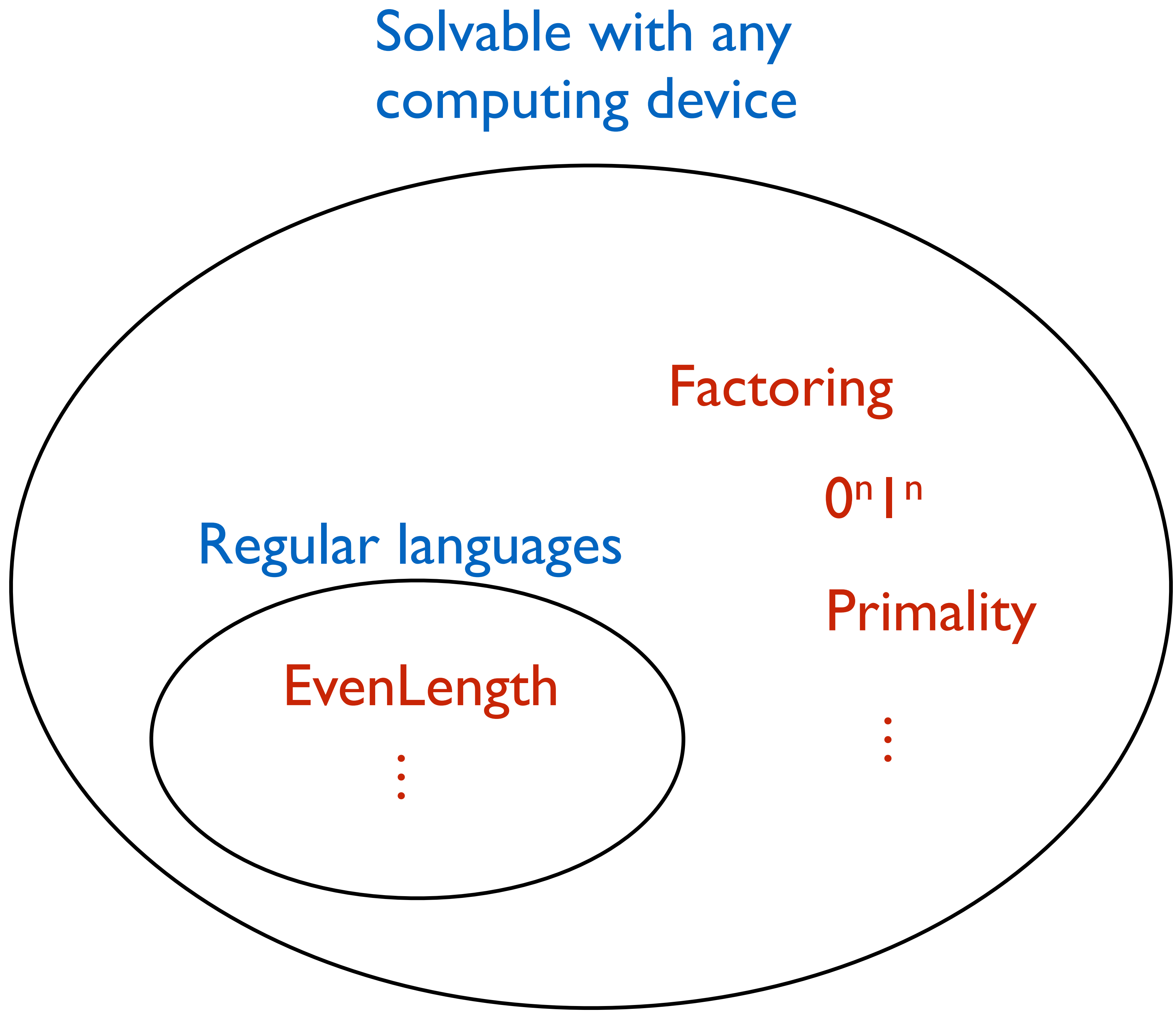


2 important observations:

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.

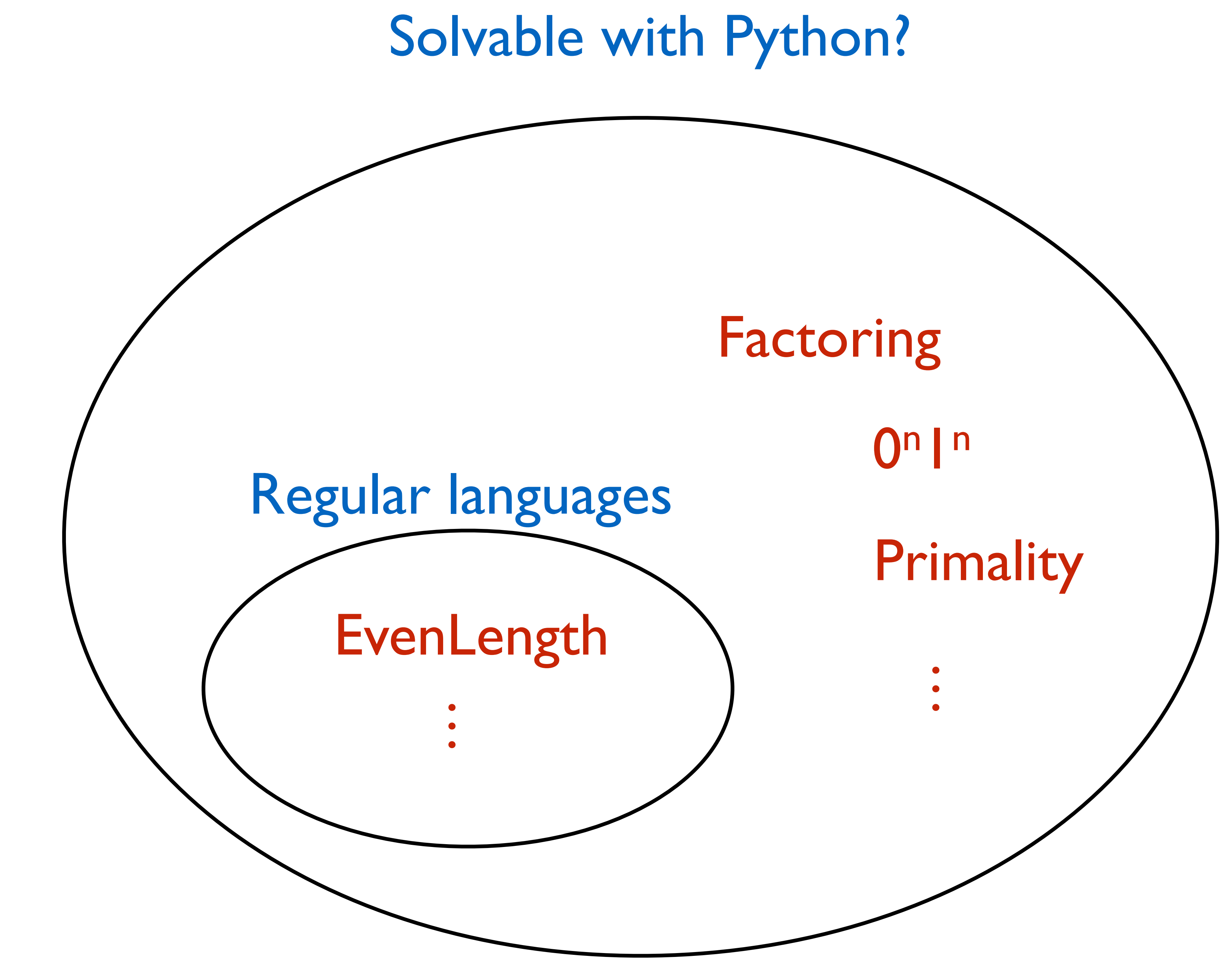

Solvable with any computing device
Factoring
0^n1^n
Regular languages
Primality
EvenLength
⋮
⋮

Solving $0^n 1^n$ in 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

```
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?
Factoring
0^n1^n
Regular languages
Primality
EvenLength
⋮
⋮

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)

<u>So what we want is:</u>

A totally minimal (TM) programming language such that

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

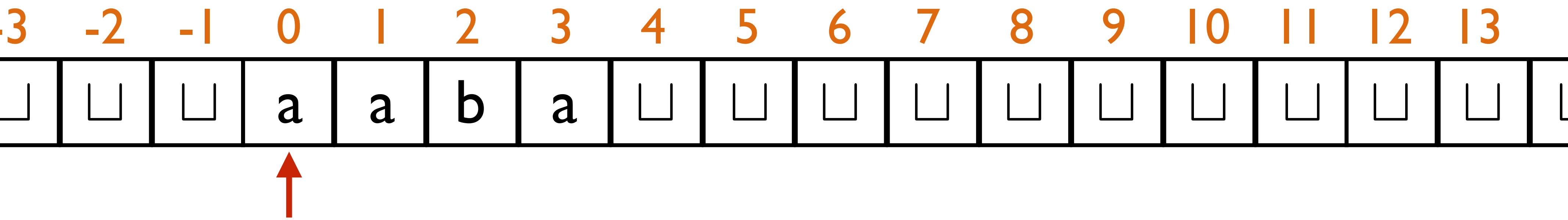
- it is simple to define and reason about completely
  mathematically rigorously

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 $\approx$ DFA + infinite tape



Input is written on the tape starting at index 0.

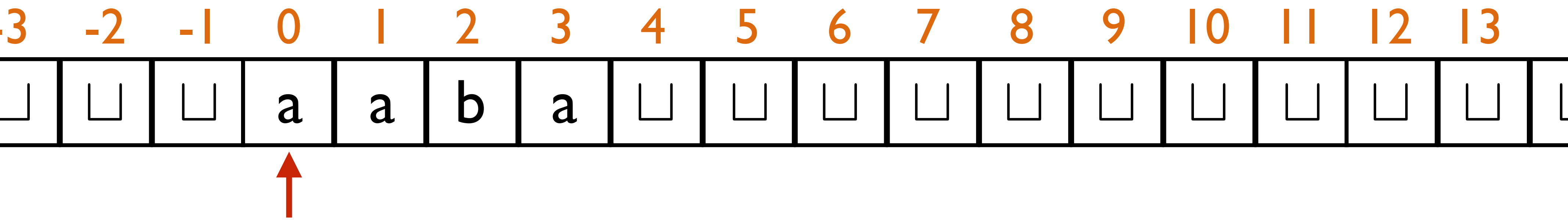
All other cells contain the *blank* symbol ⊔ .

There is a tape pointer/head (initially at position 0).

You can read & write to the tape cell pointed to.

Turing machine description

TM $\approx$ DFA + infinite tape



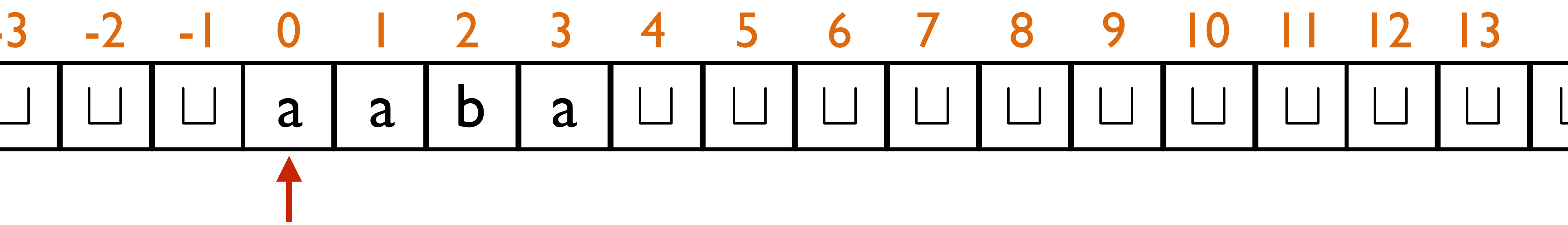
TM could have been defined as a sequence of instructions, where the allowed instructions are:

> 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 def'n as simple as possible.

Turing machine description

TM ≈ DFA + infinite tape



So a TM is a sequence of steps (states), each looking like:

STATE 0:
    switch(letter under the head):
        case ‘a’: **write** ‘b’; **move** Left; **go to** STATE 2;
        case ‘b’: **write** ‘␣’; **move** Right; **go to** STATE 0;
        case ‘␣’: **write** ‘b’; **move** Left; **go to** STATE 1;

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:

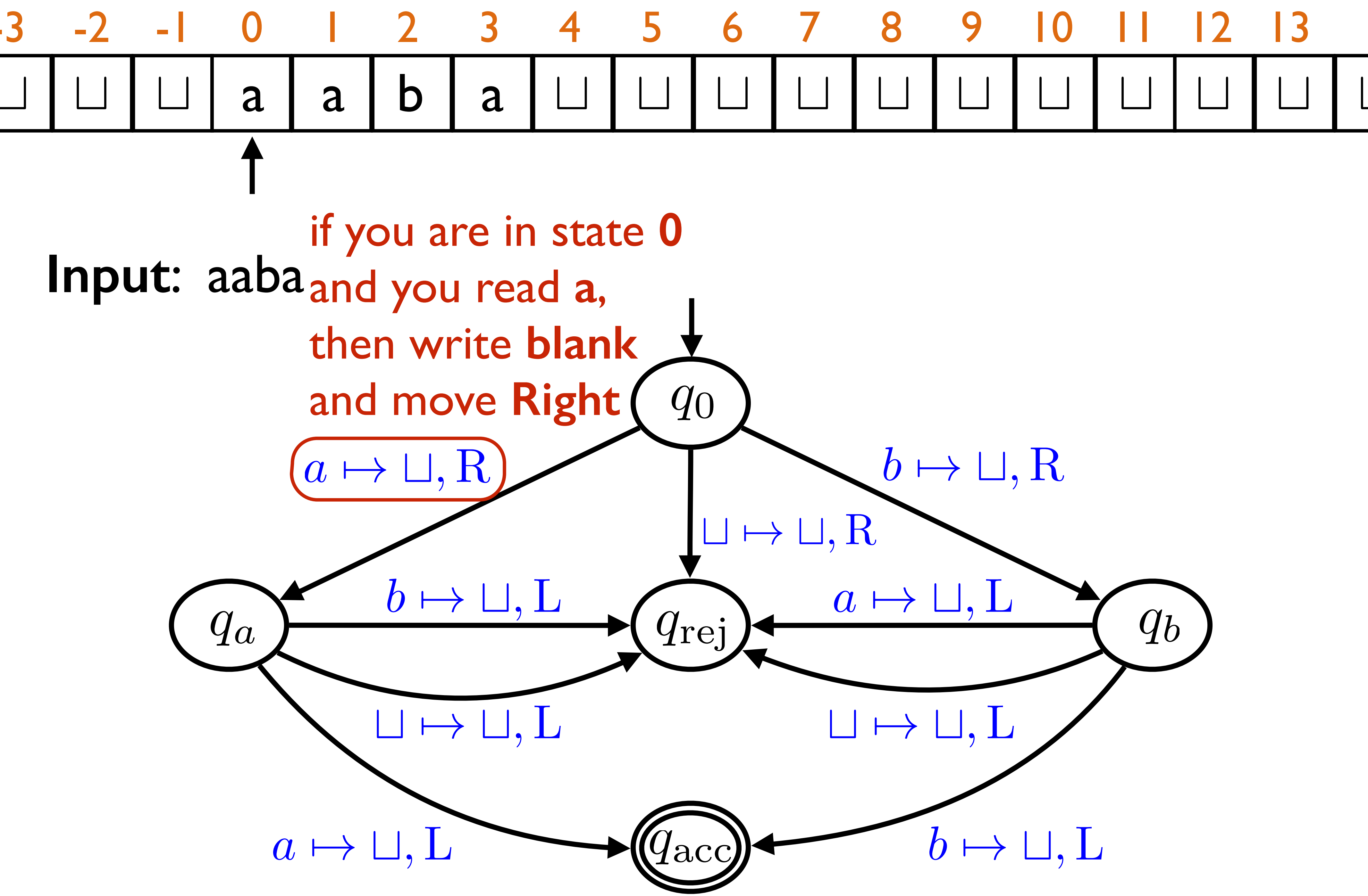
- write a *new* symbol to the cell under the head
- move tape head Left or Right
- go to a *new* state

Don’t want to change the symbol: **write** the same symbol.

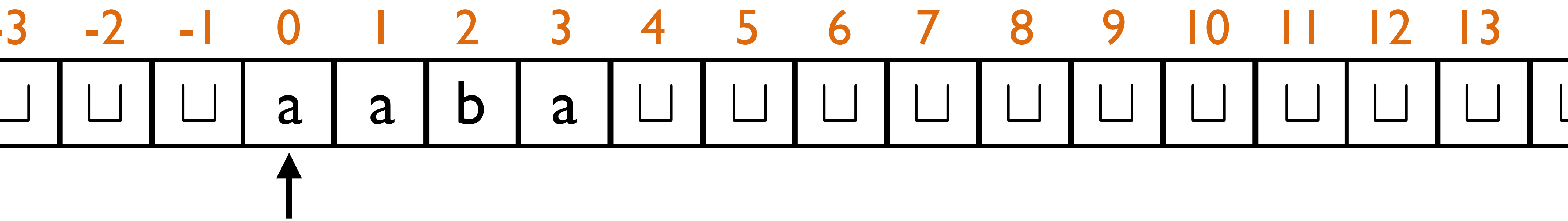
Want to stay put: **move** Left then Right.

Don’t want to change state: **go to** the same state.

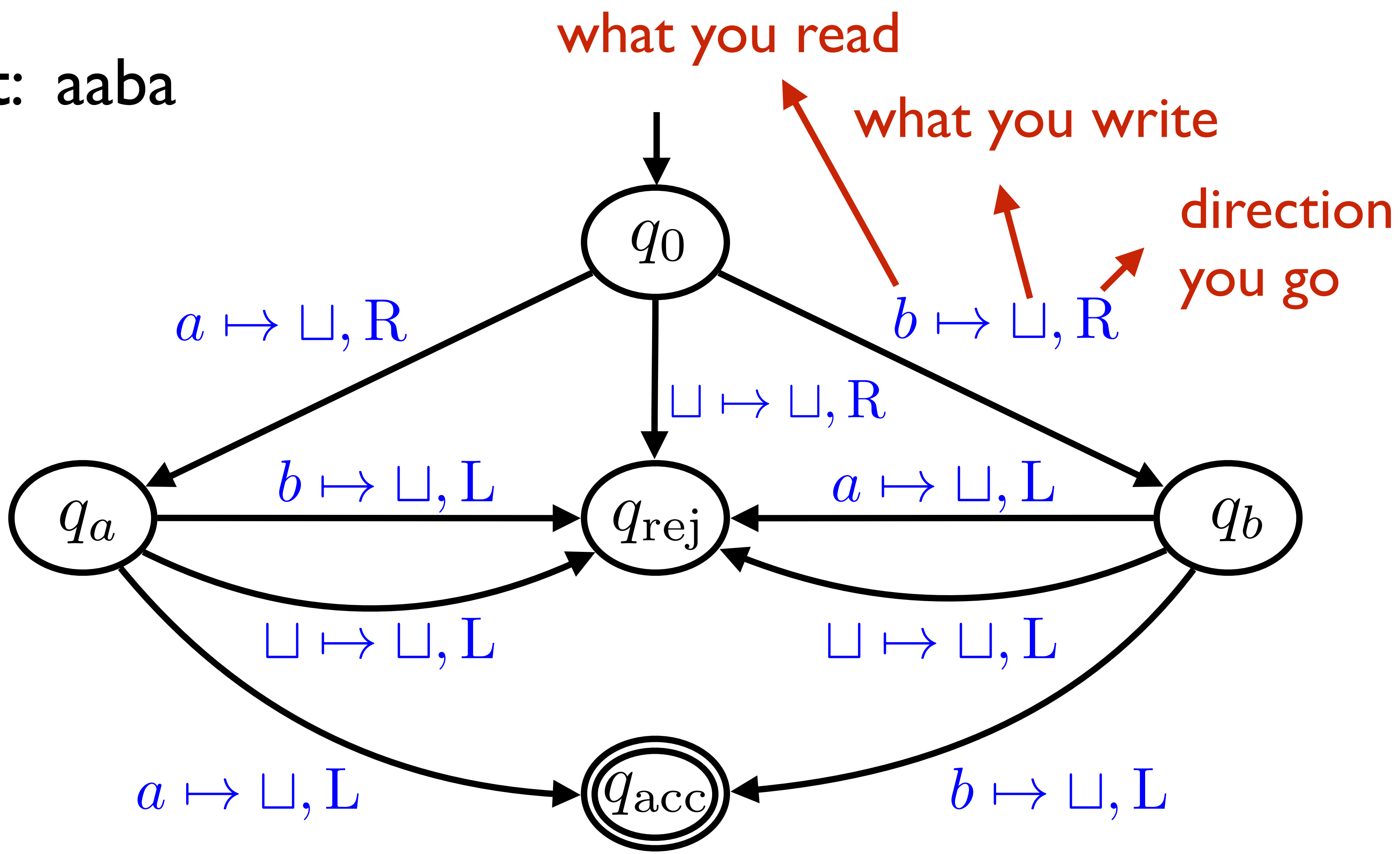
Turing machine official picture



Turing machine official picture



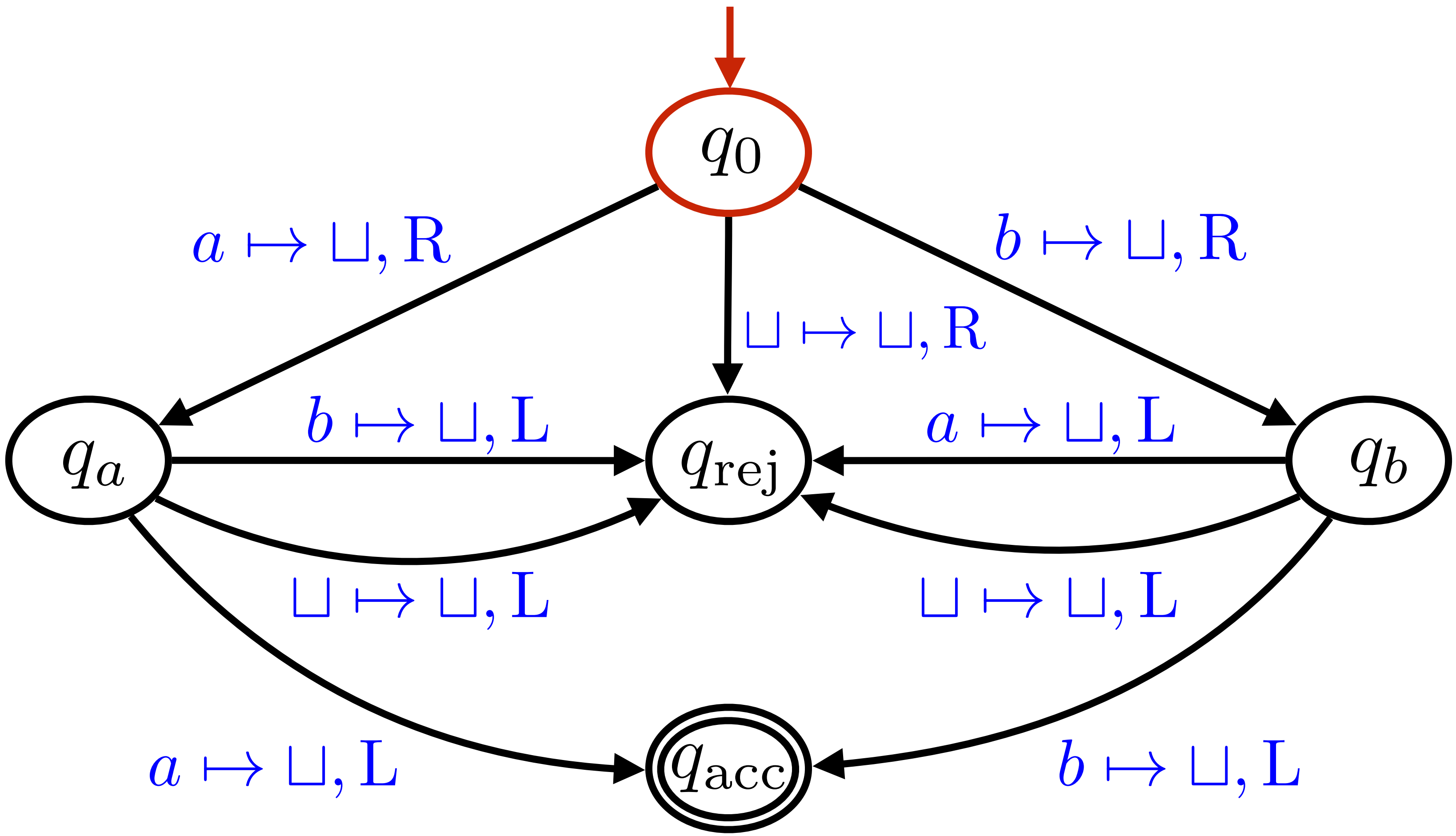
Input: aaba



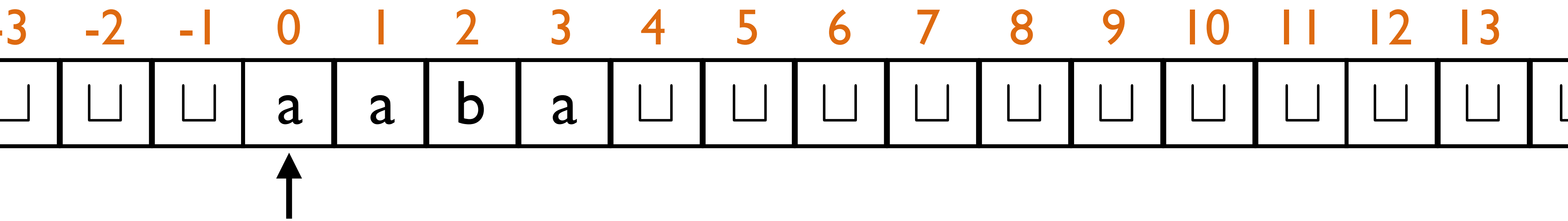
Turing machine simulation example

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

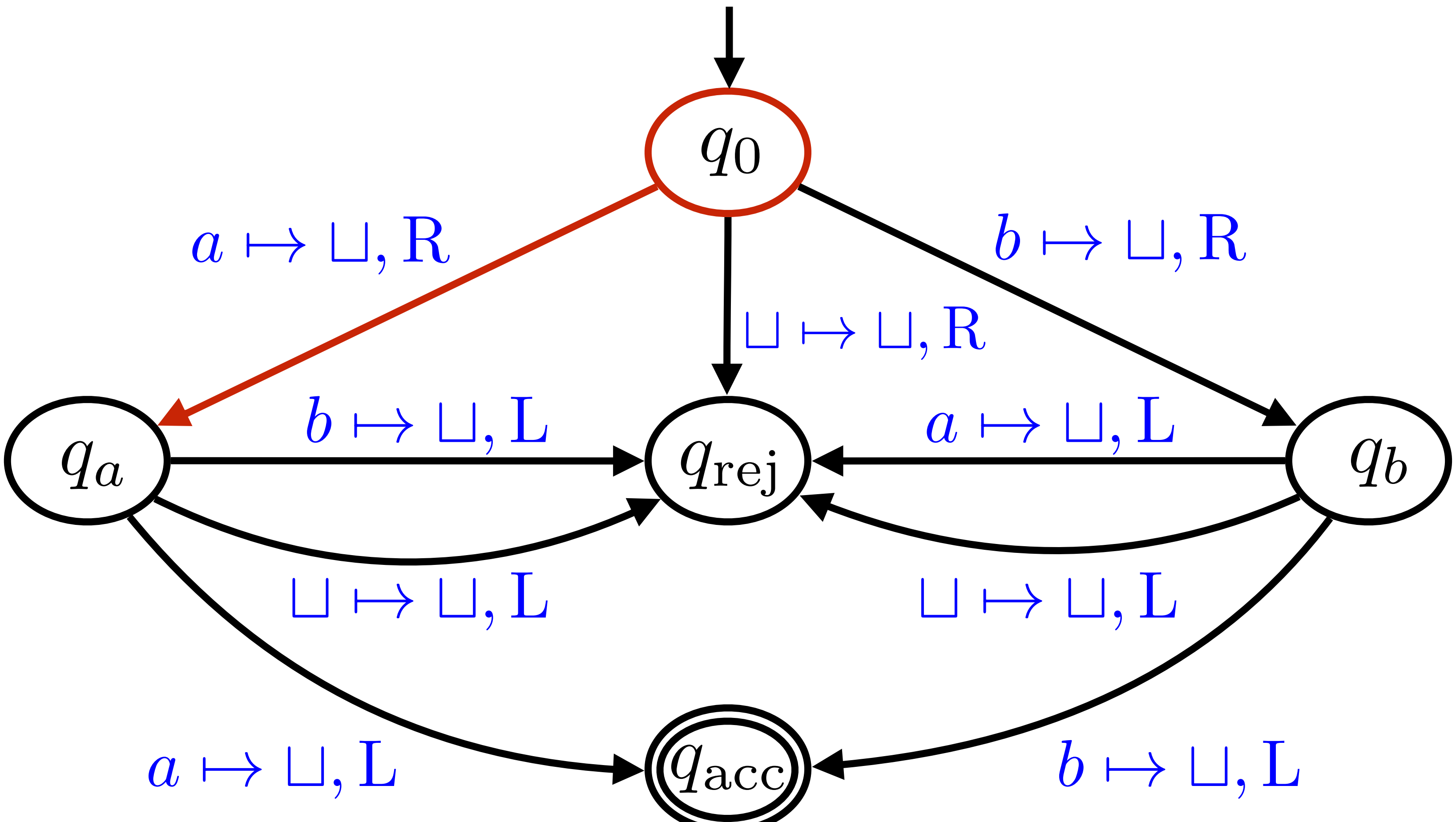
Input: aaba



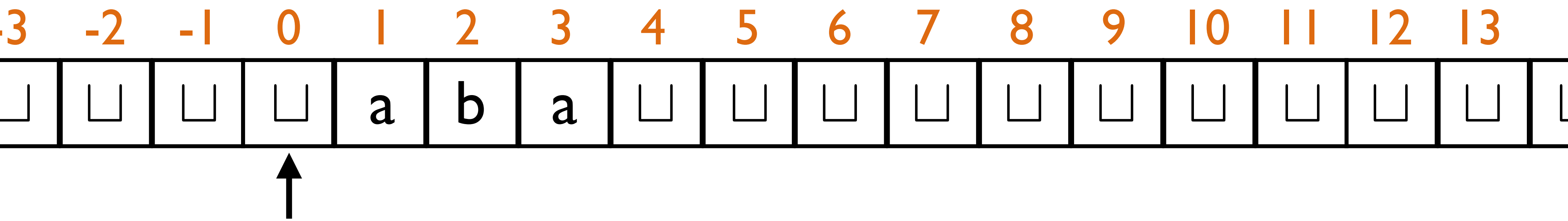
Turing machine simulation example



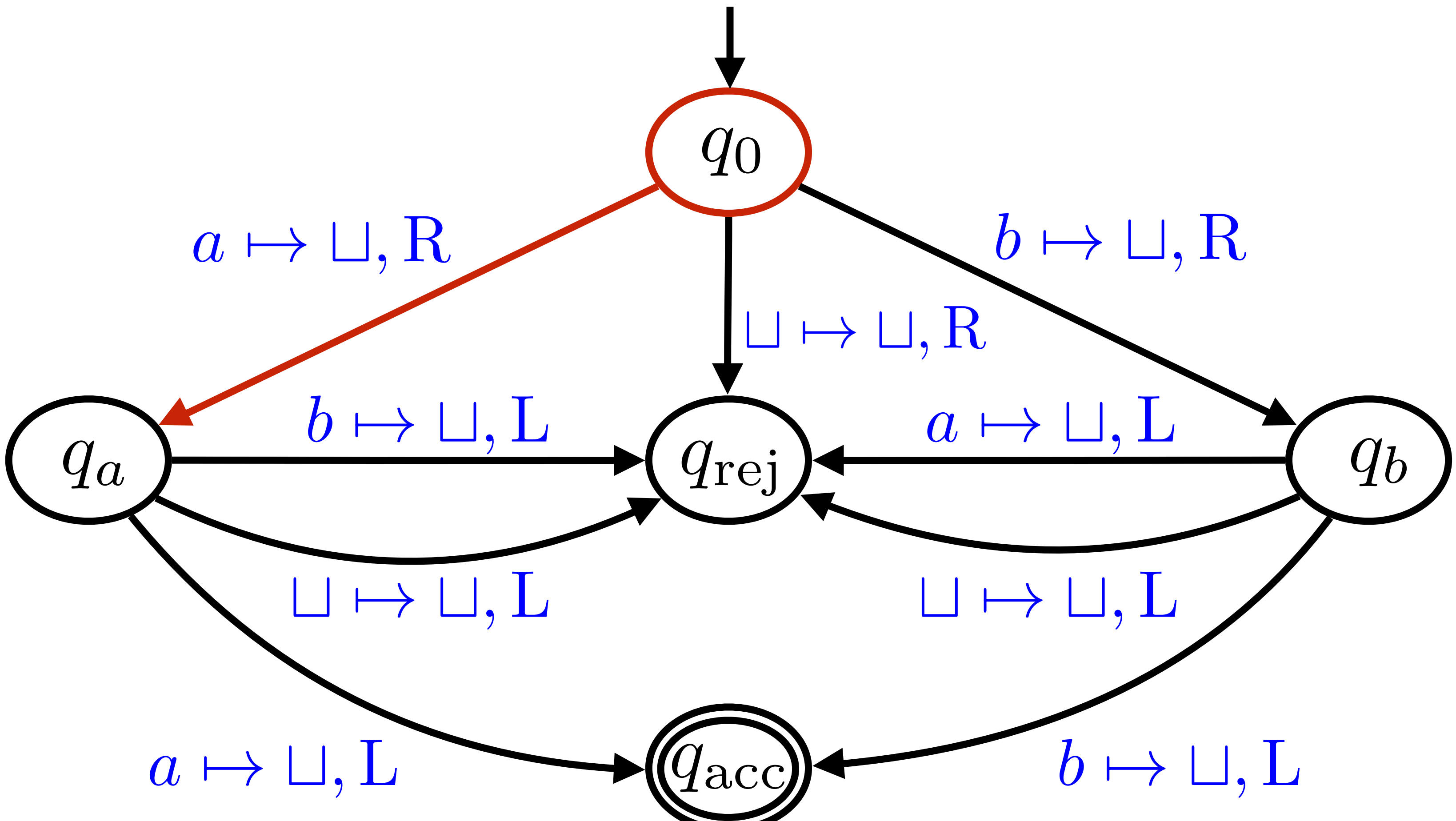
Input: aaba



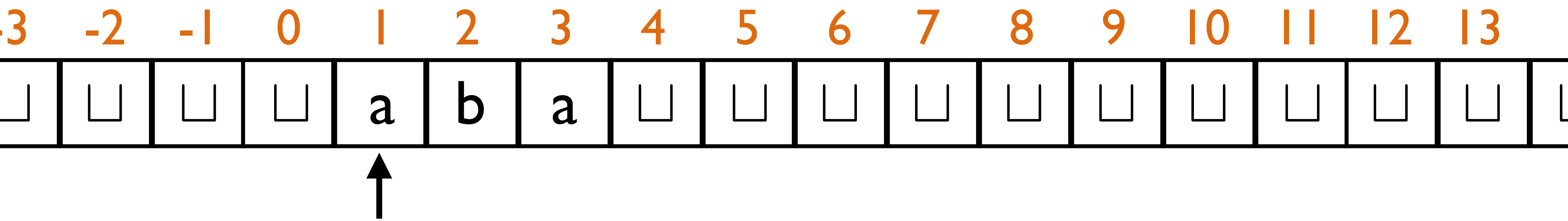
Turing machine simulation example



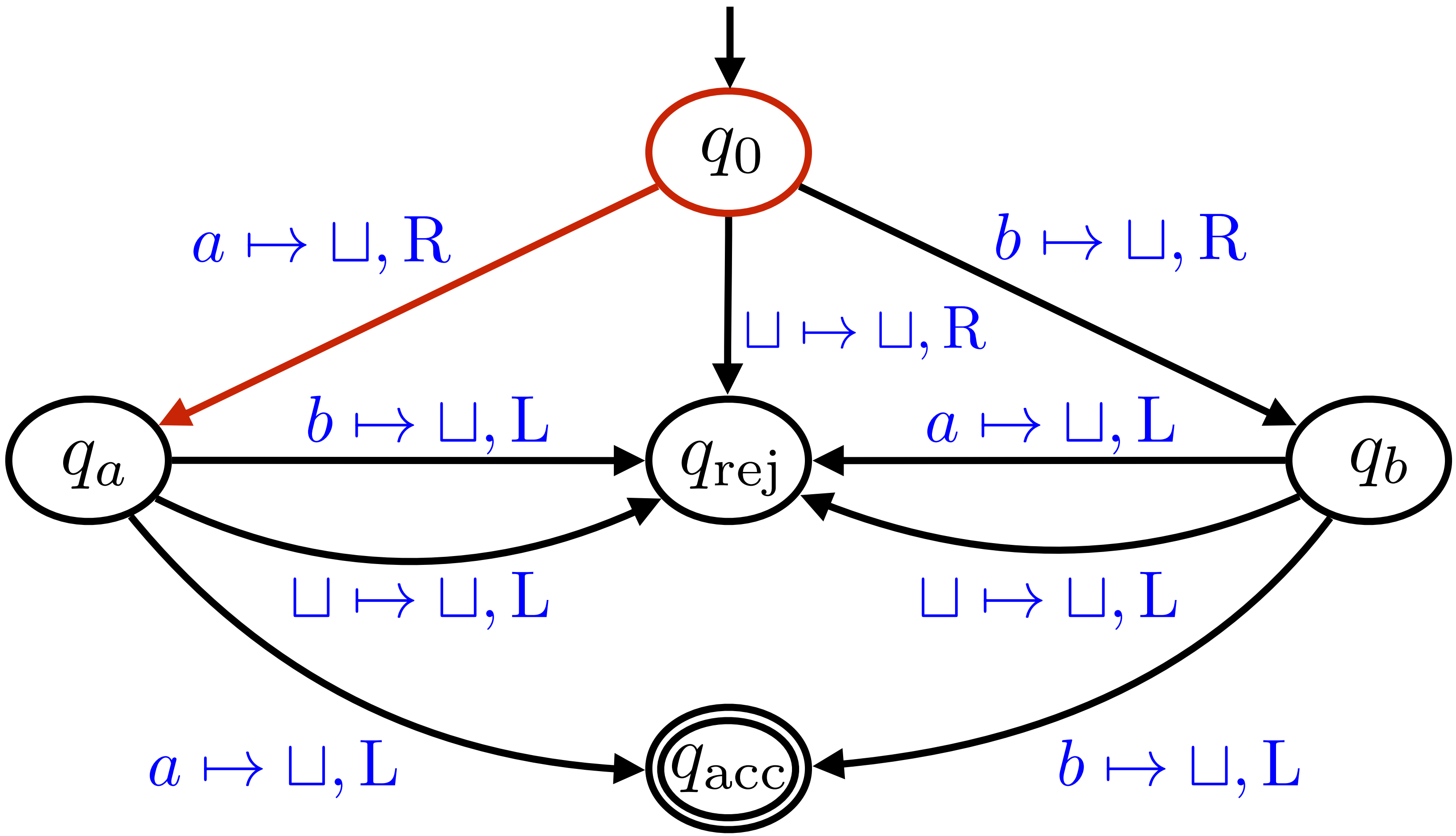
Input: aaba



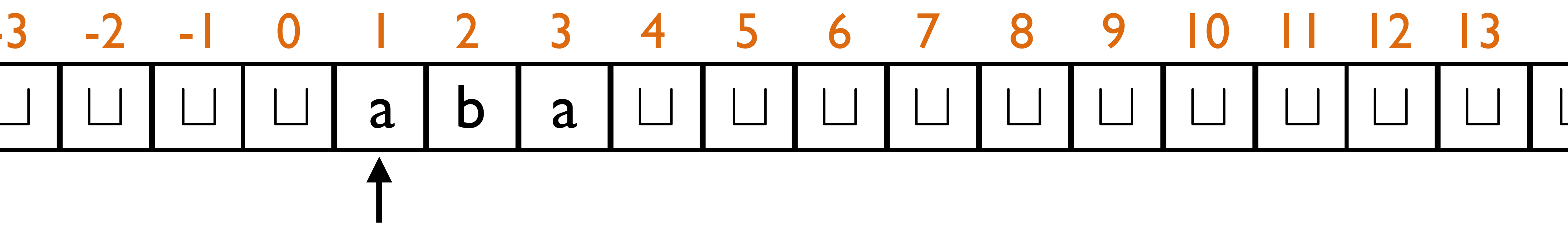
Turing machine simulation example



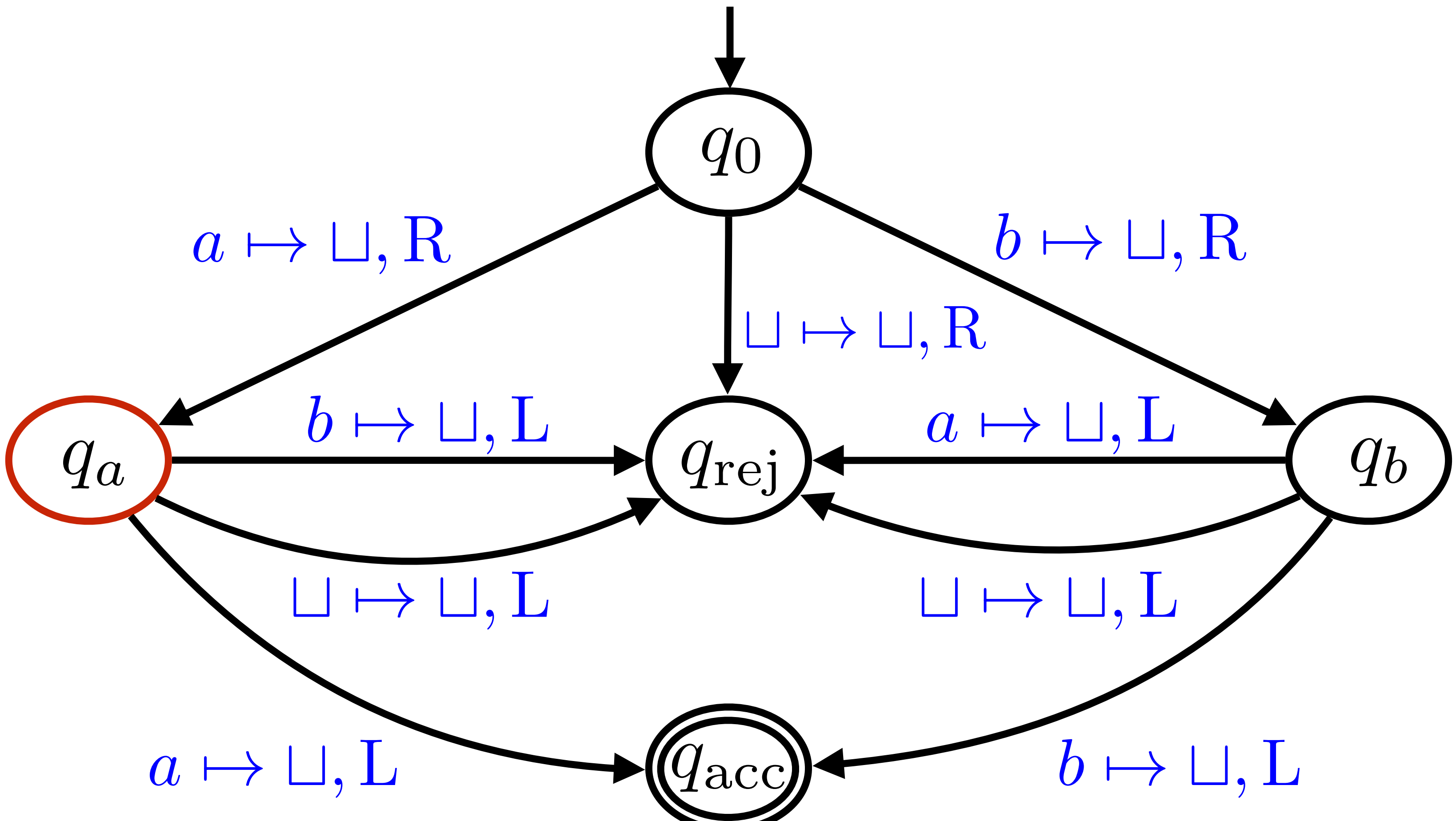
Input: aaba



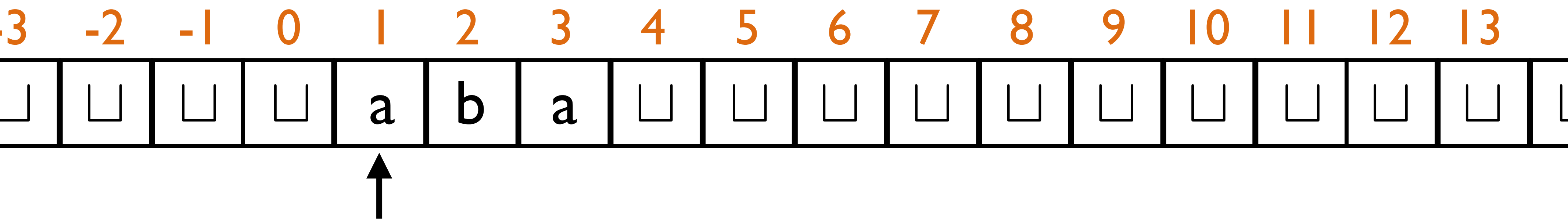
Turing machine simulation example



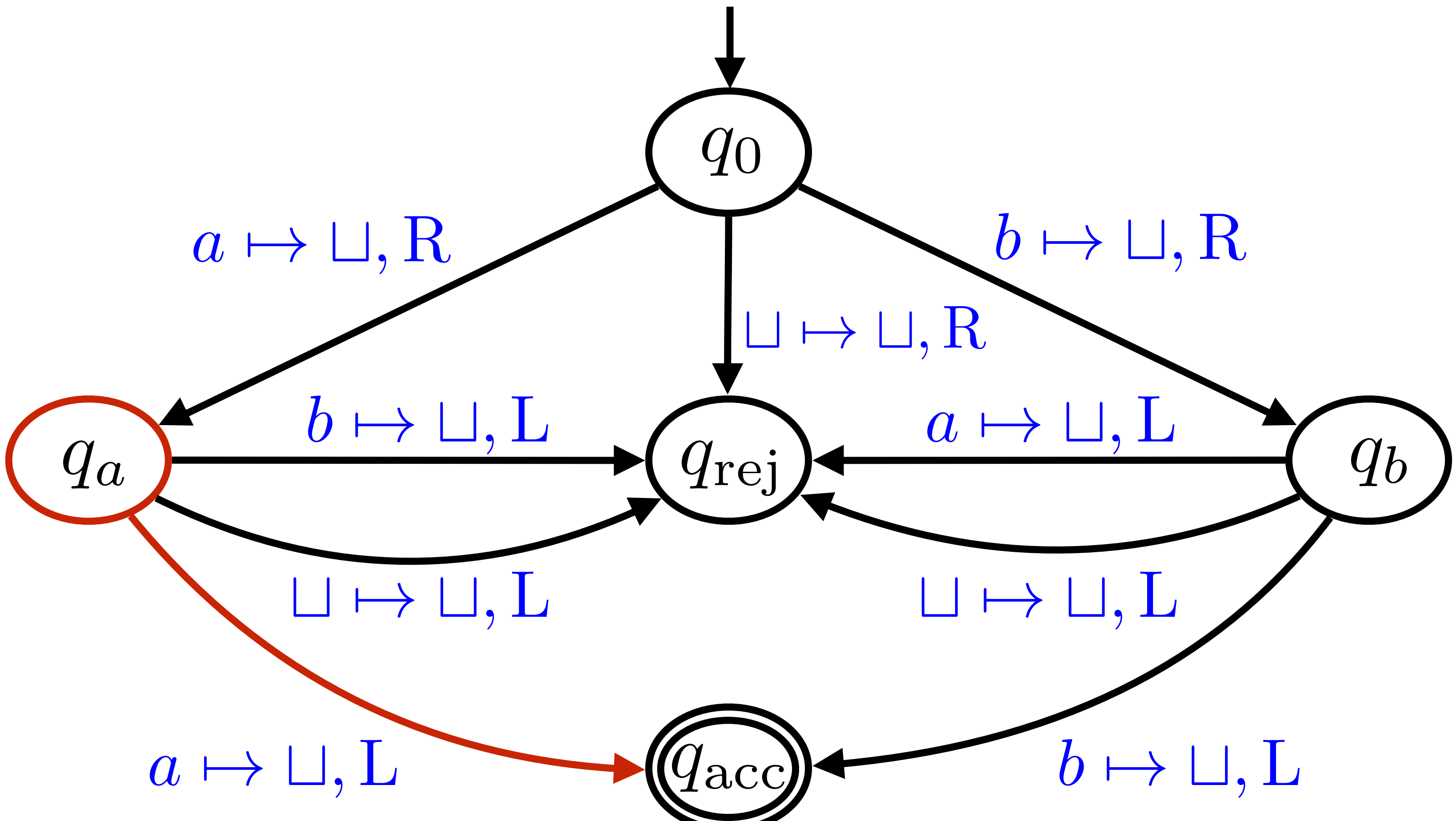
Input: aaba



Turing machine simulation example



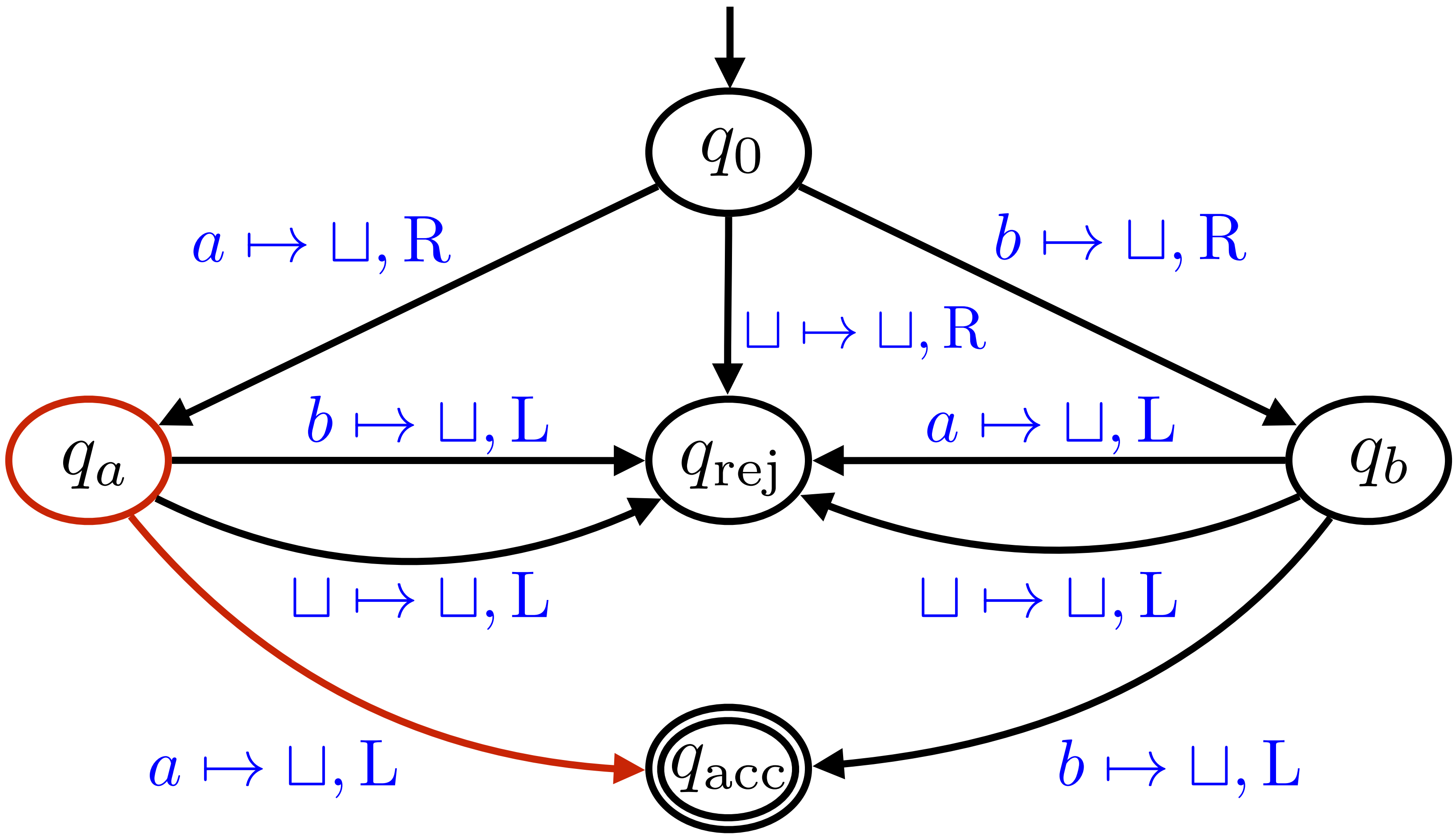
Input: aaba



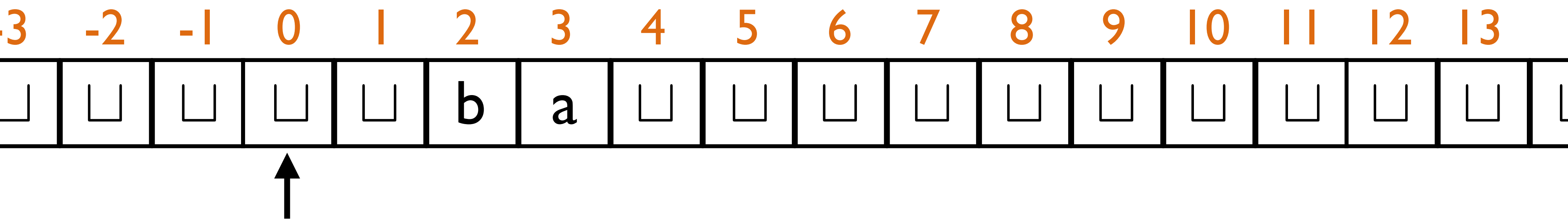
Turing machine simulation example

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

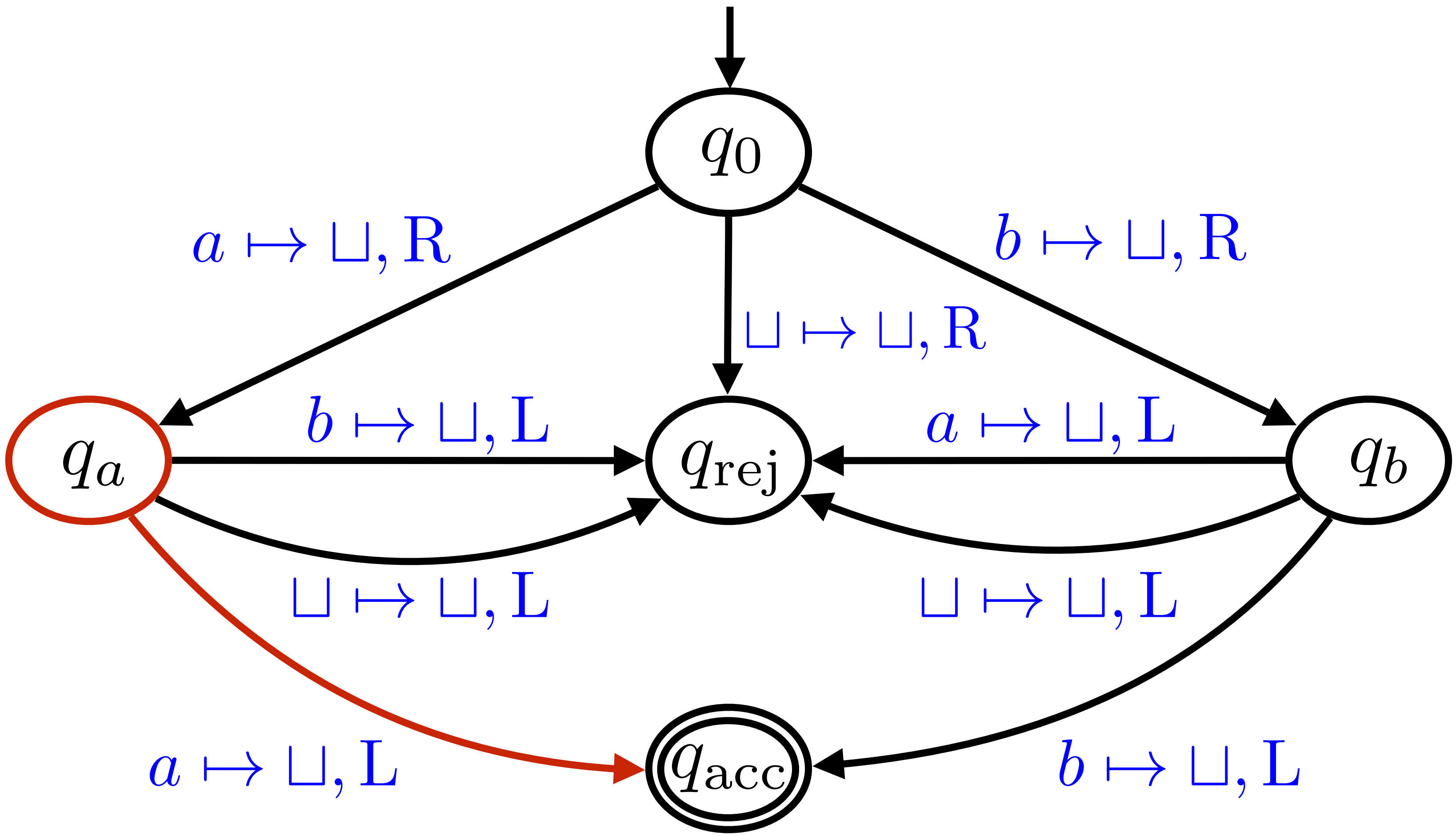
Input: aaba



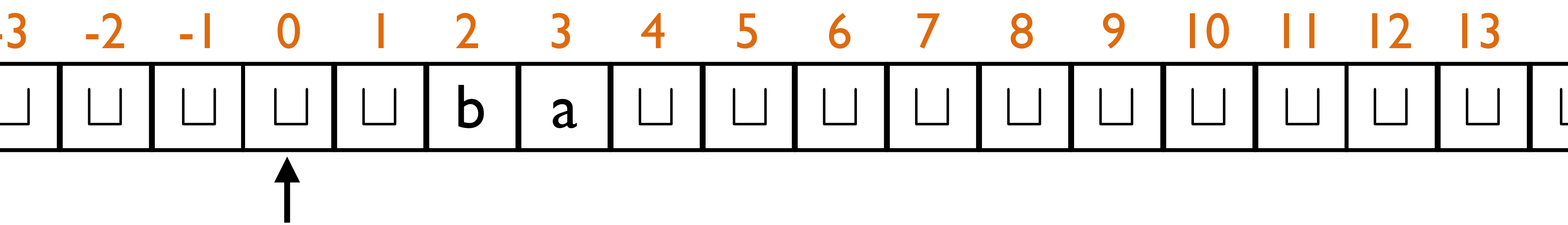
Turing machine simulation example



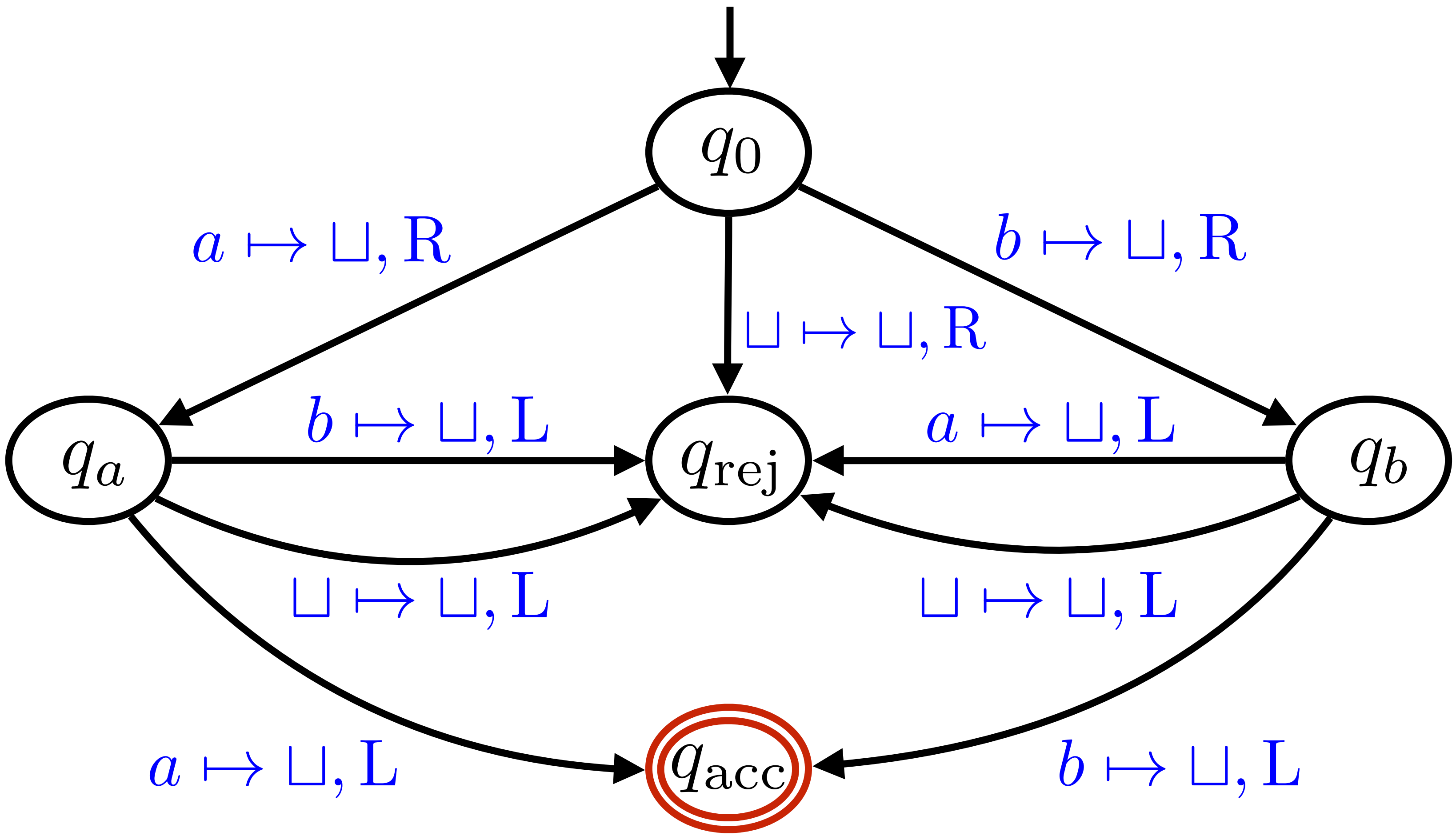
Input: aaba



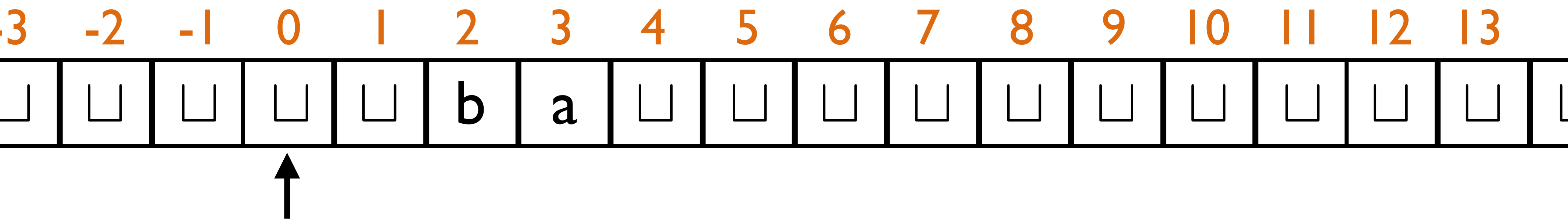
Turing machine simulation example



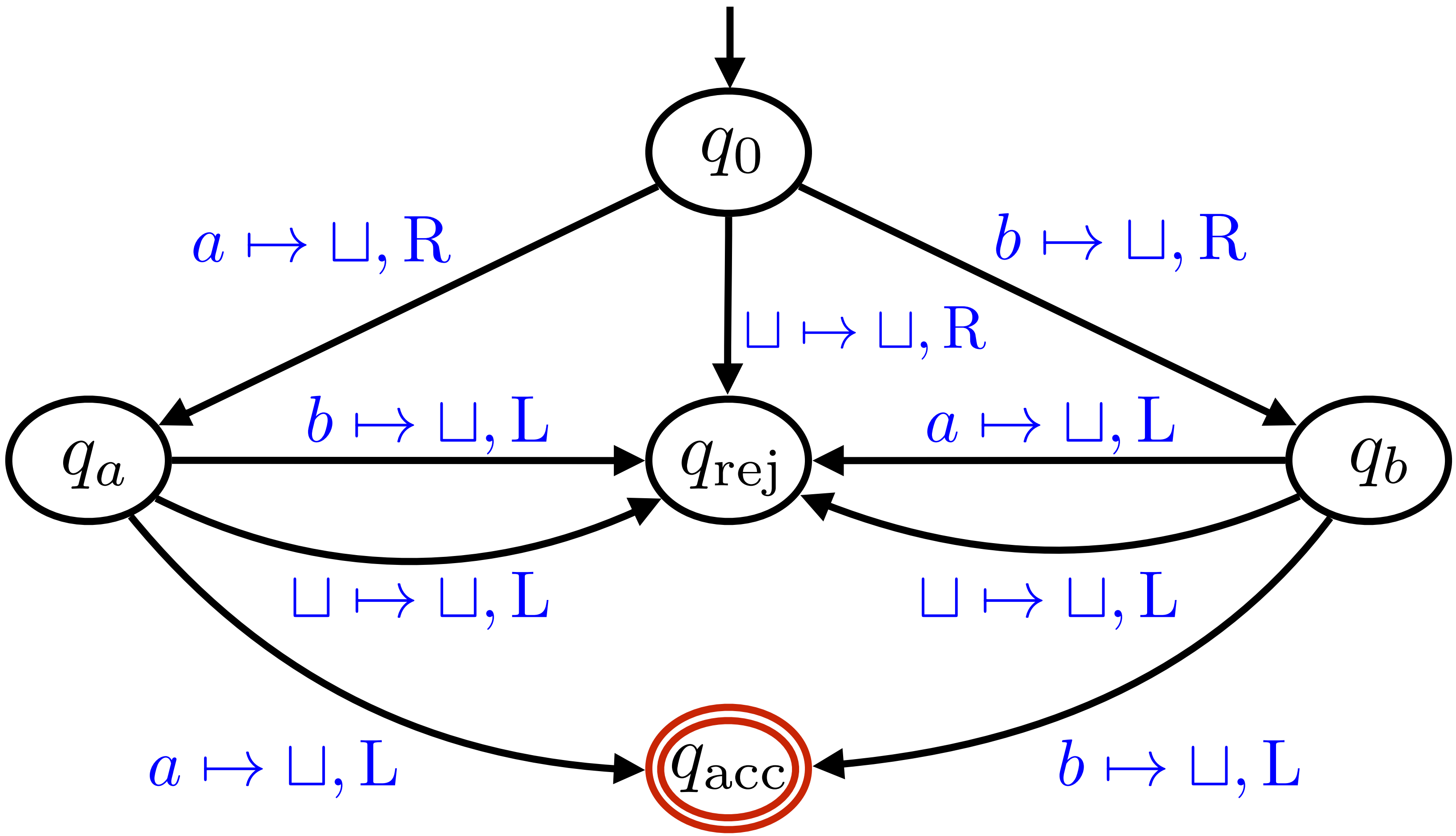
Input: aaba



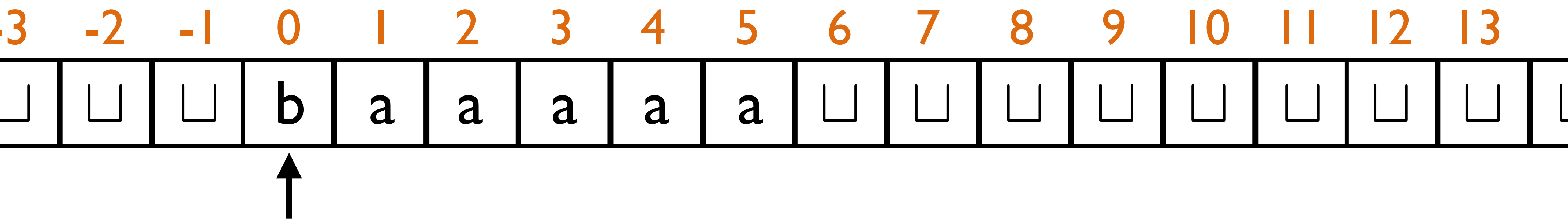
Turing machine simulation example



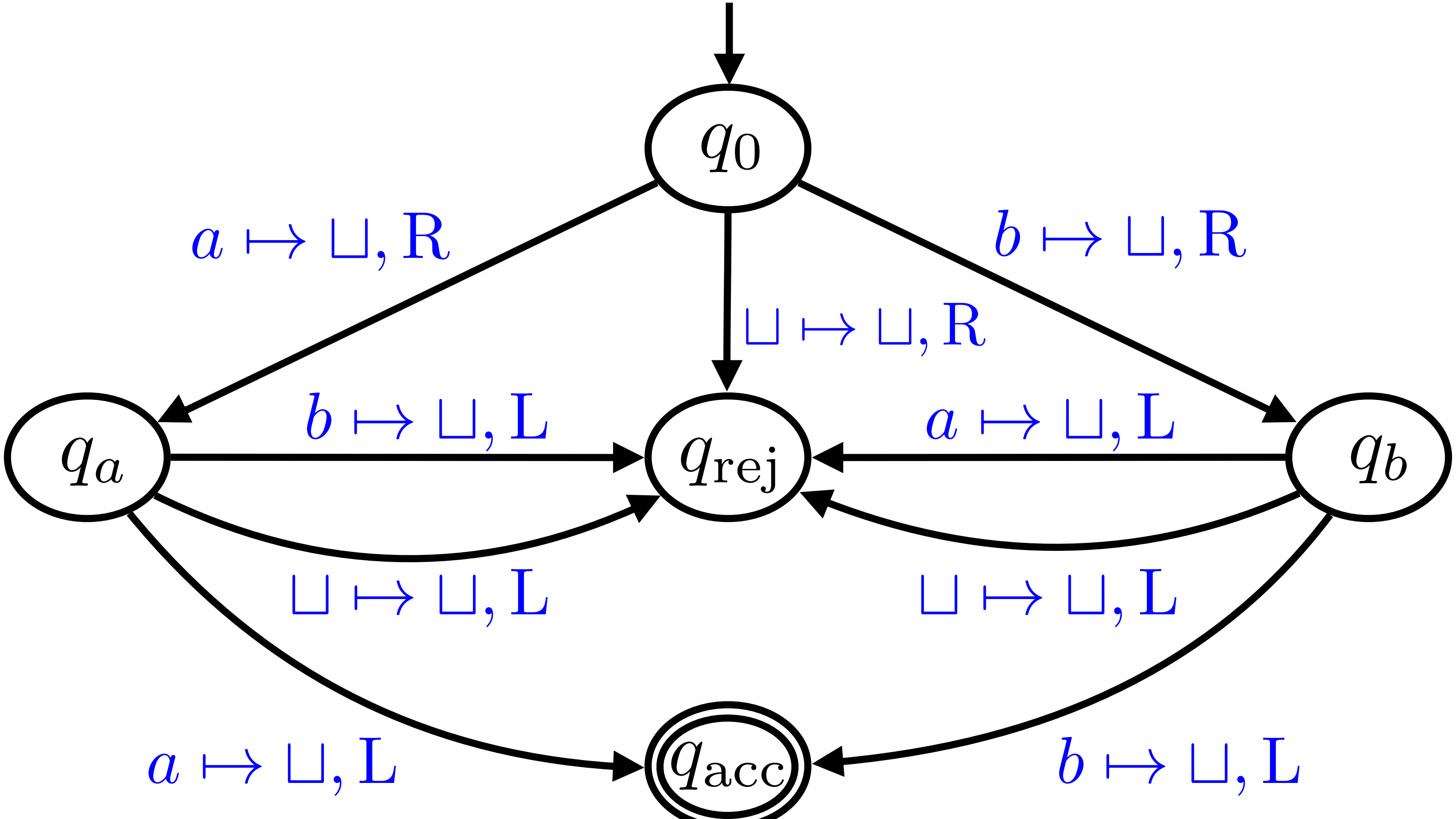
Input: aaba **Decision:** Accept



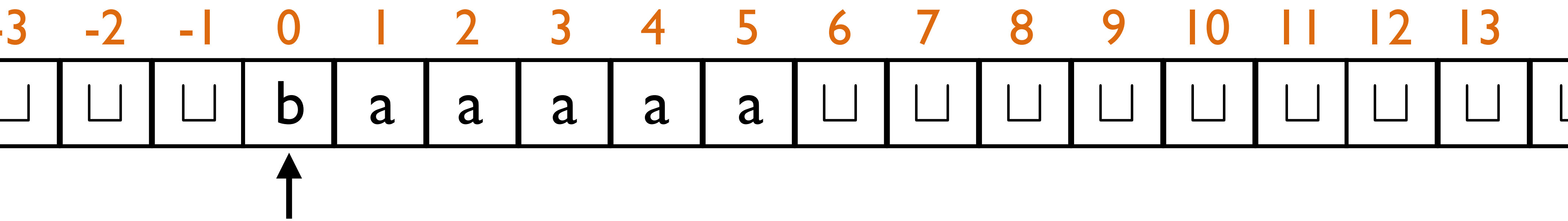
Turing machine simulation example



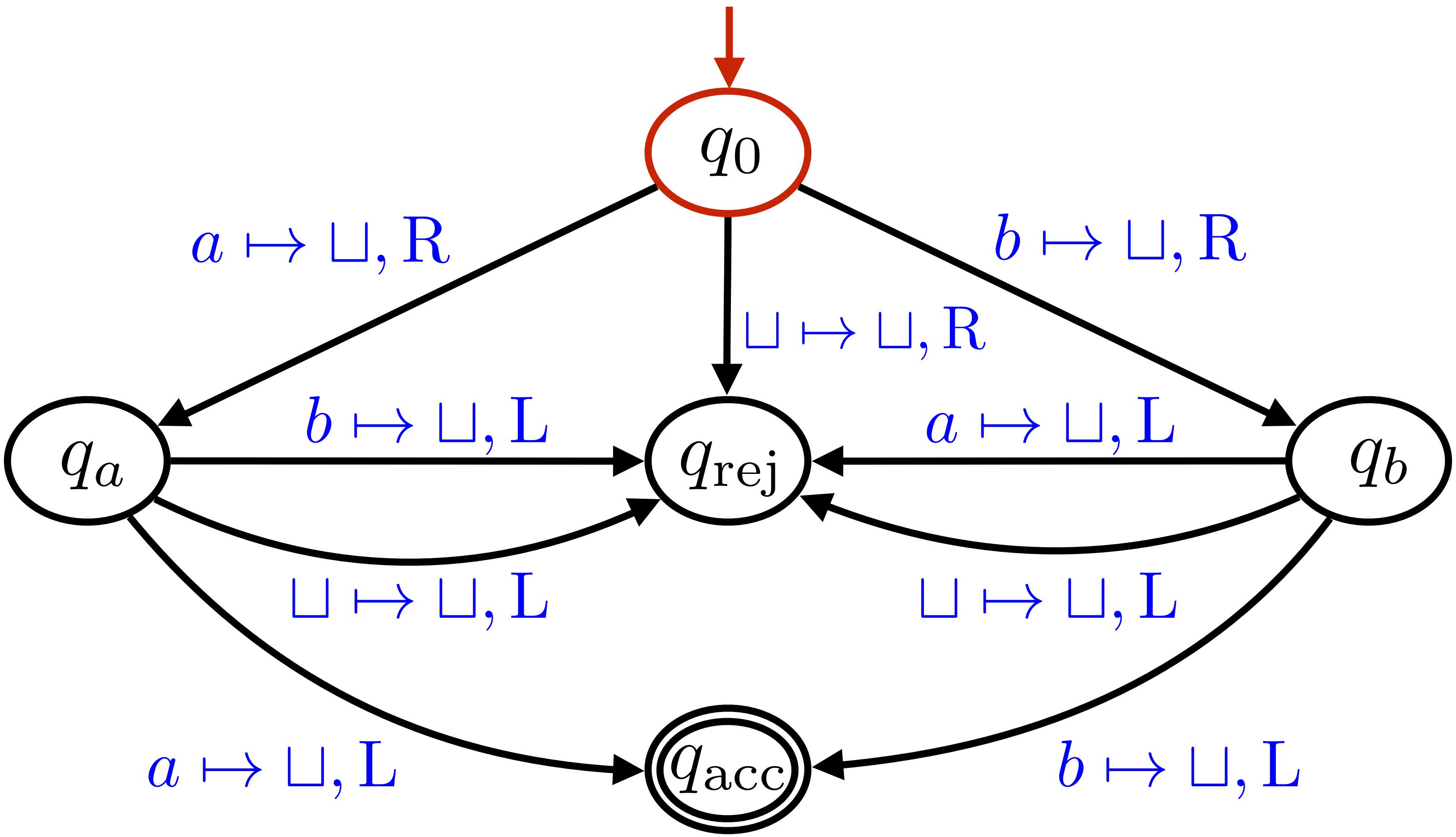
Input: baaaaa



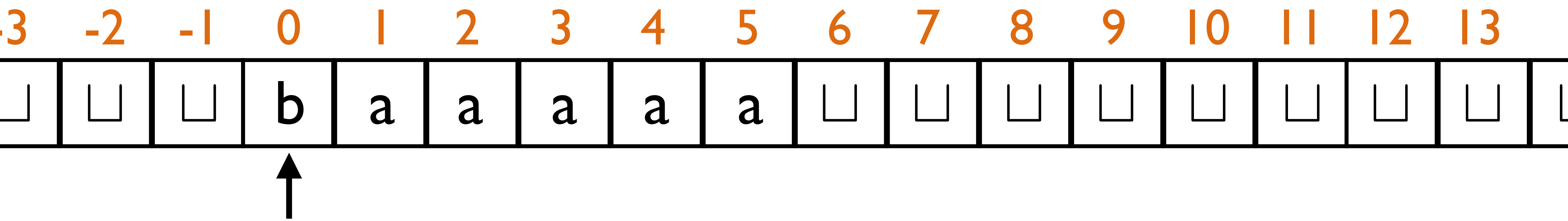
Turing machine simulation example



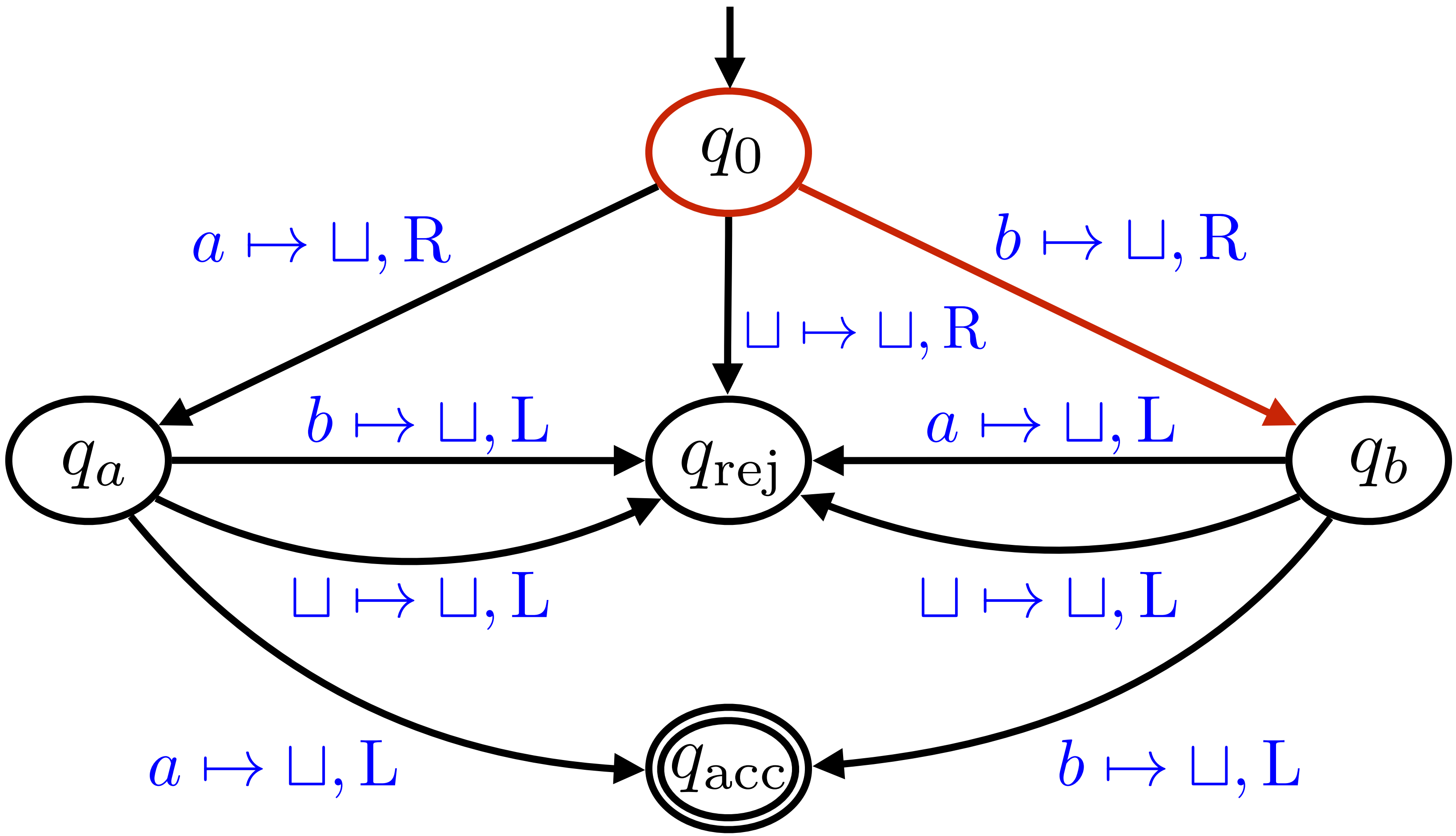
Input: baaaaa



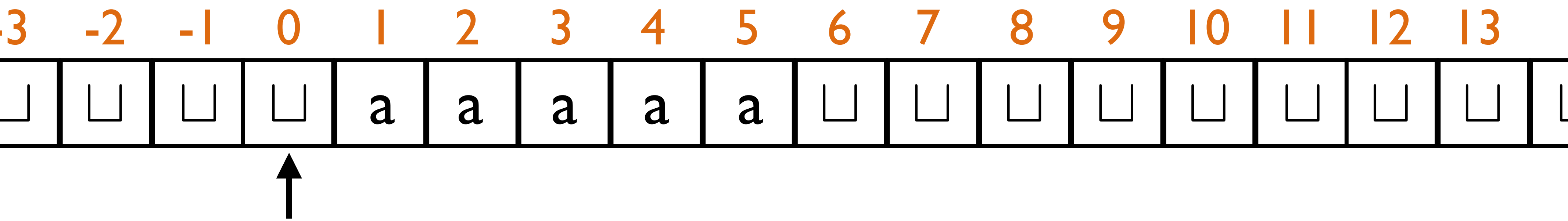
Turing machine simulation example



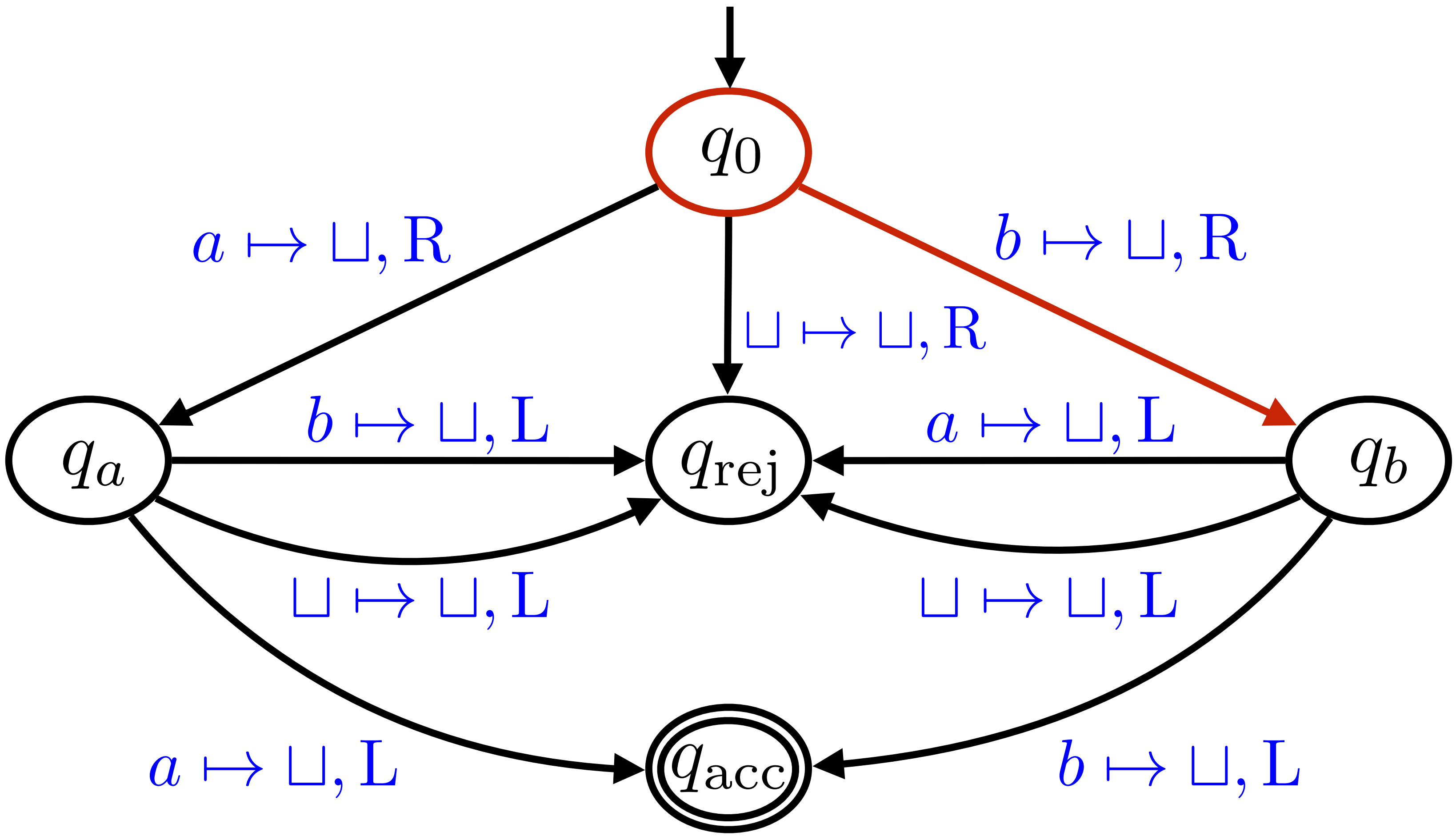
Input: baaaaa



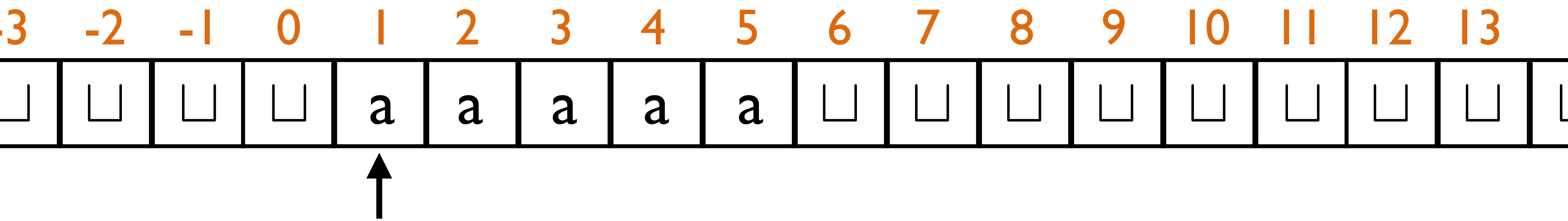
Turing machine simulation example



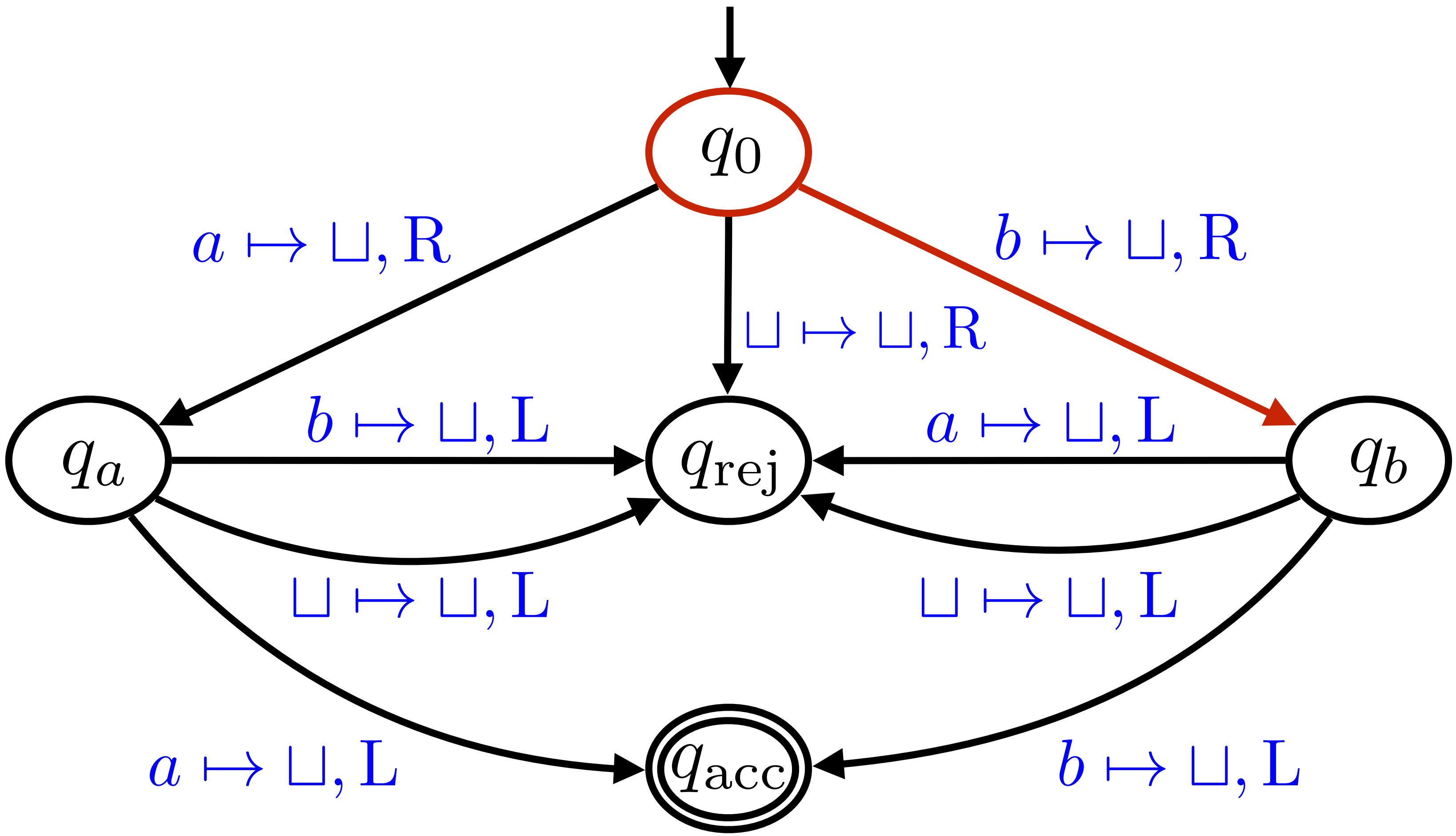
Input: baaaaa



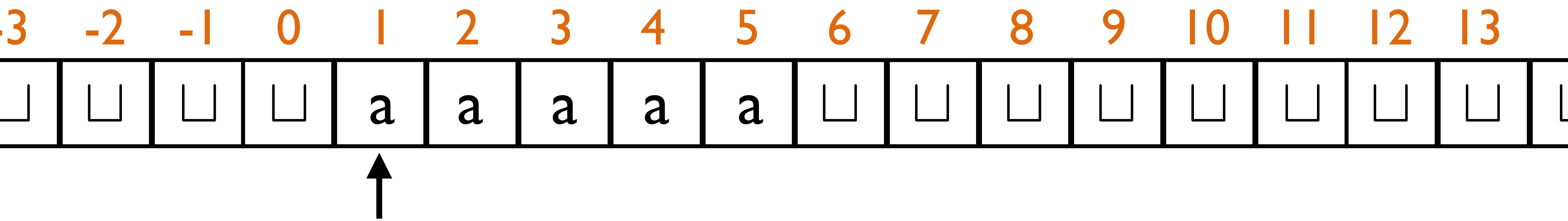
Turing machine simulation example



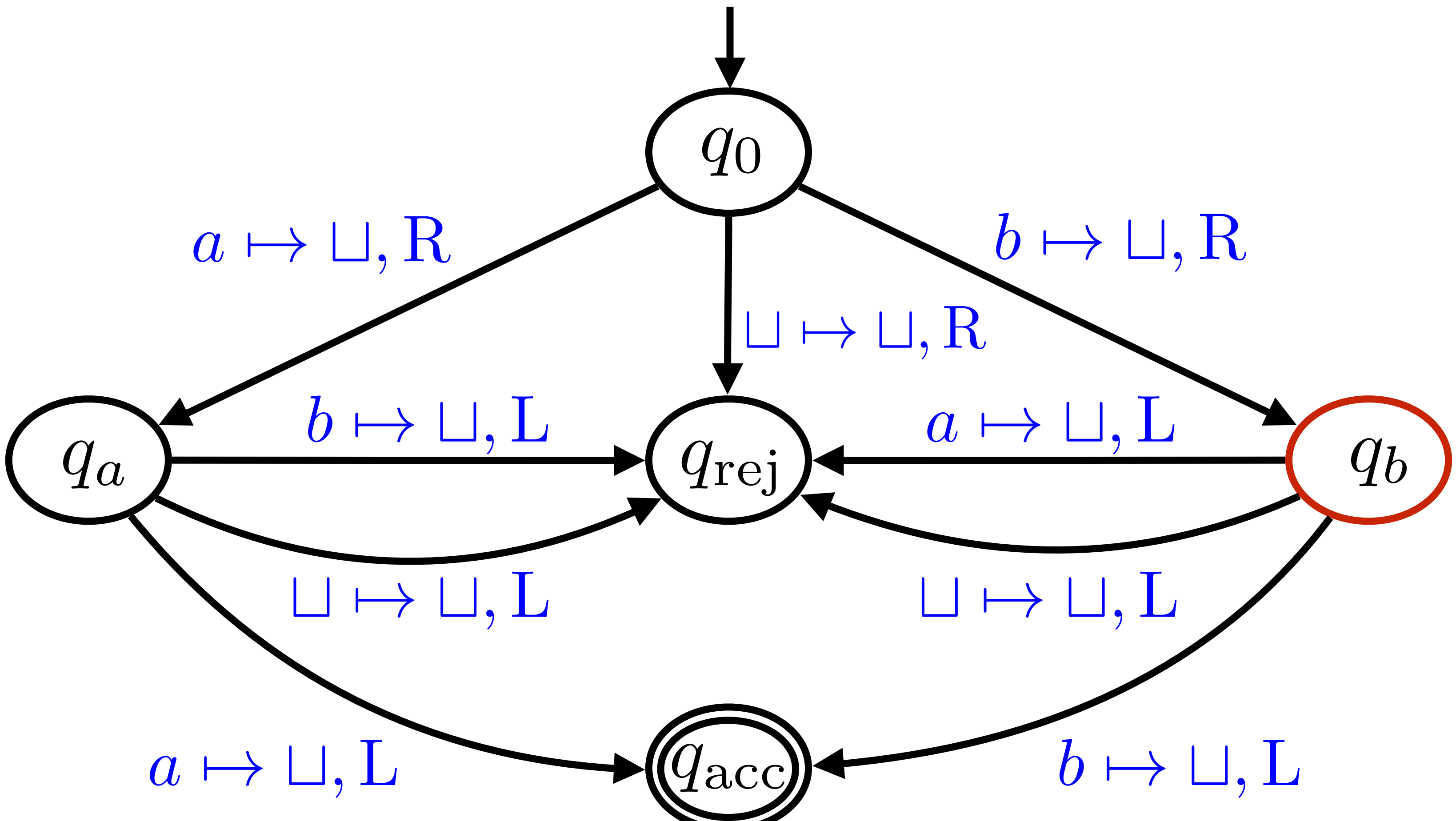
Input: baaaaa



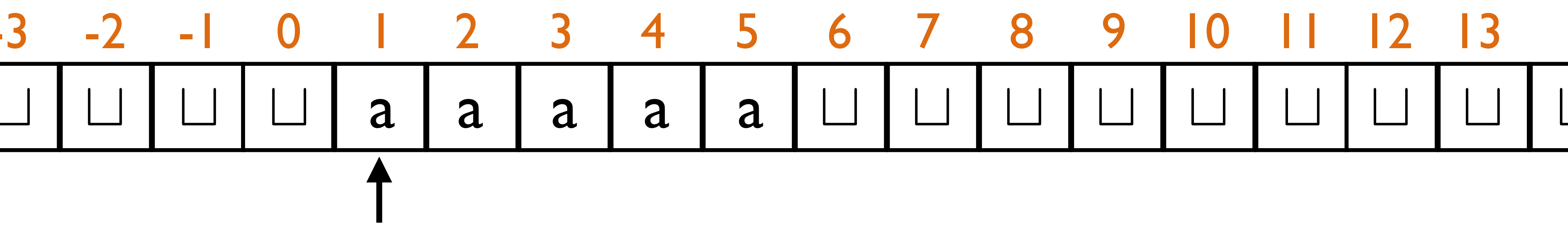
Turing machine simulation example



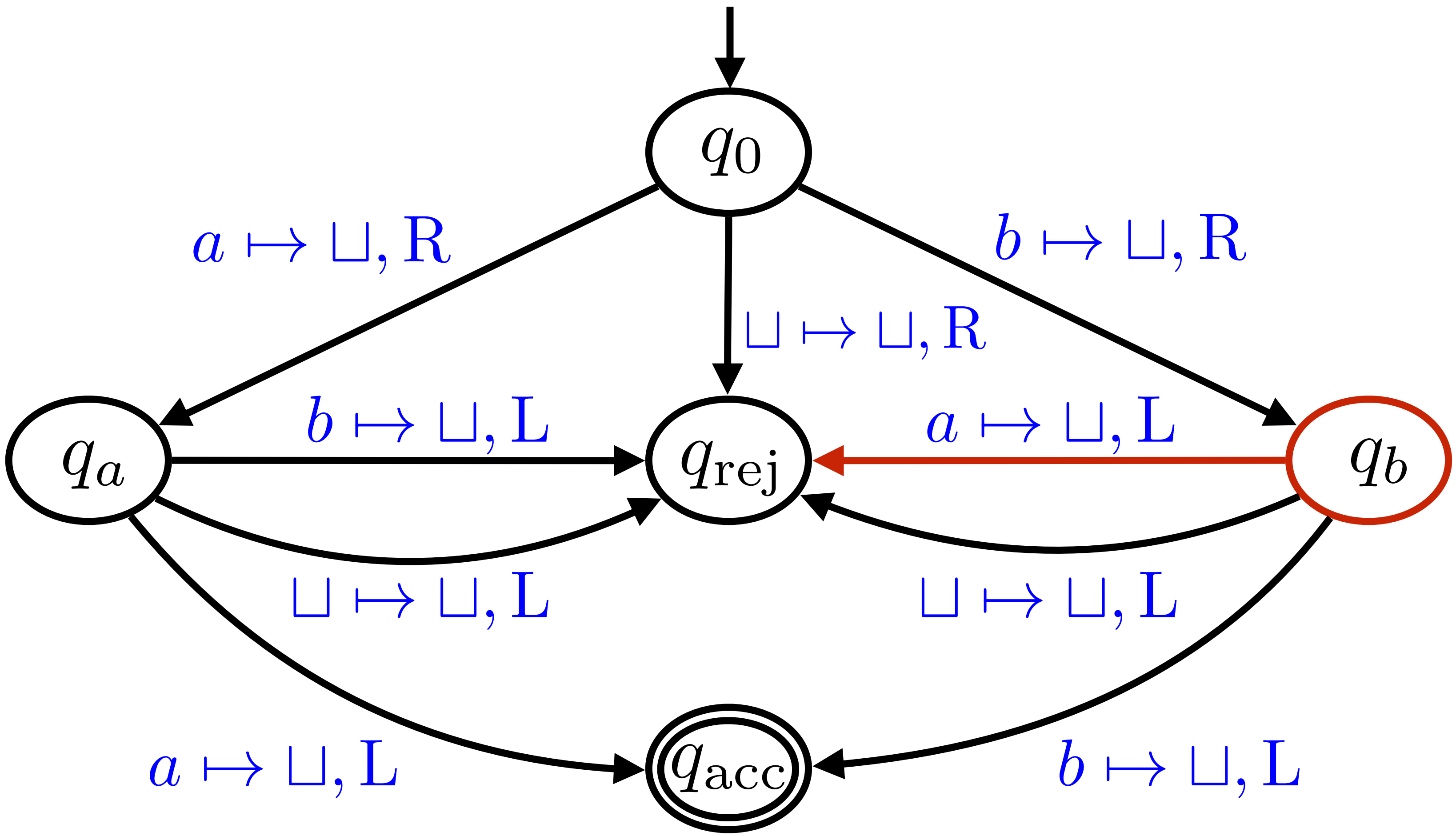
Input: baaaaa



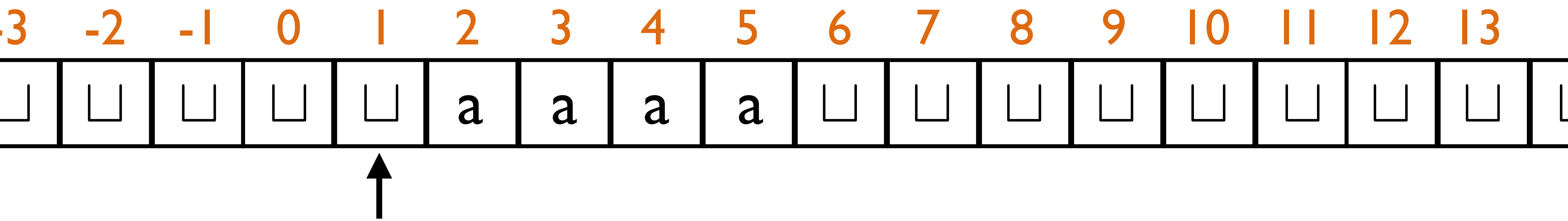
Turing machine simulation example



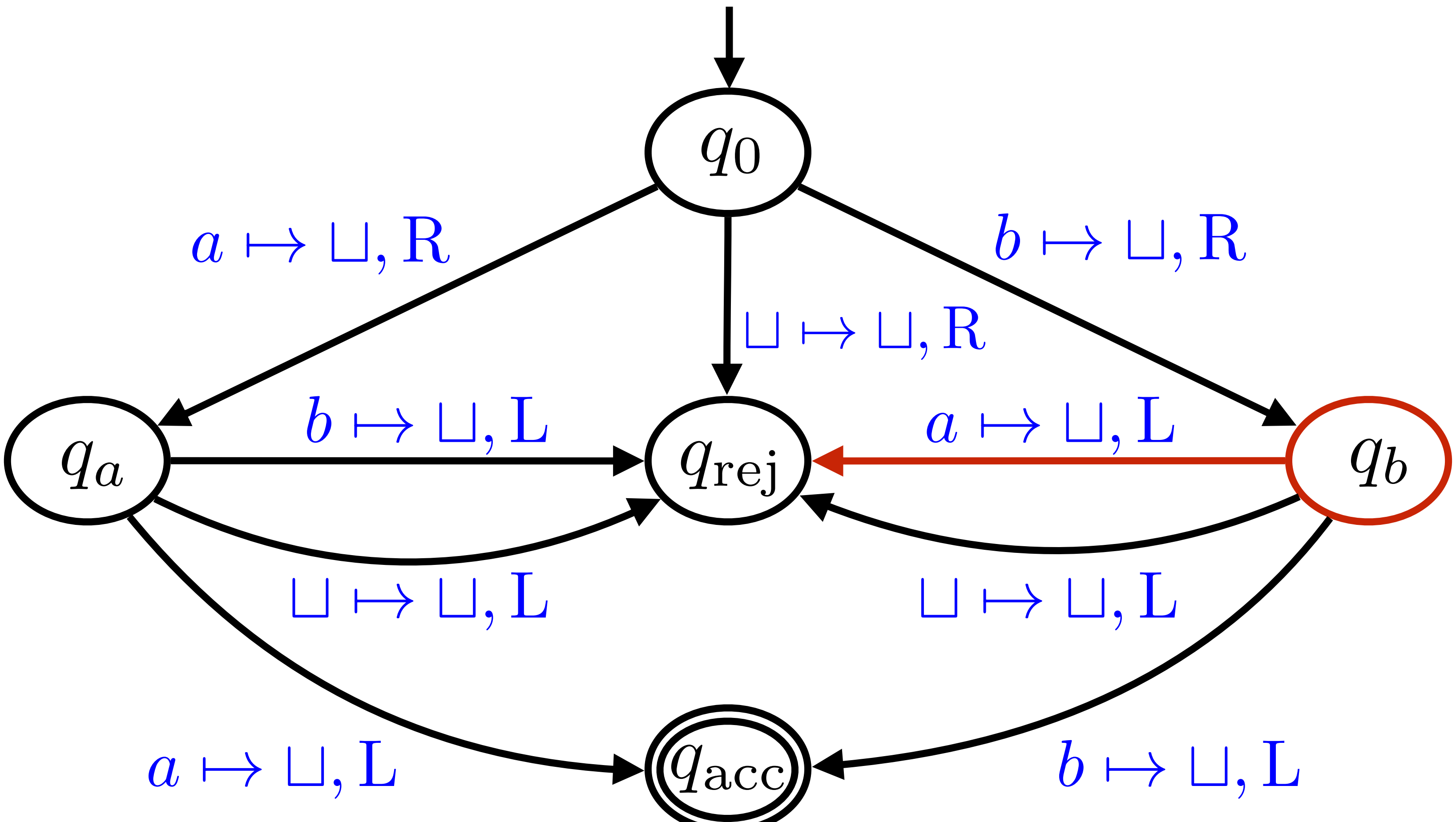
Input: baaaaa



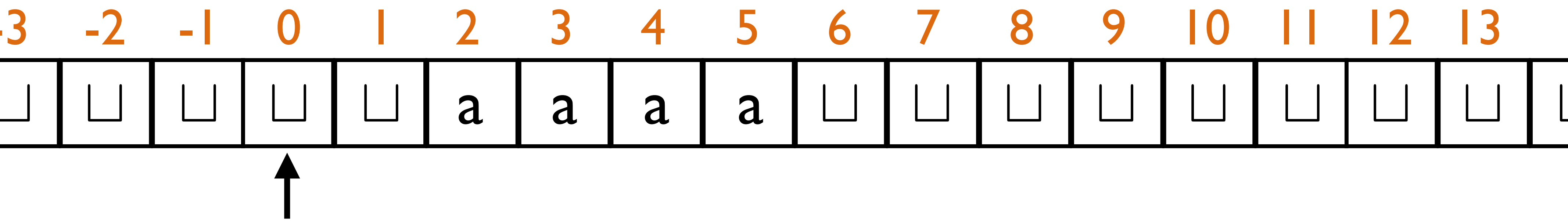
Turing machine simulation example



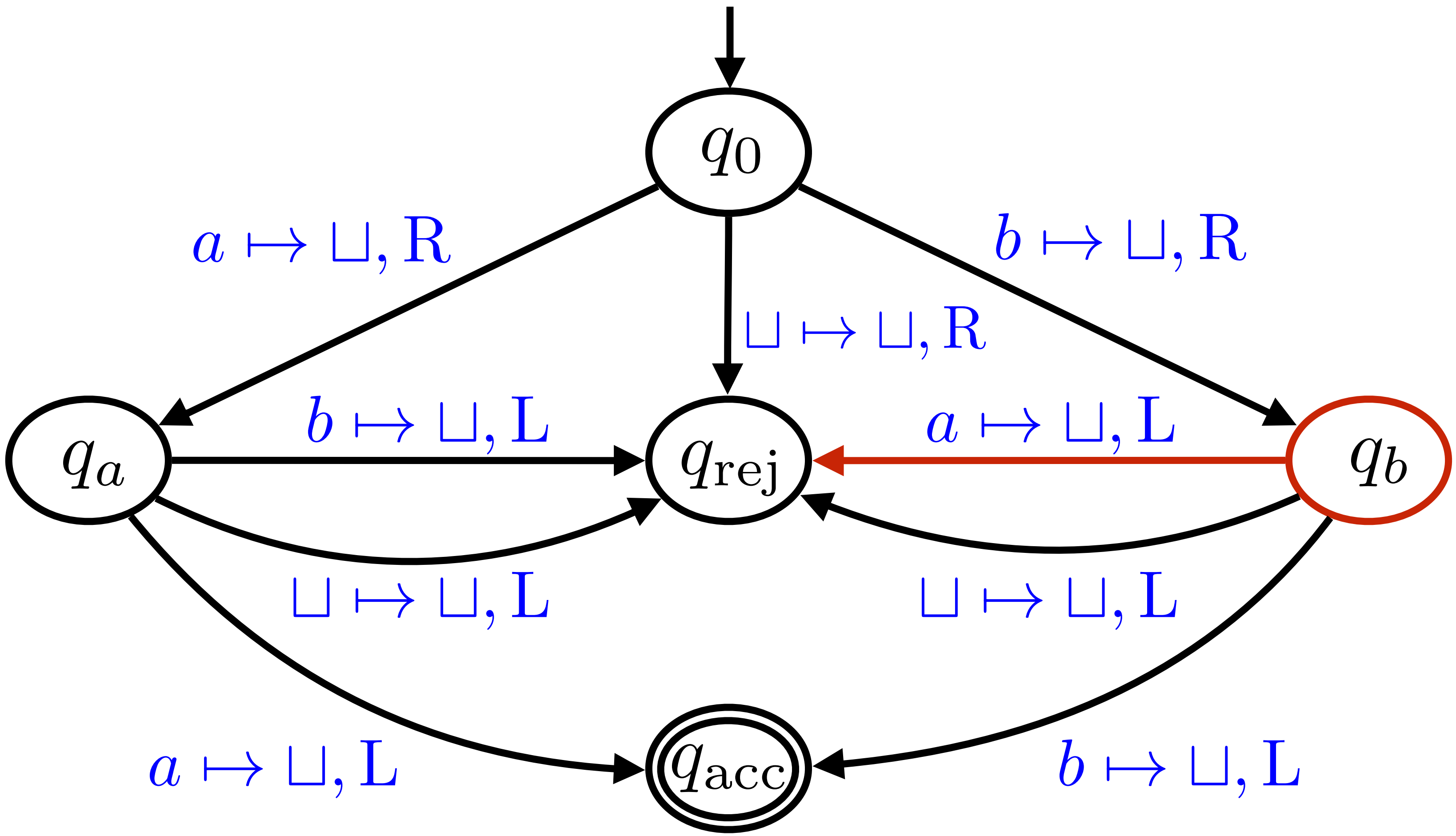
Input: baaaaa



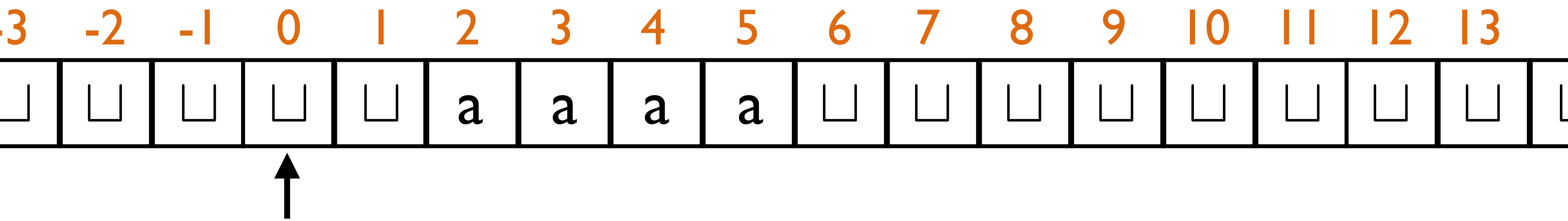
Turing machine simulation example



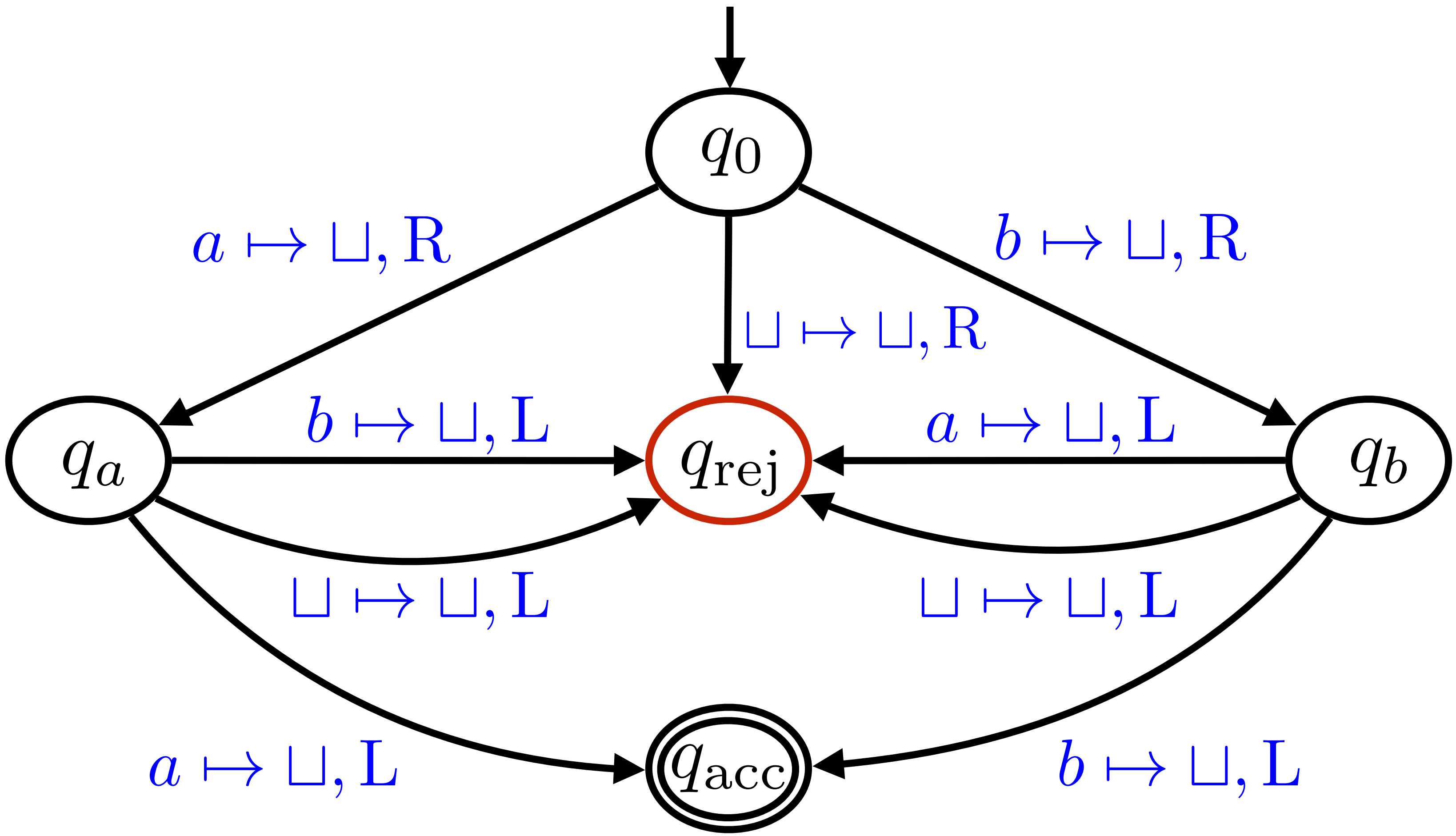
Input: baaaaa



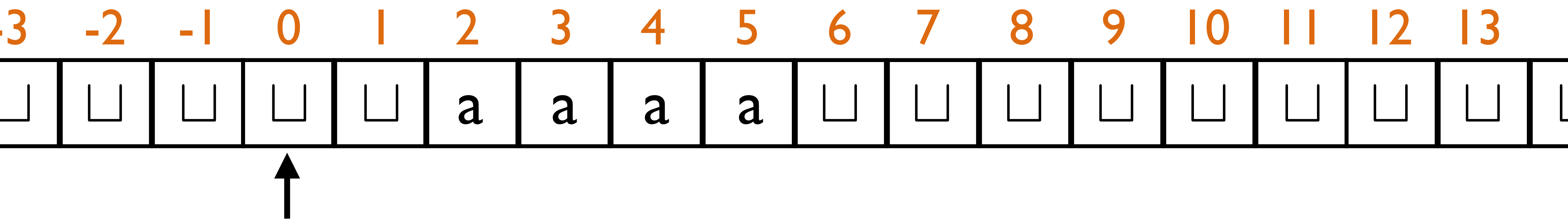
Turing machine simulation example



Input: baaaaa

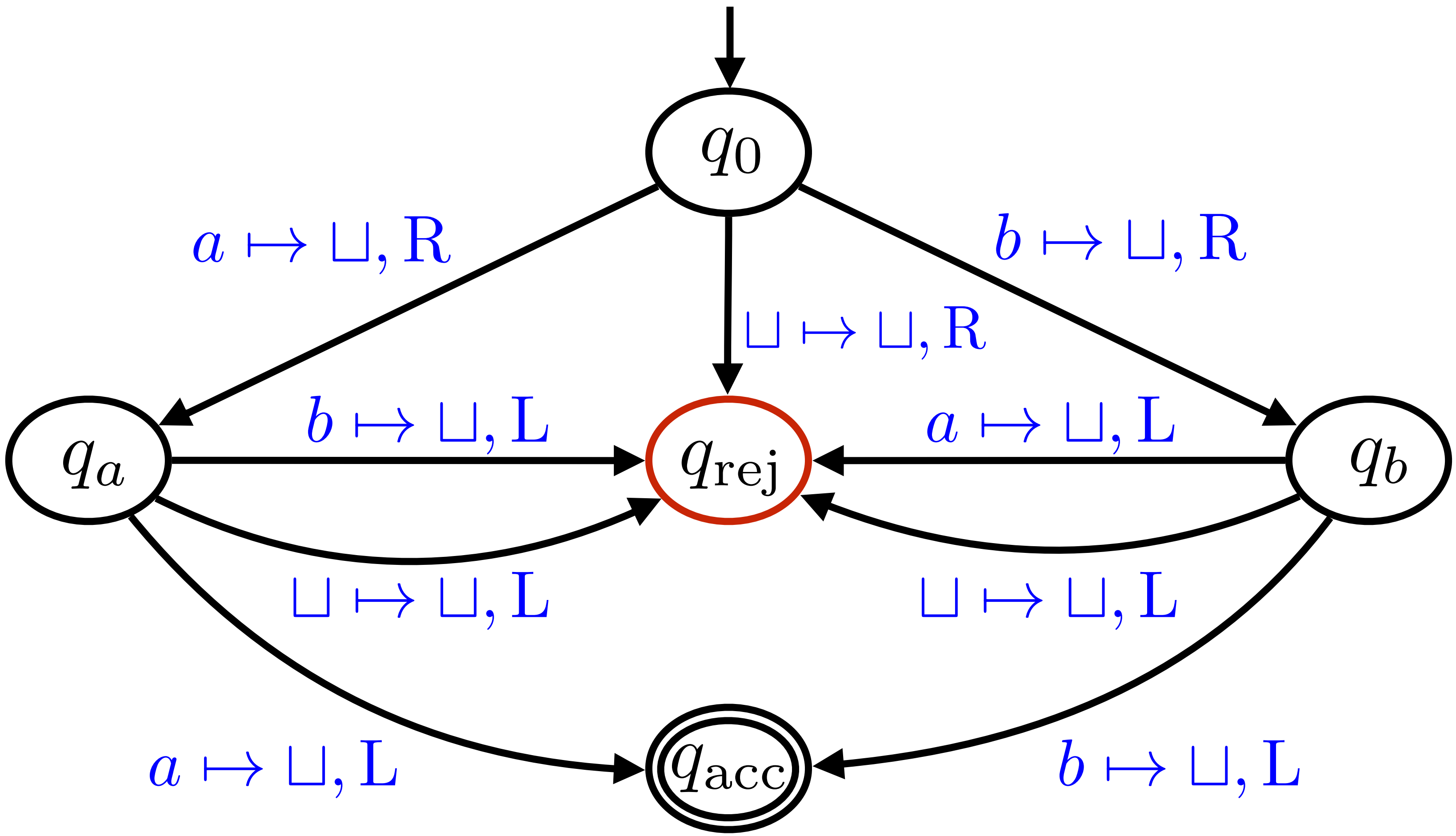


Turing machine simulation example

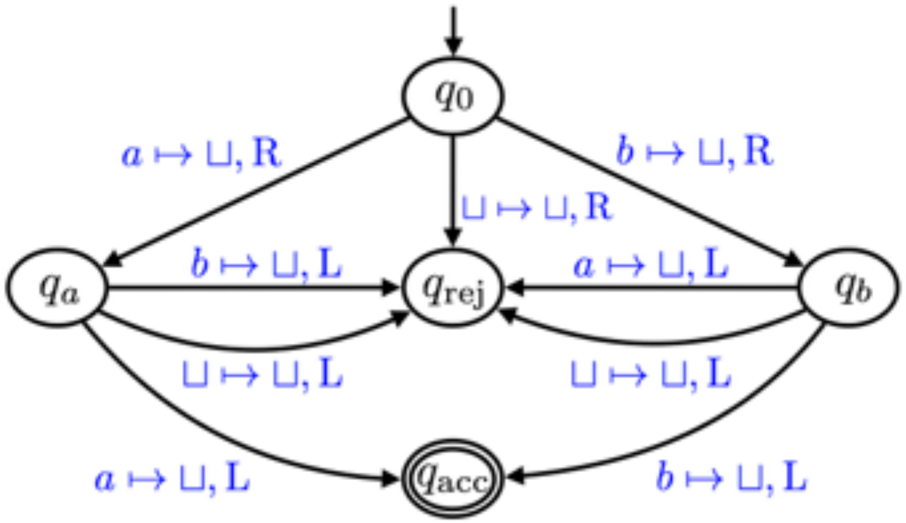


Input: baaaaa

Decision: Reject



TM as a programming language



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**;

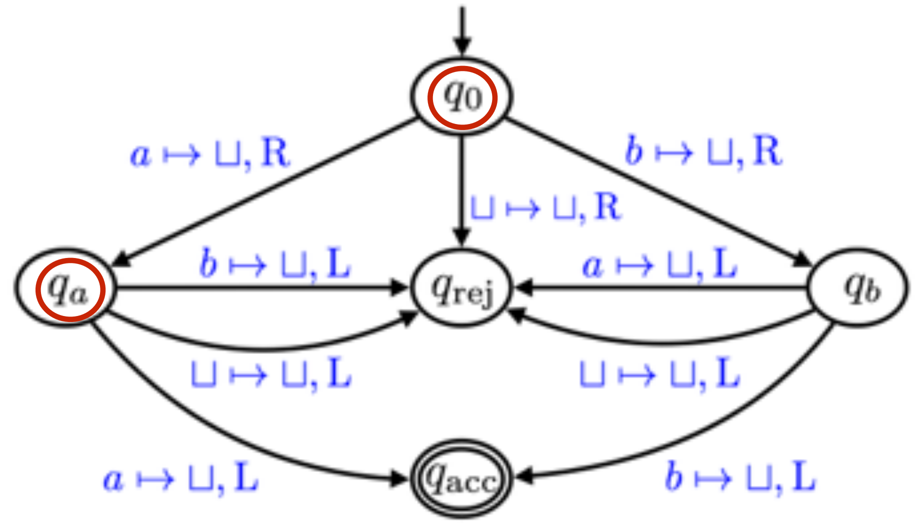
            case ‘b’: input[i] = ‘ ’; i--; go to STATE **rej**;

            case ‘ ’: input[i] = ‘ ’; i--; go to STATE **rej**;

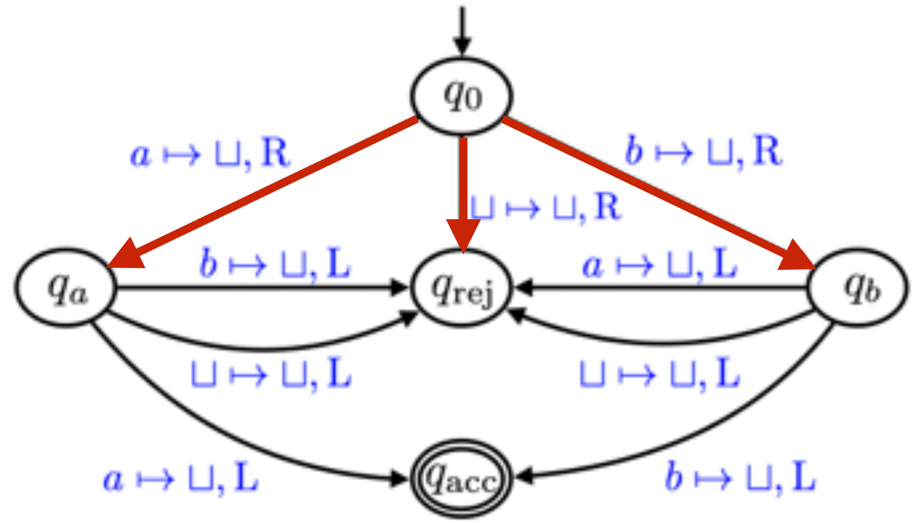
    ⋮

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;

    ⋮
```



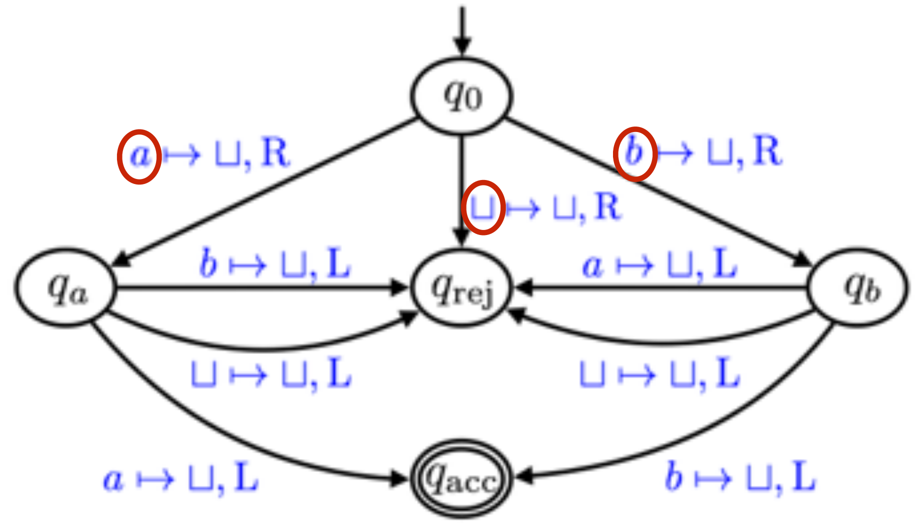
TM as a programming language



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**;

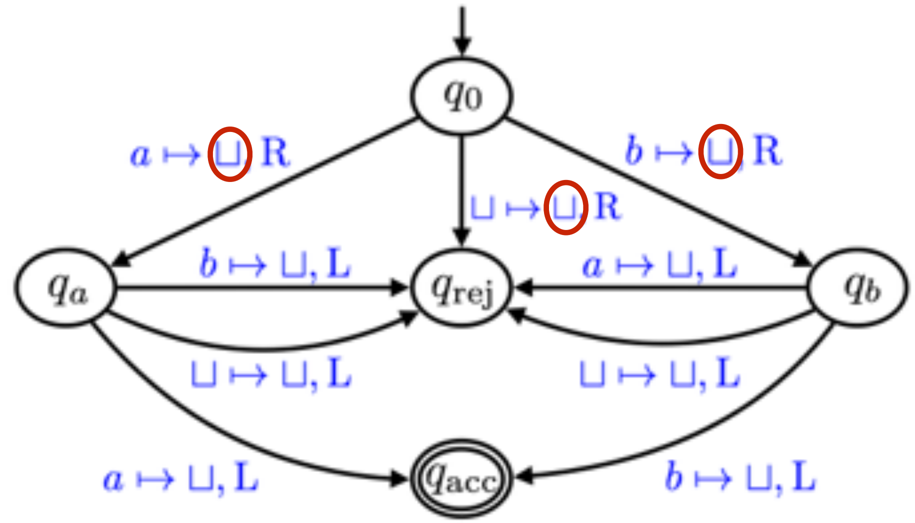
⋮

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;

    ⋮
```

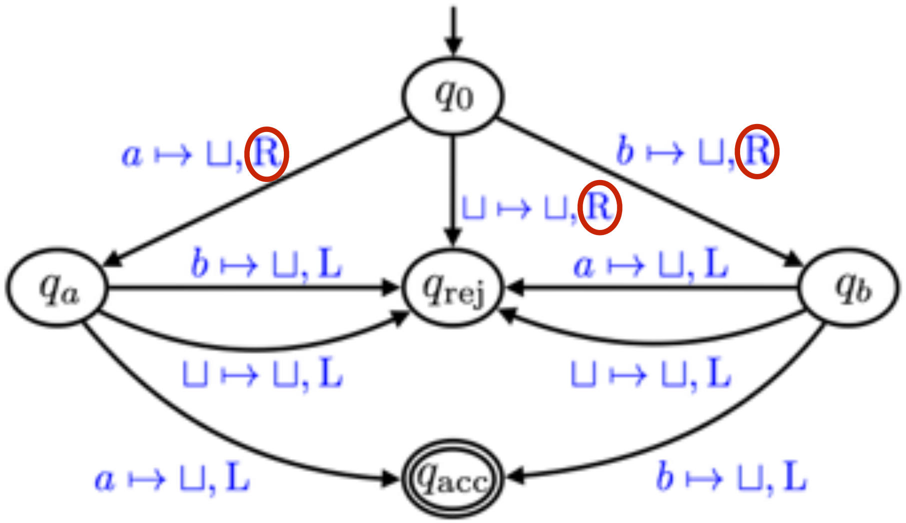
TM as a programming language




```
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;

   ⋮
```

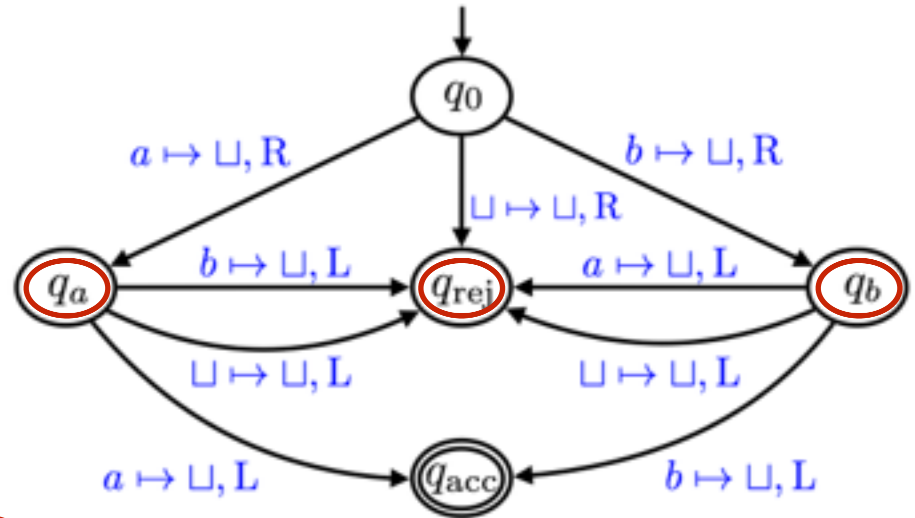
TM as a programming language

```
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;

   ⋮
```

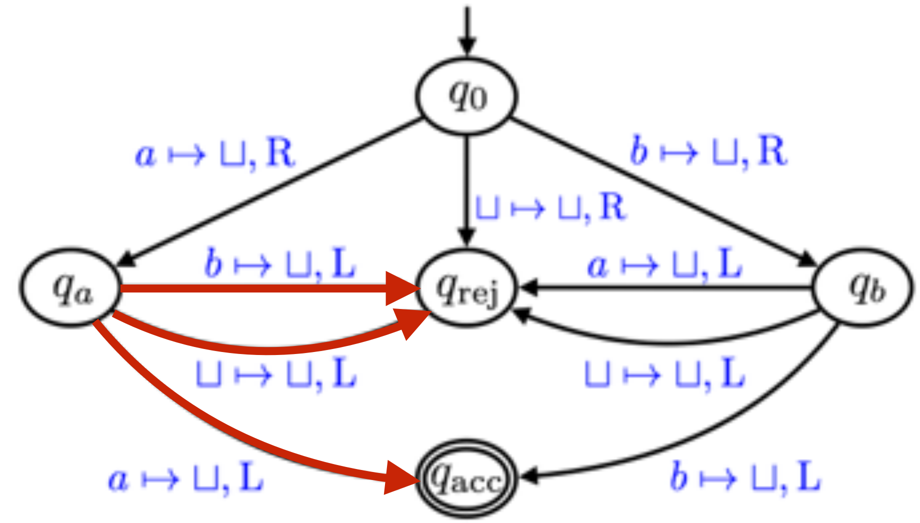


TM as a programming language

```
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;

    ⋮
```

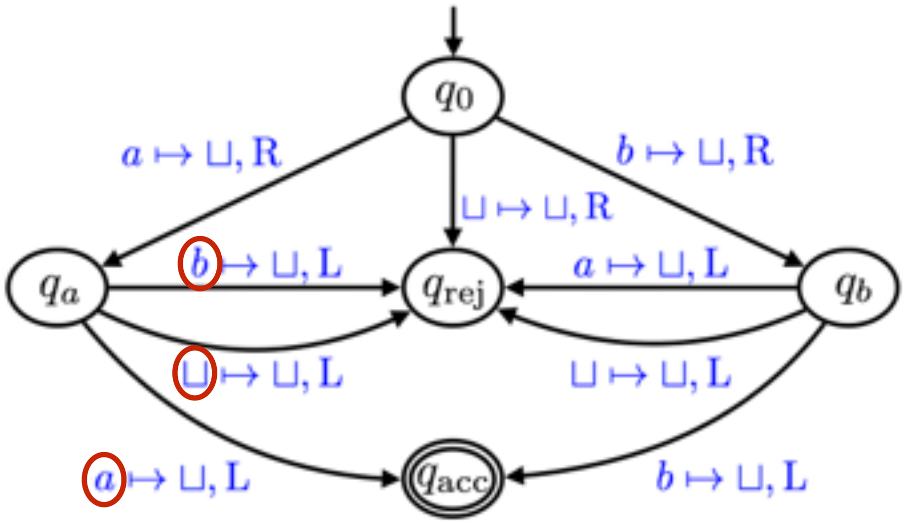


TM as a programming language




```
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;

    ⋮
```

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):

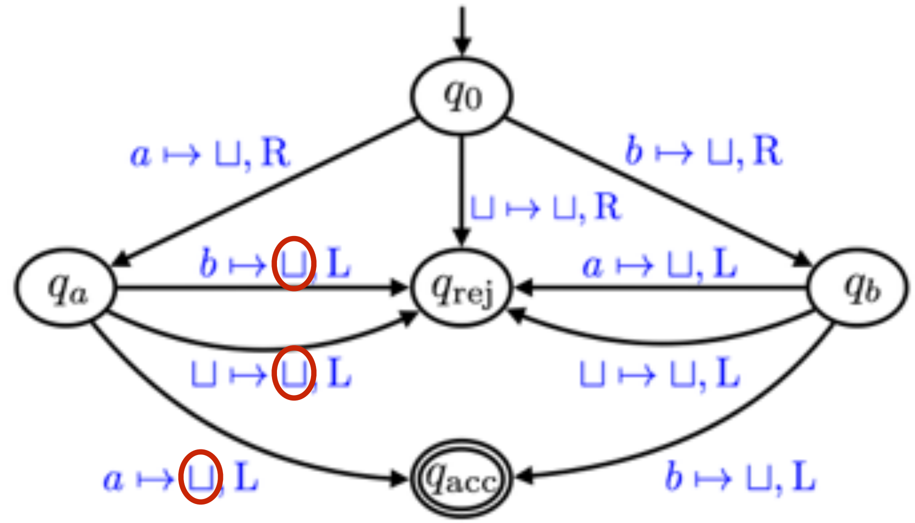
      case 'a': input[i] = ' '; i--; go to STATE **acc**;

      case 'b': input[i] = ' '; i--; go to STATE **rej**;

      case ' ': input[i] = ' '; i--; go to STATE **rej**;

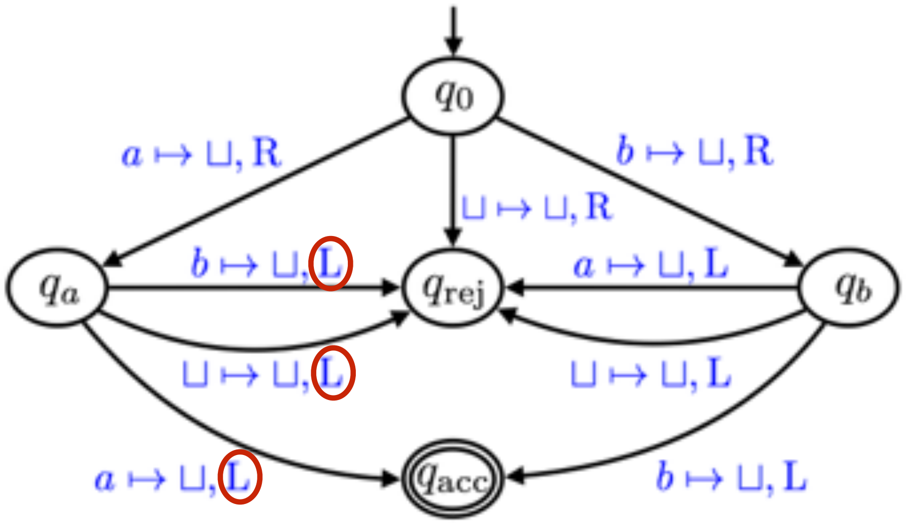
⋮

TM as a programming language



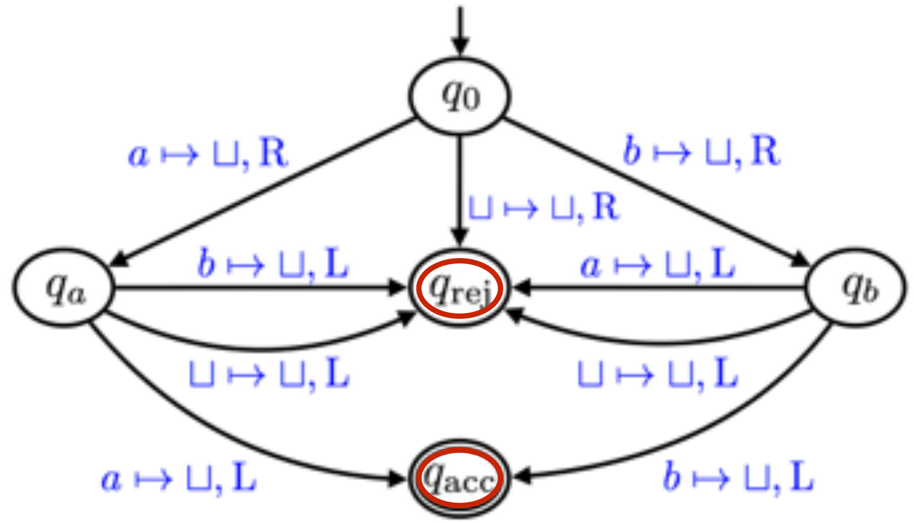

```
def foo(input):
    i = 0
    STATE 0:
        letter = input[i];
        switch(letter):
            case 'a':  input[i] = ' '; i++; go to STATE a;
            case 'b':  input[i] = ' '; i++; go to STATE b;
            case ' ':  input[i] = ' '; i++; go to STATE rej;
    STATE a:
        letter = 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;

    ⋮
```

TM as a programming language



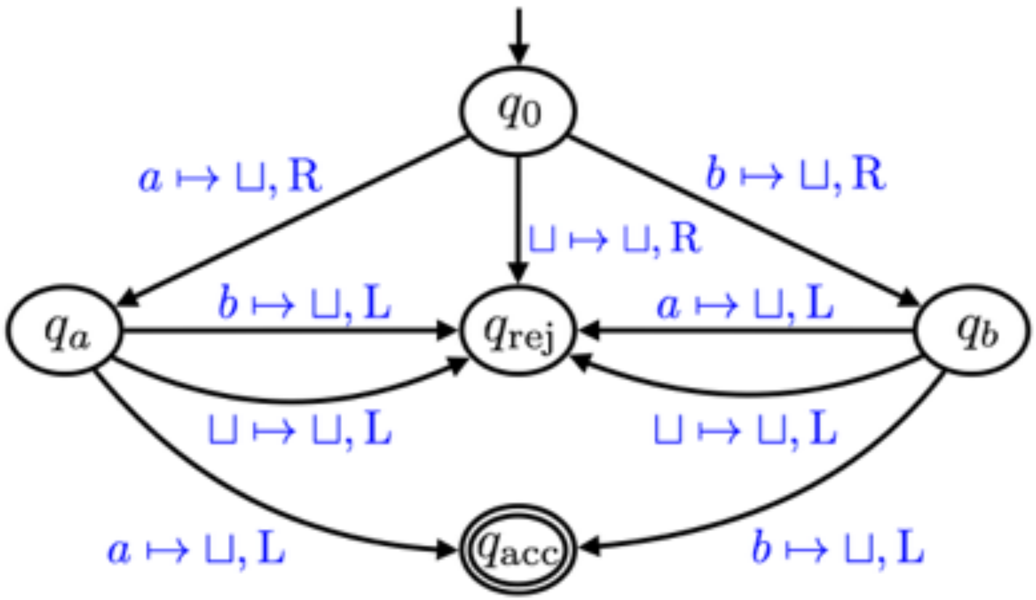

```
def foo(input):
    i = 0
    STATE 0:
        letter = input[i];
        switch(letter):
            case 'a':  input[i] = ' '; i++; go to STATE a;
            case 'b':  input[i] = ' '; i++; go to STATE b;
            case '  ':  input[i] = ' '; i++; go to STATE rej;
    STATE a:
        letter = 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;

    ⋮
```

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;

    ⋮
```

Poll



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] ≠ x[1]
- |x| > 1 and x[0] = x[1]
- x[0] = x[1]
- None of these.
- Beats me.

Exercise

Let $\Sigma = \{a, b\}$.

Draw the state diagram of a TM that accepts a string iff it starts and ends with an a.

Formal definition: Turing machine

A Turing machine (TM) M is a 7-tuple

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

where

- Q is a finite set (which we call the set of states);
- Σ is a finite set with $\sqcup \notin \Sigma$
  (which we call the input alphabet);
- Γ is a finite set with $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$
  (which we call the tape alphabet);
- δ is a function of the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$
  (which we call the transition function);
- $q_0 \in Q$ (which we call the start state);
- $q_{\mathrm{acc}} \in Q$ (which we call the accept state);
- $q_{\mathrm{rej}} \in Q$, $q_{\mathrm{rej}} \neq q_{\mathrm{acc}}$ (which we call the reject state);

Formal definition: TM accepting a string

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?

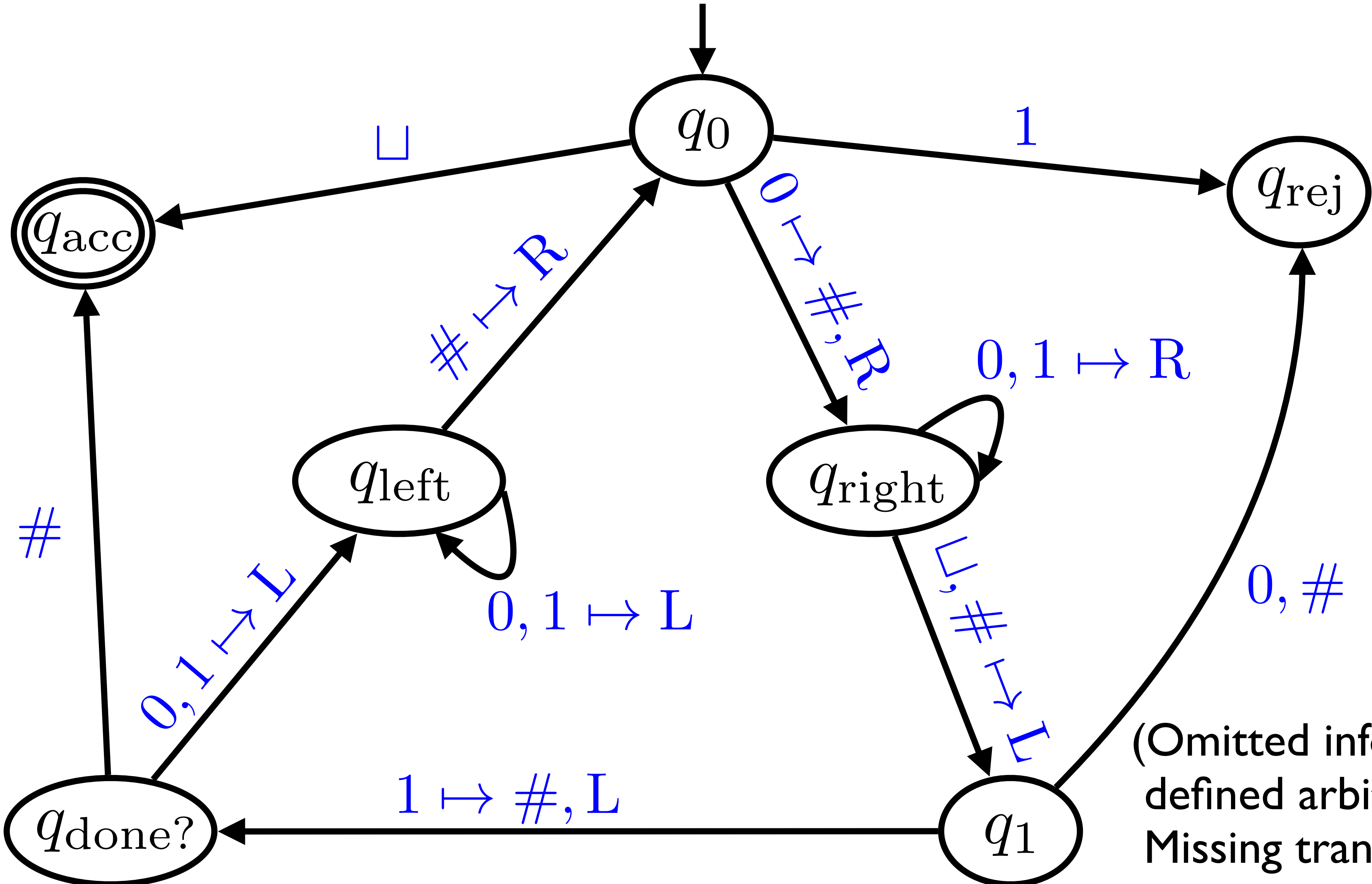
Definition: A TM is called a *decider* if it halts on all inputs.

Definition: A language L is called *decidable* (or *computable*) if $L = L(M)$ for some decider TM M.

regular languages $\stackrel{?}{=}$ decidable languages

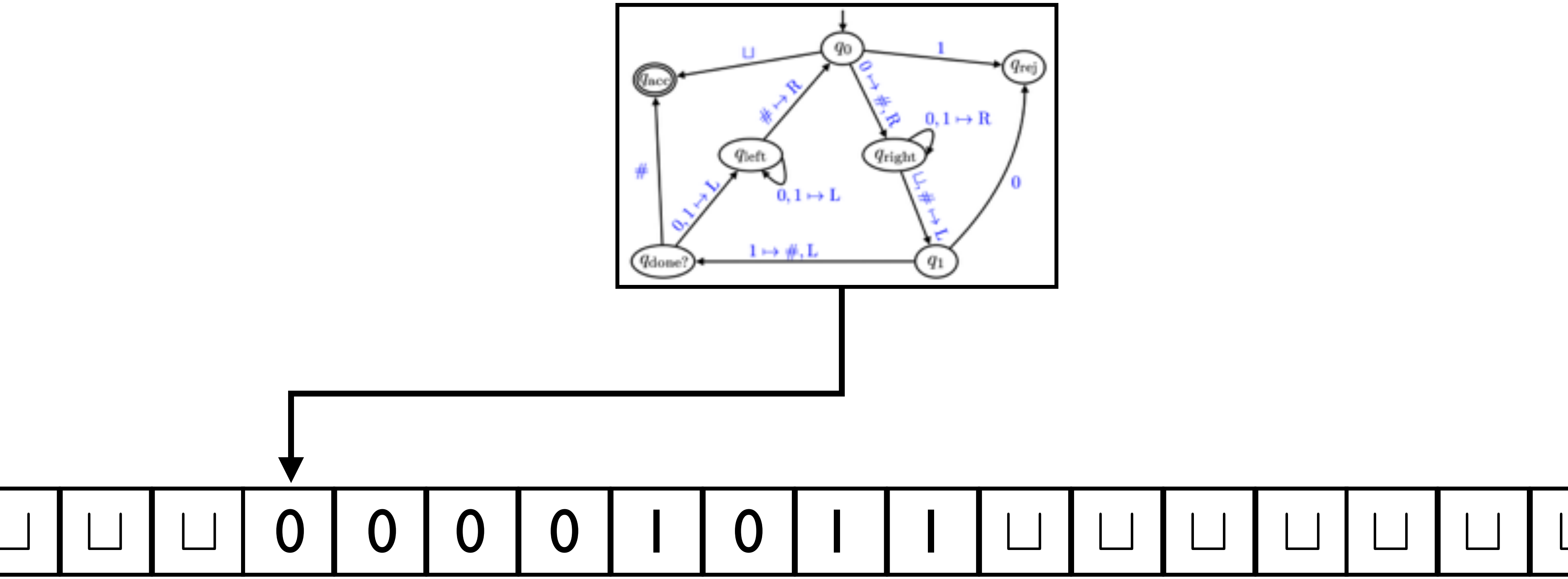
Turing machine that decides $0^n 1^n$

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



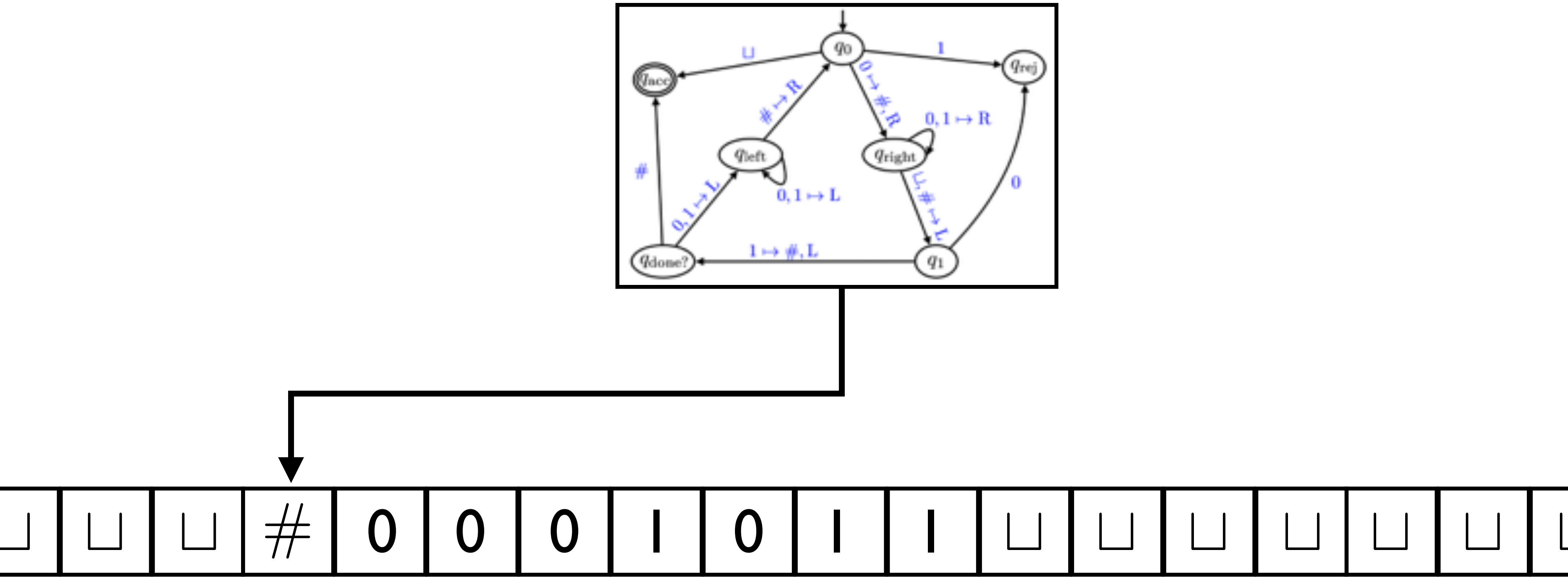
(Omitted information defined arbitrarily. Missing transitions go to the reject state.)

Turing machine that decides 0^n1^n



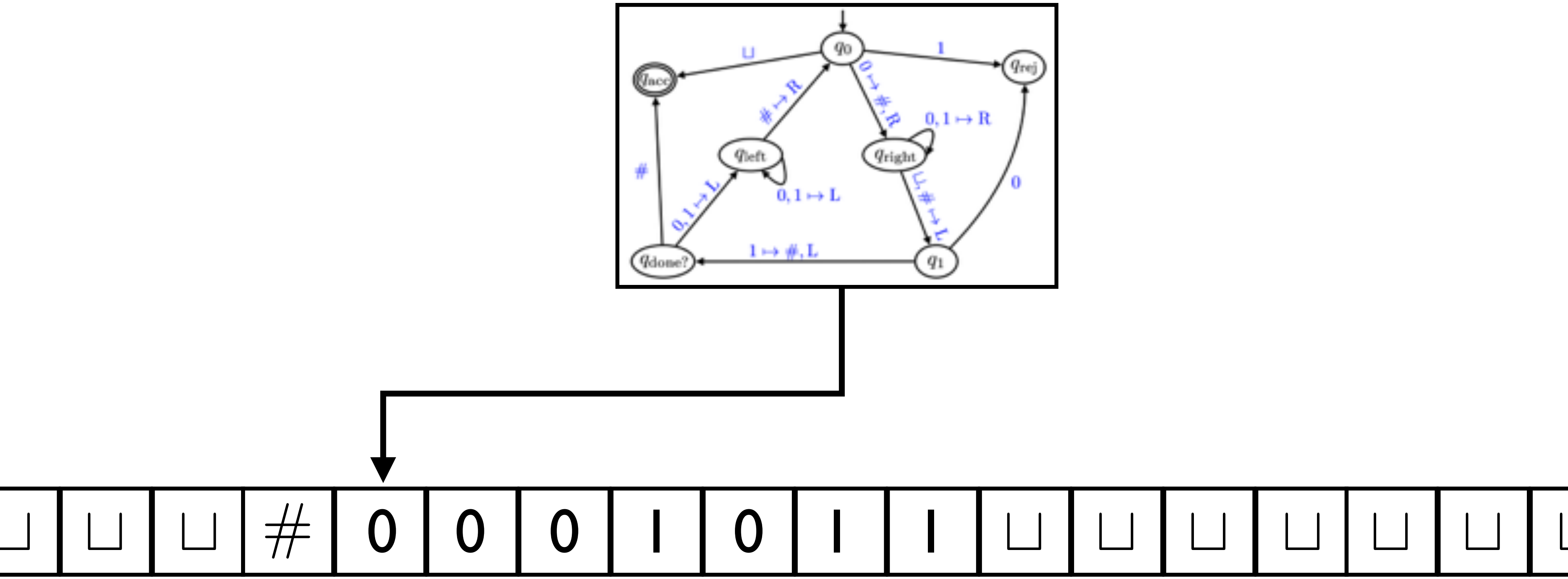
Input: 00001011

Turing machine that decides 0^n1^n



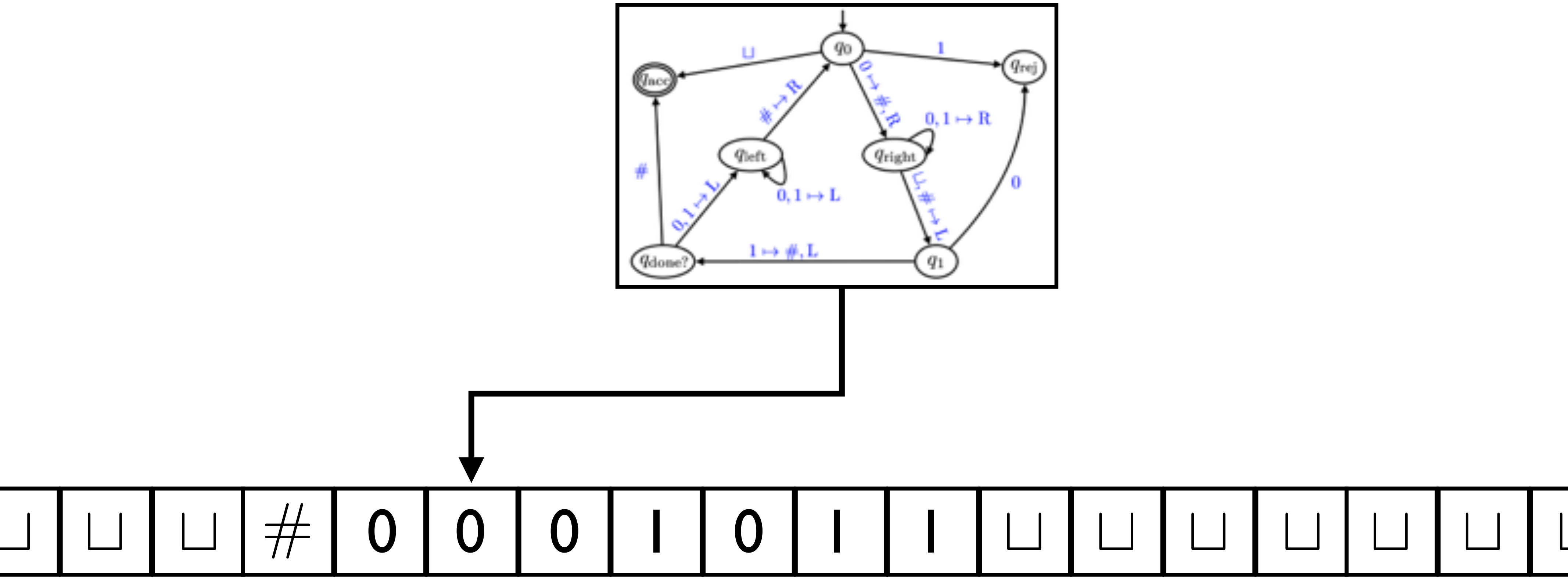
Input: 00001011

Turing machine that decides 0^n1^n



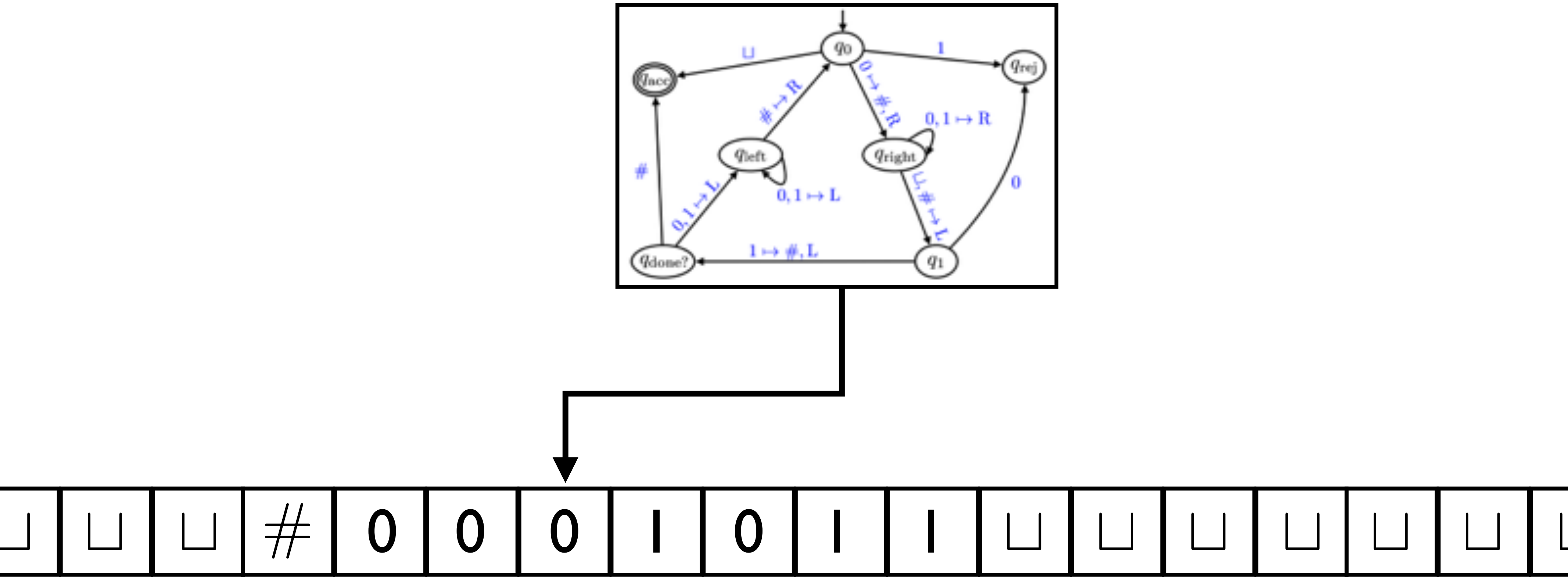
Input: 00001011

Turing machine that decides 0^n1^n



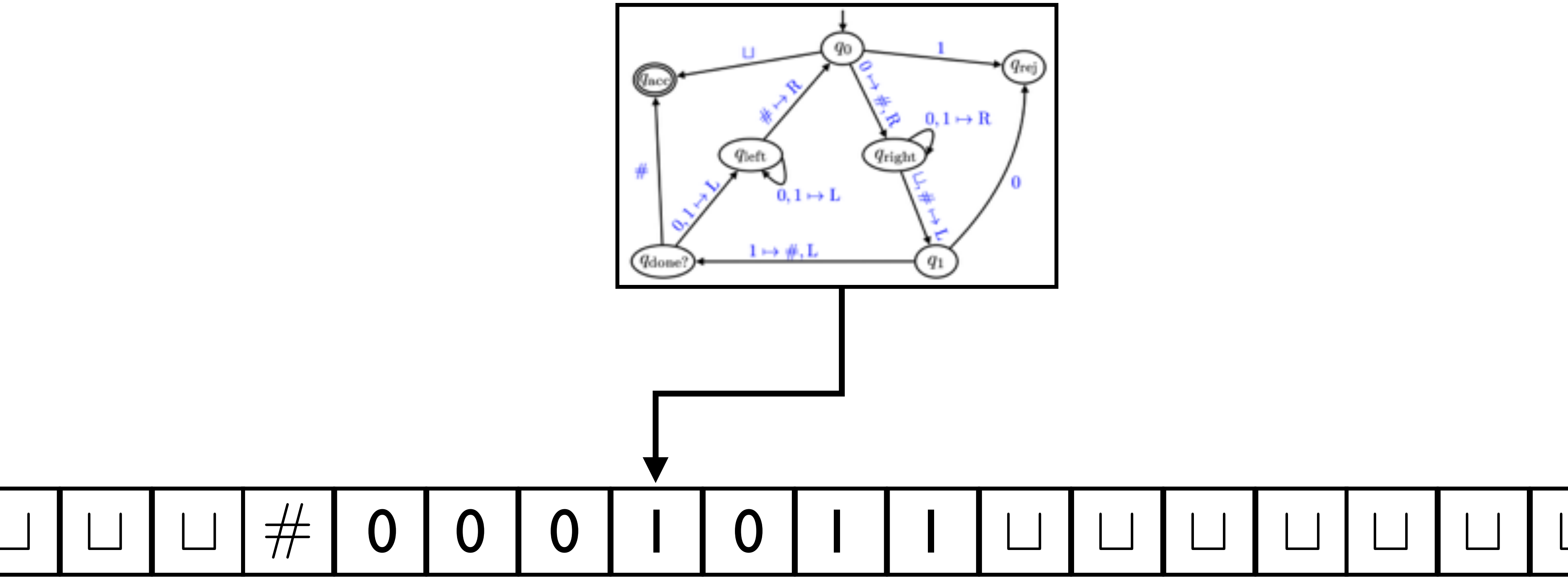
Input: 00001011

Turing machine that decides 0^n1^n



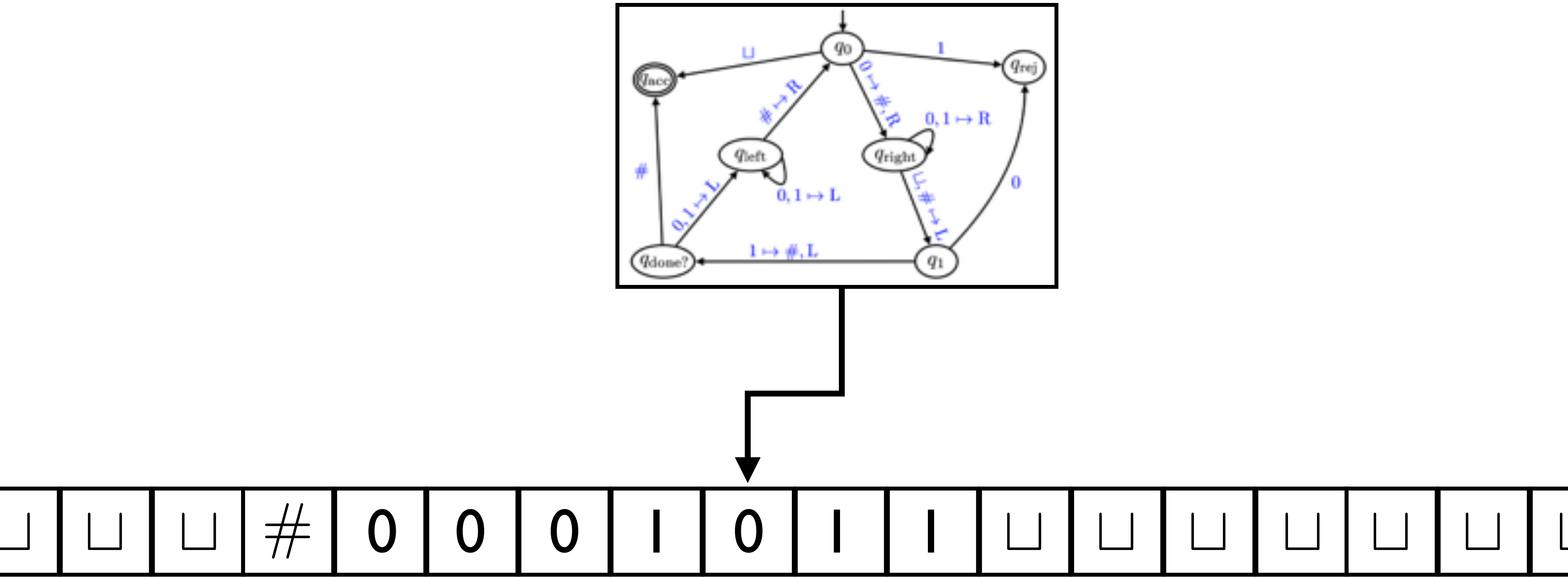
Input: 00001011

Turing machine that decides 0^n1^n



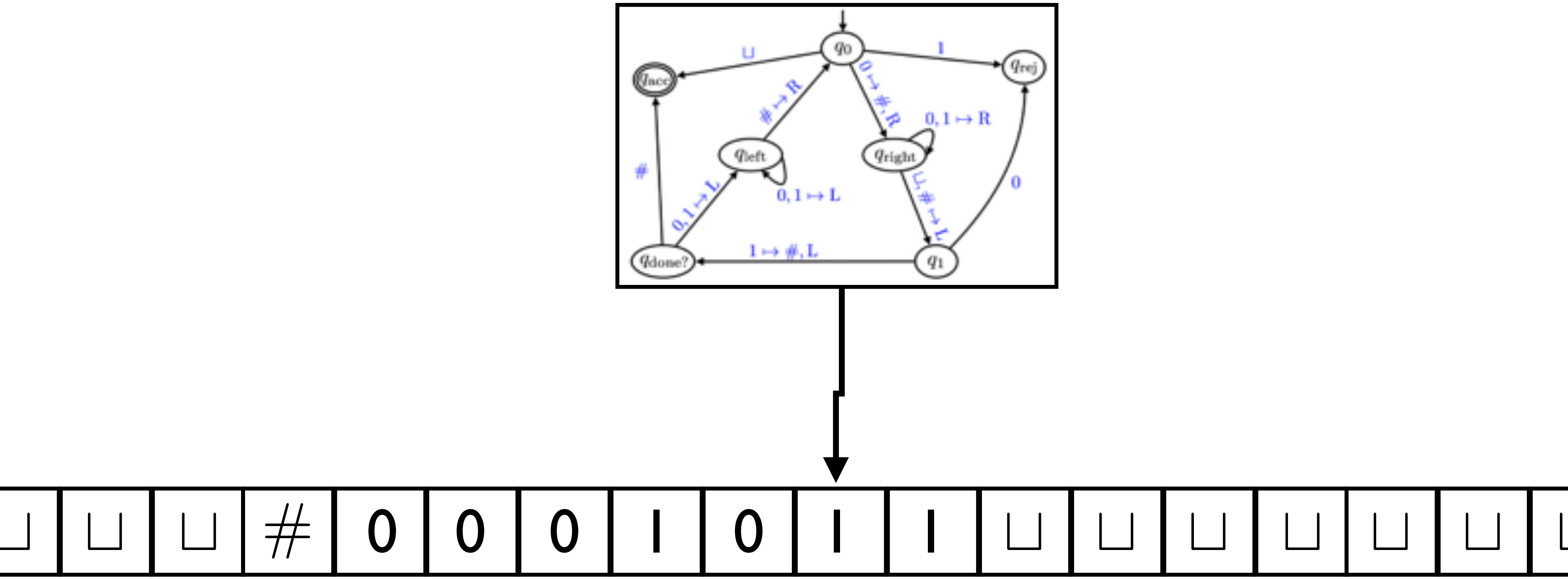
Input: 00001011

Turing machine that decides 0^n1^n



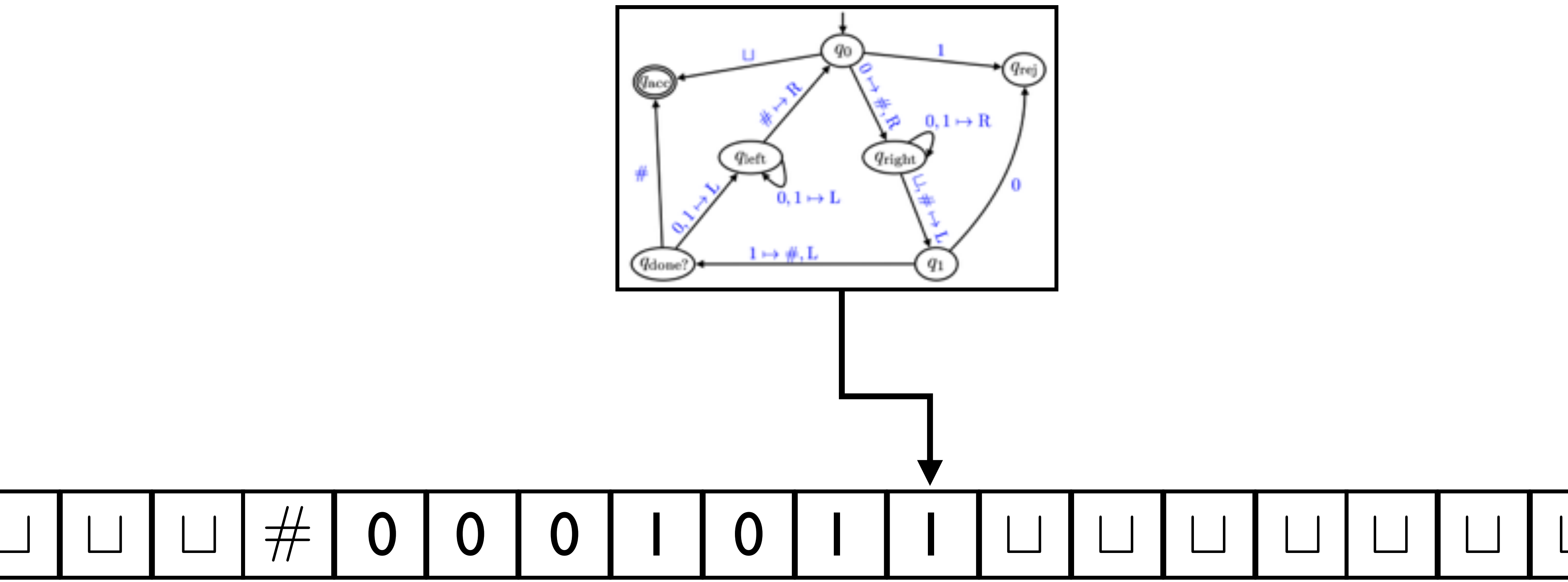
Input: 00001011

Turing machine that decides 0^n1^n



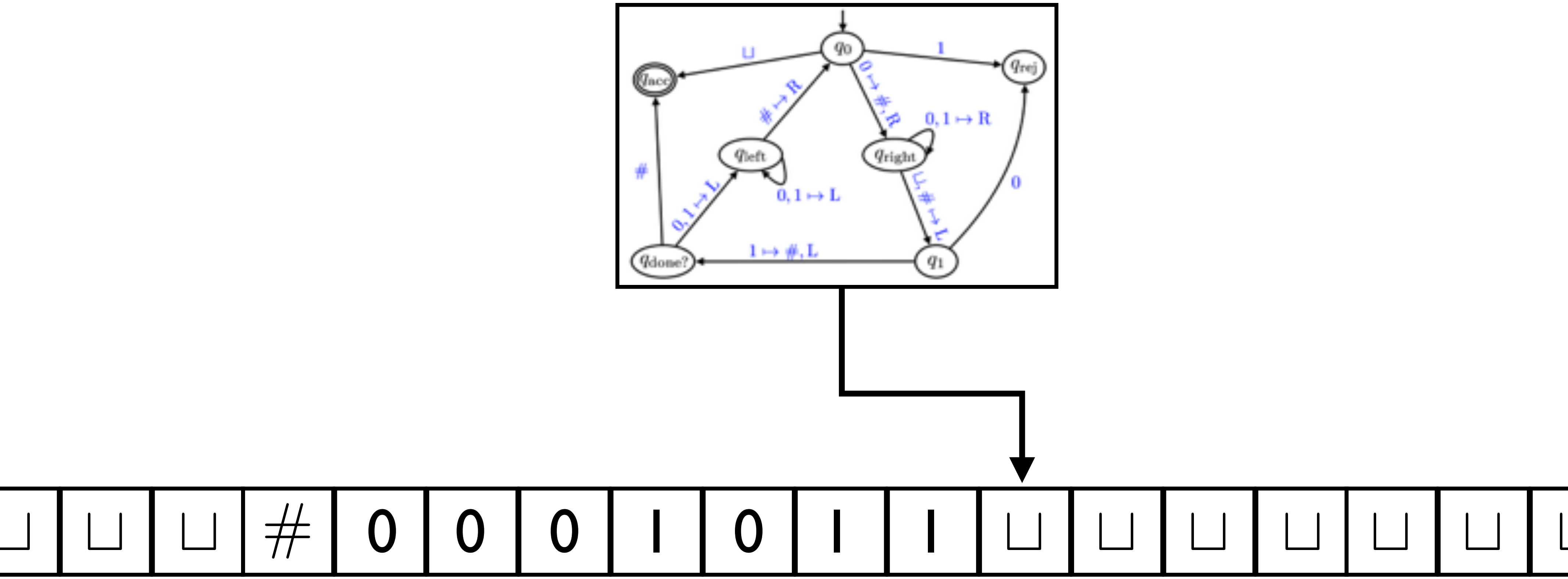
Input: 00001011

Turing machine that decides 0^n1^n



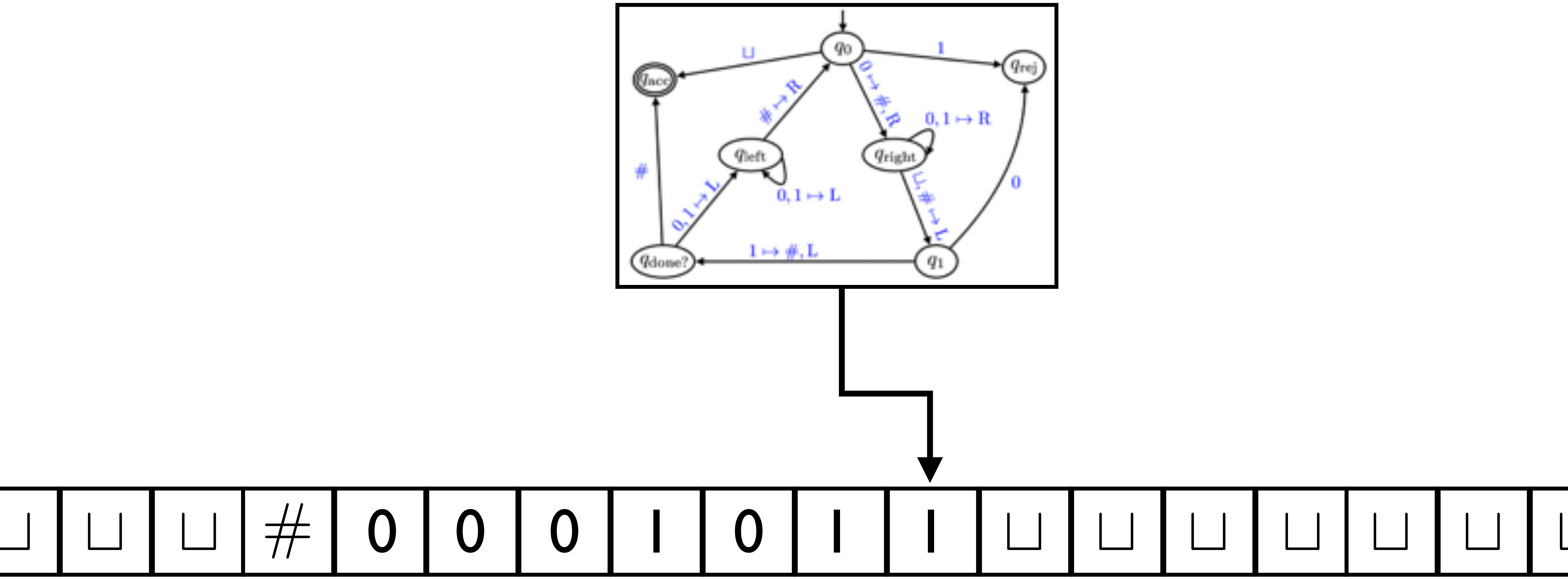
Input: 00001011

Turing machine that decides 0^n1^n



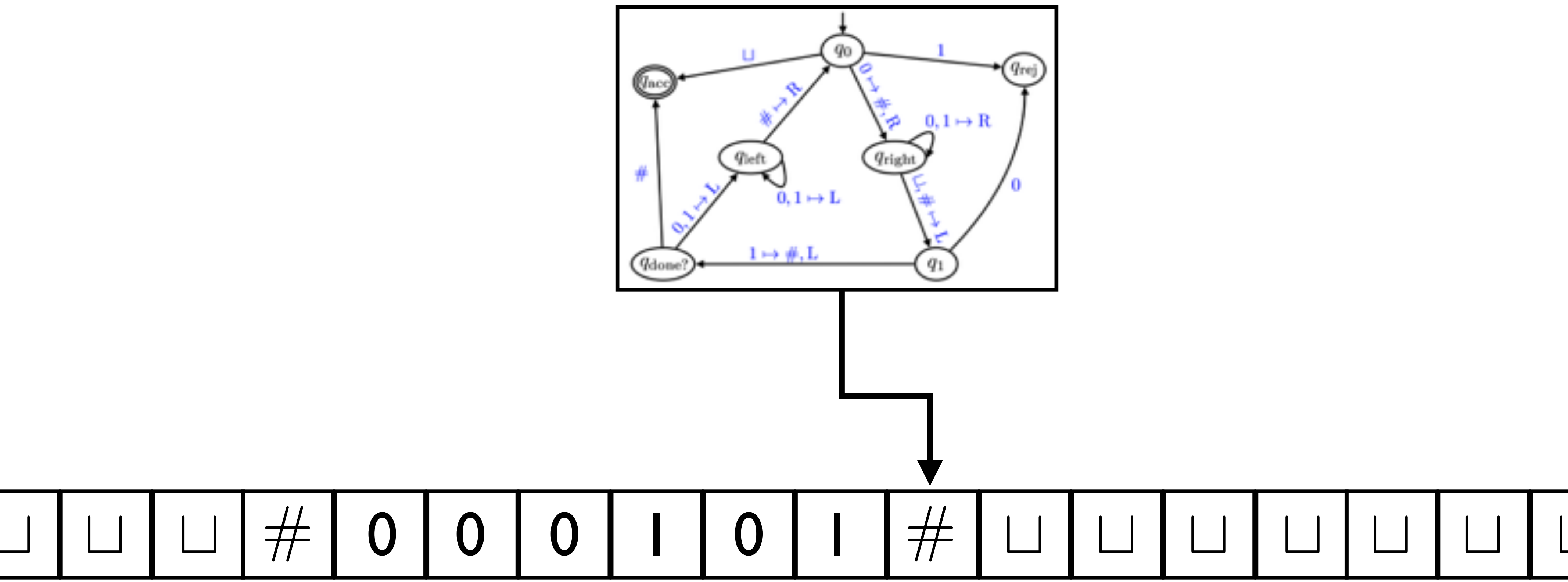
Input: 00001011

Turing machine that decides 0^n1^n



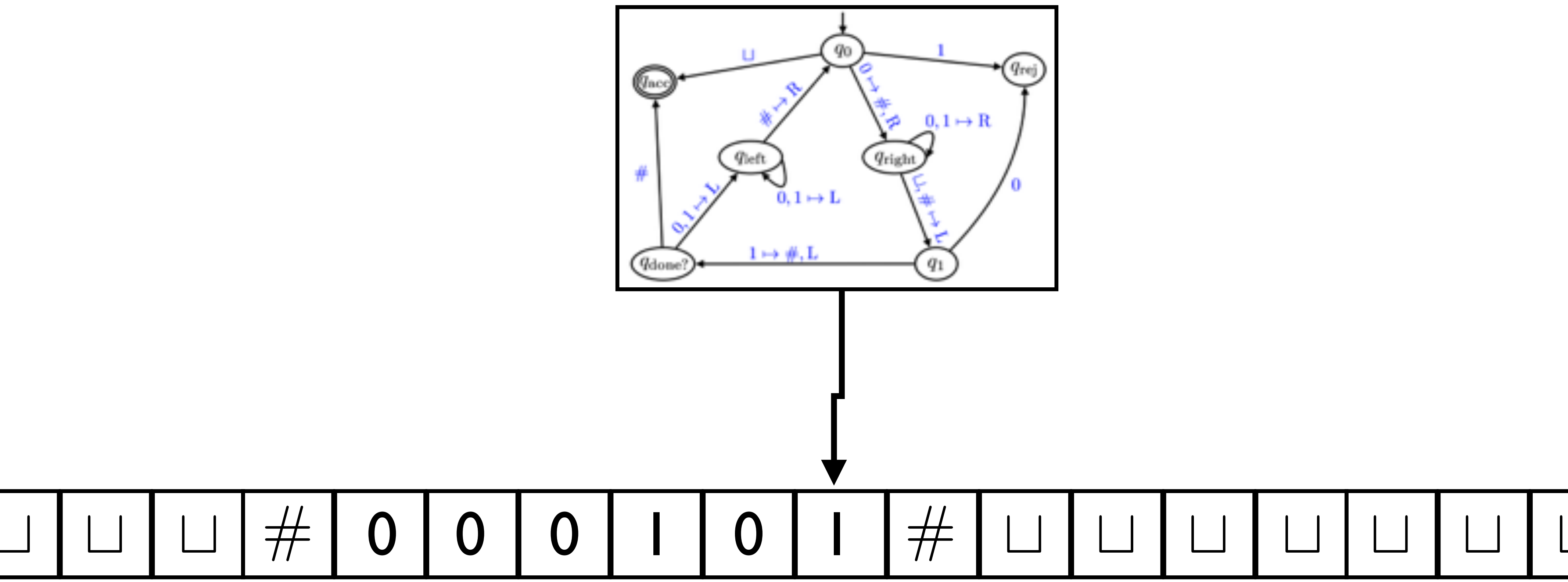
Input: 00001011

Turing machine that decides 0^n1^n



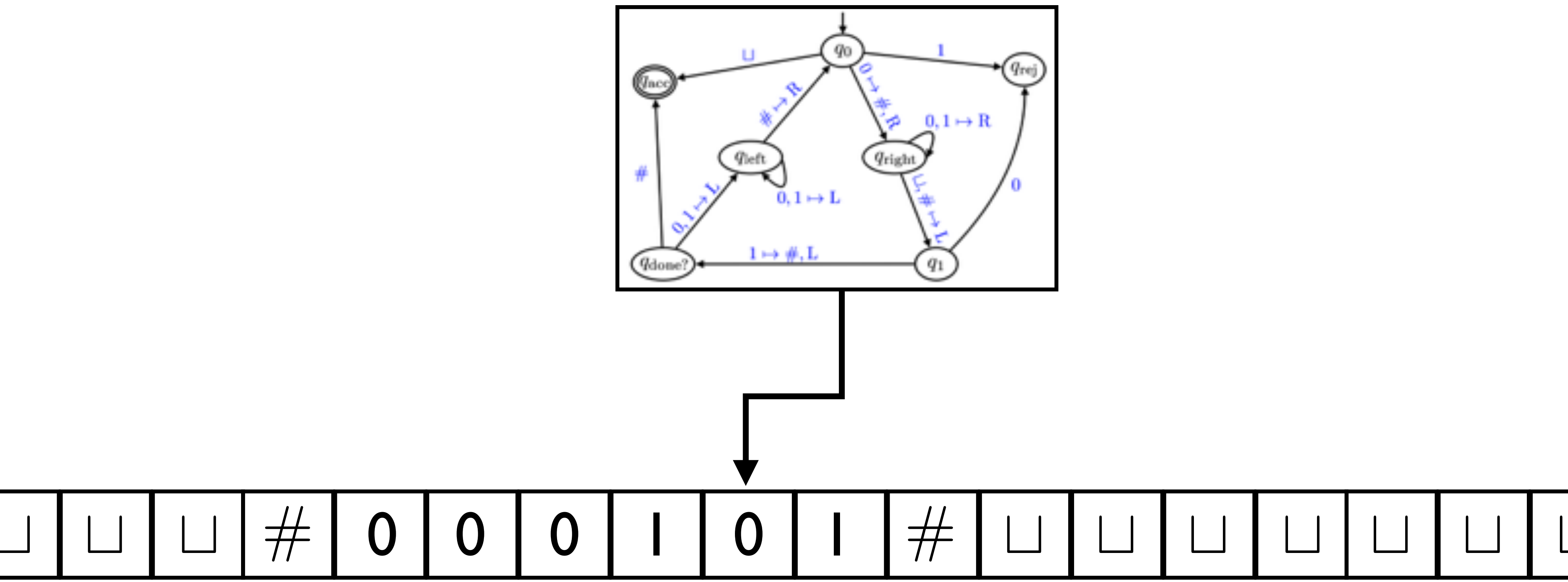
Input: 00001011

Turing machine that decides 0^n1^n



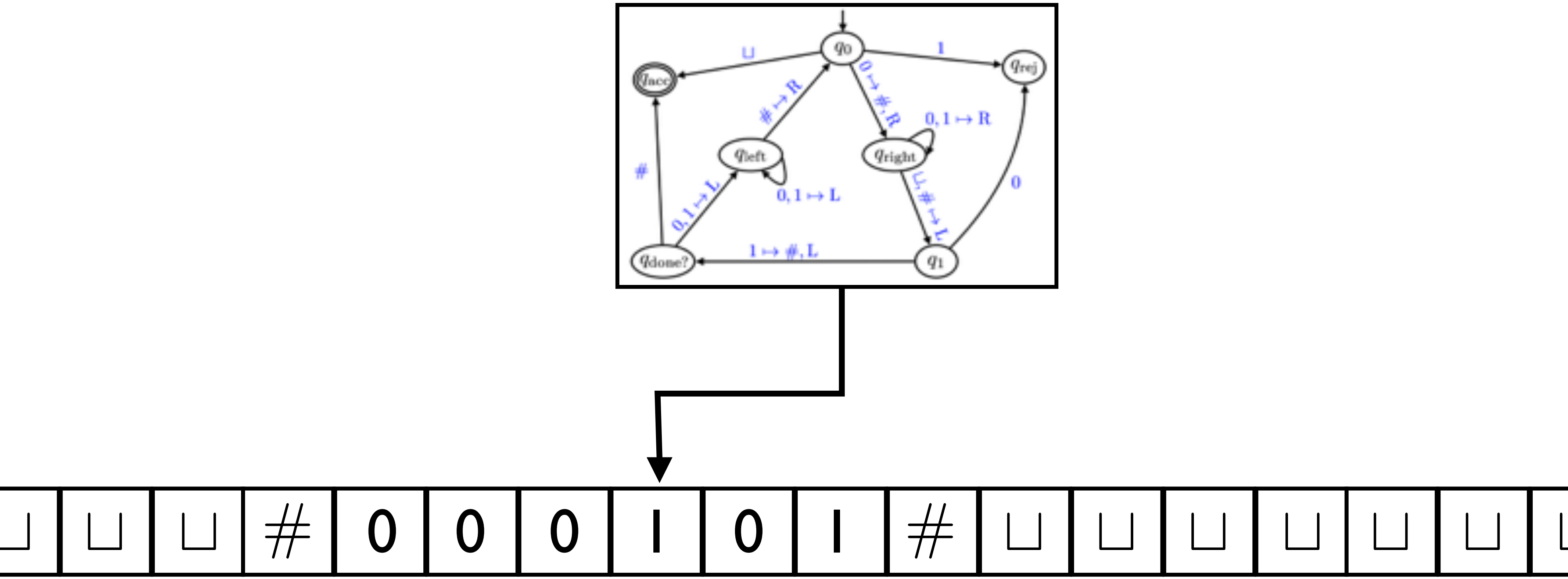
Input: 00001011

Turing machine that decides 0^n1^n



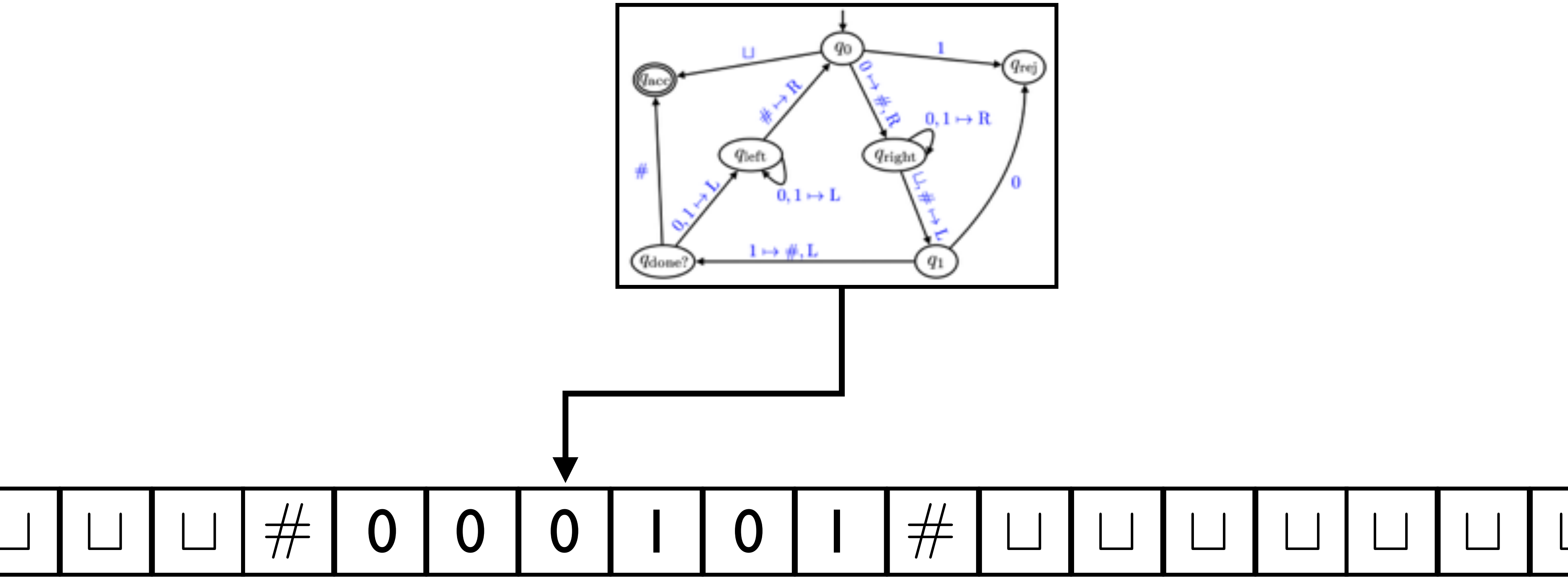
Input: 00001011

Turing machine that decides 0^n1^n



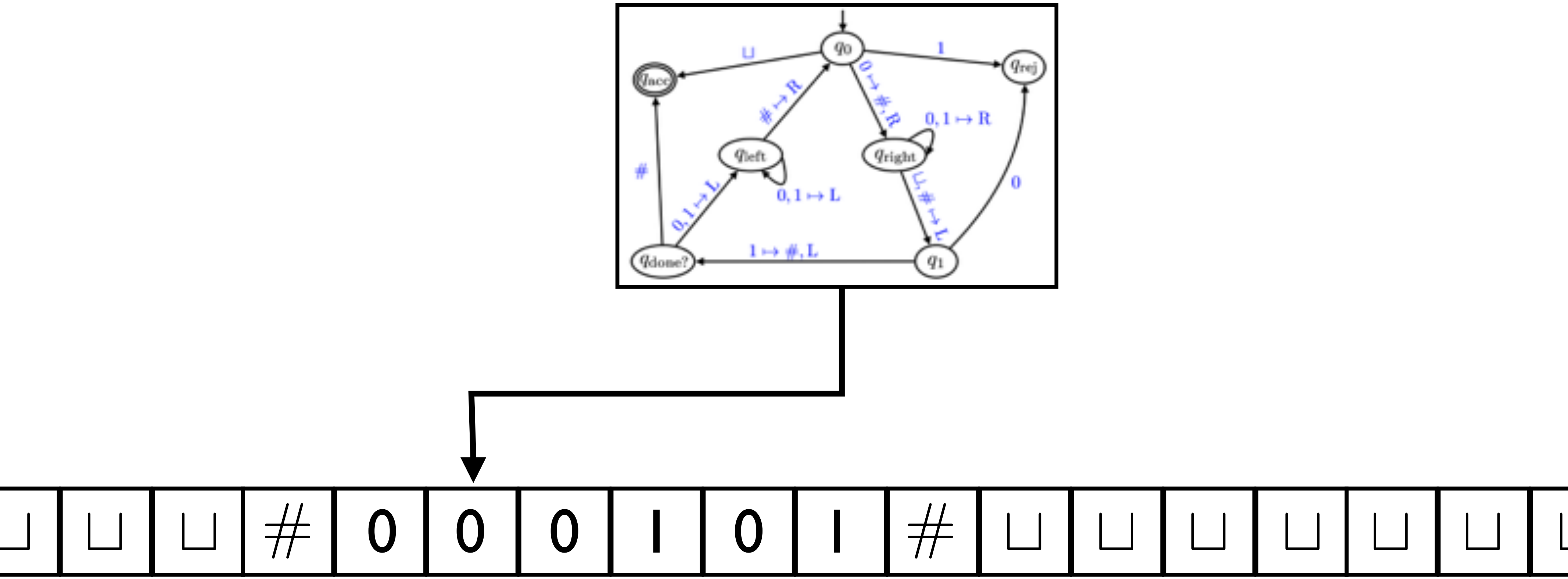
Input: 00001011

Turing machine that decides 0^n1^n



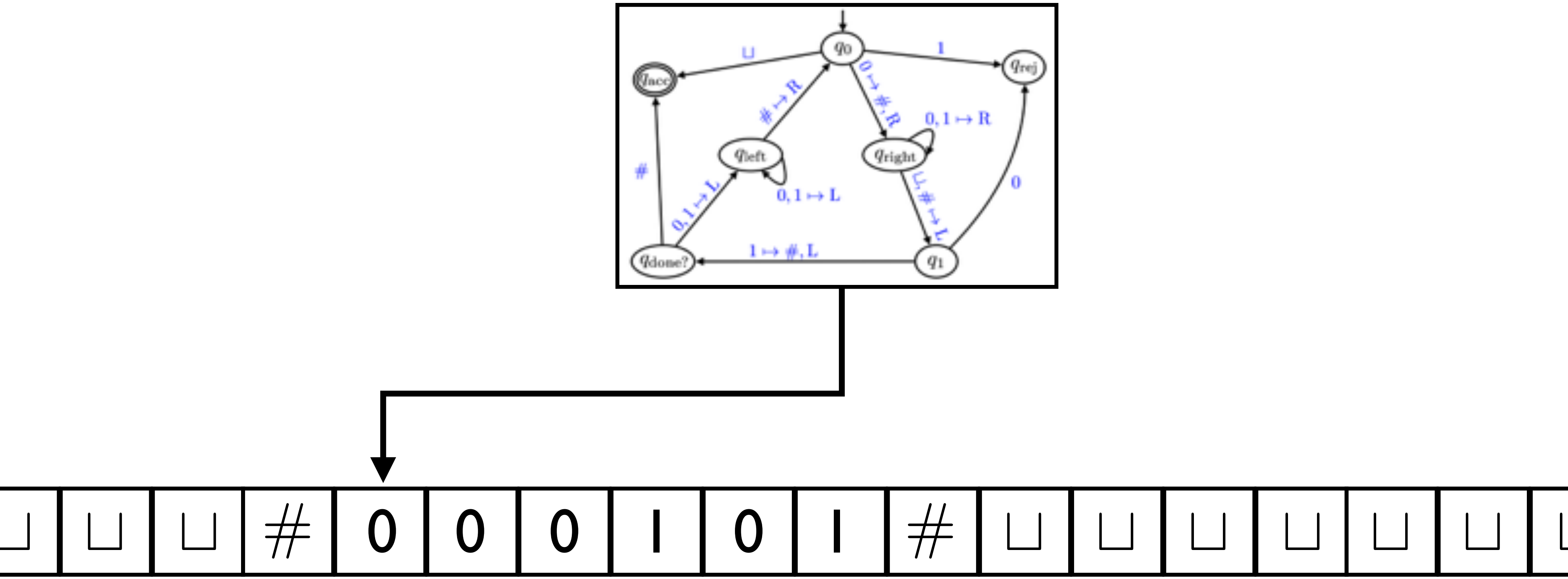
Input: 00001011

Turing machine that decides 0^n1^n



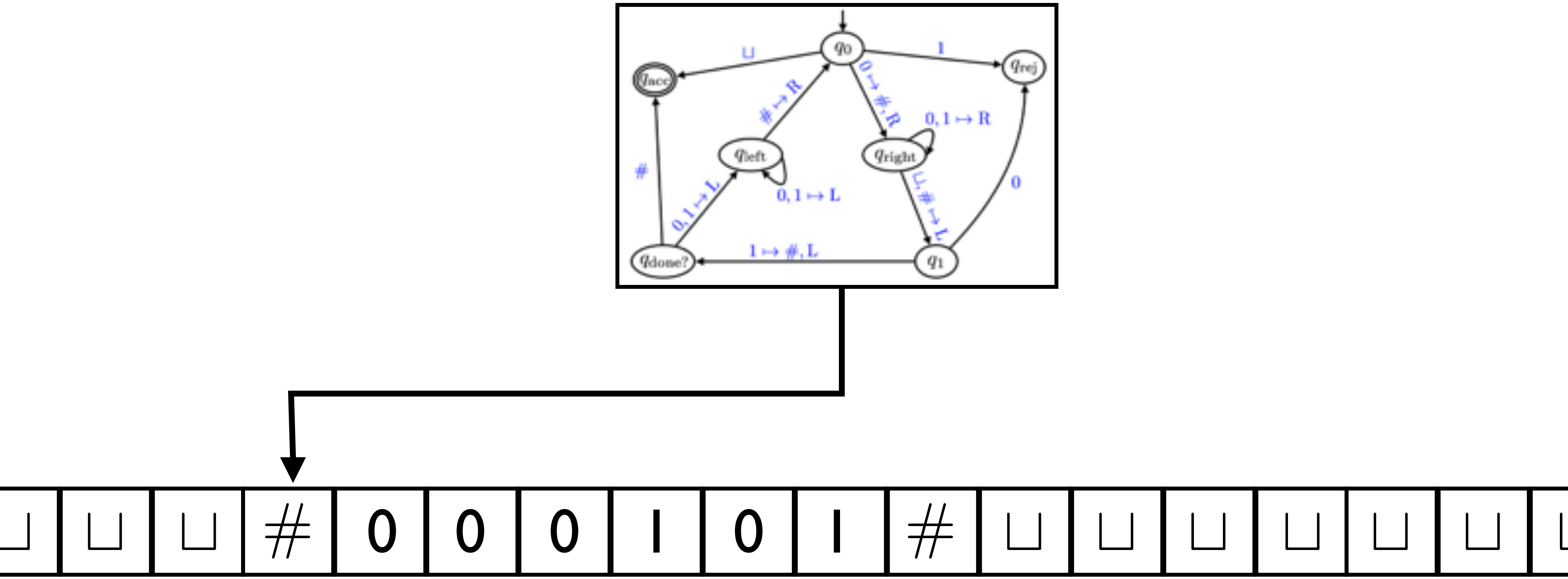
Input: 00001011

Turing machine that decides 0^n1^n



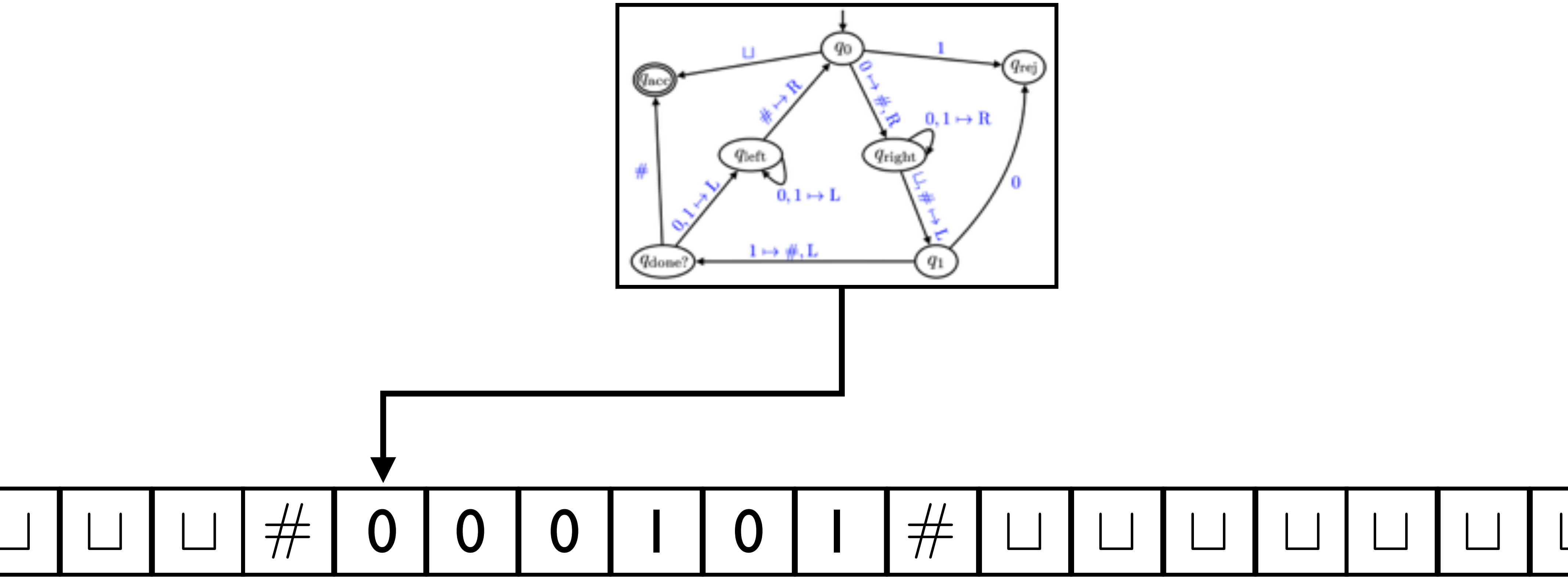
Input: 00001011

Turing machine that decides 0^n1^n



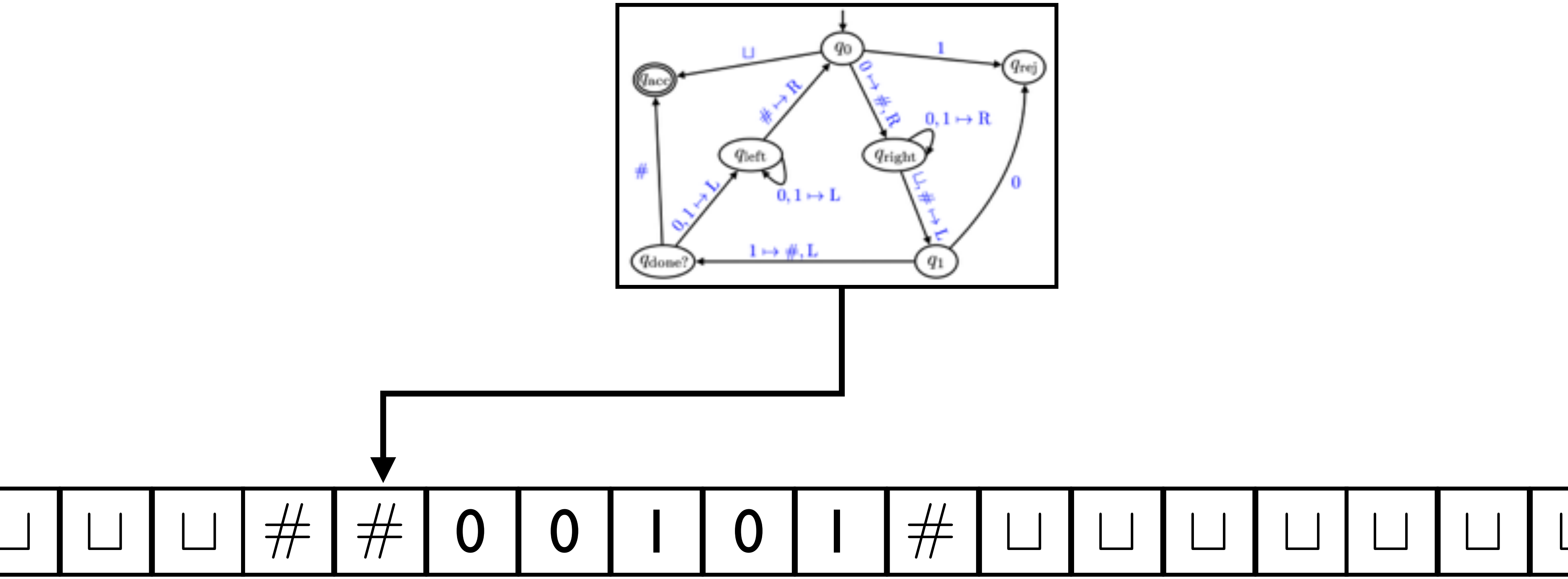
Input: 00001011

Turing machine that decides 0^n1^n



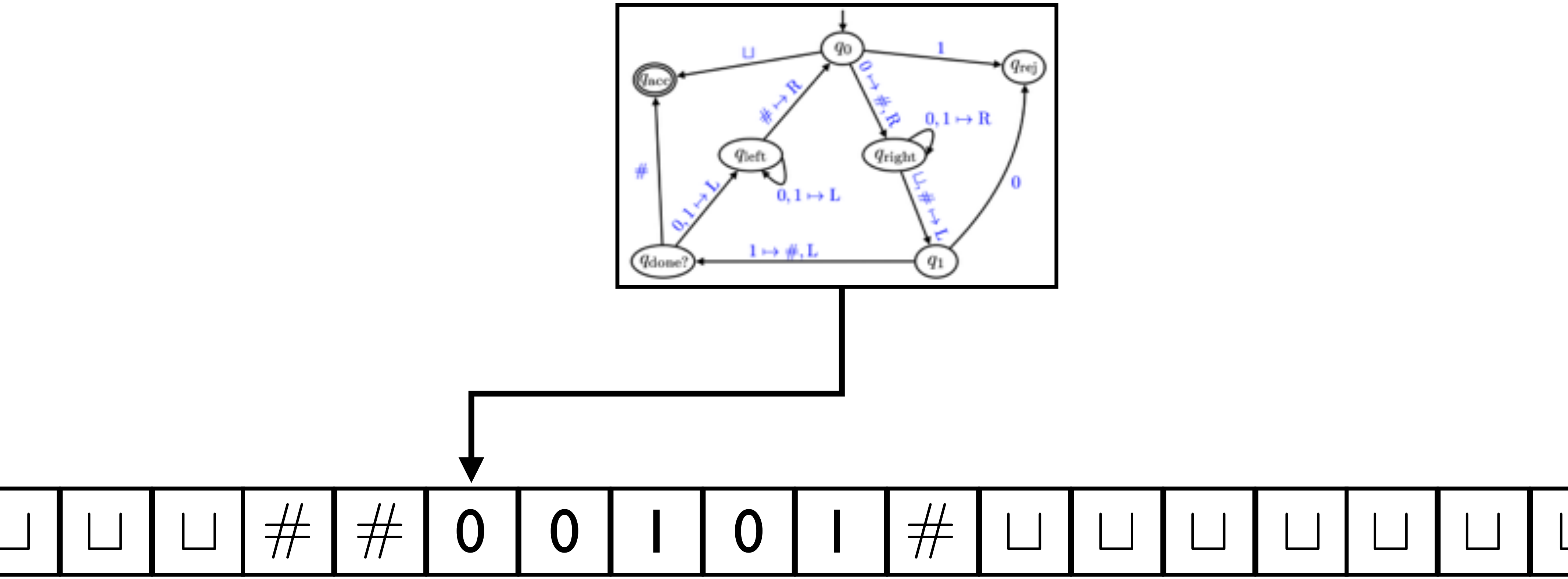
Input: 00001011

Turing machine that decides 0^n1^n



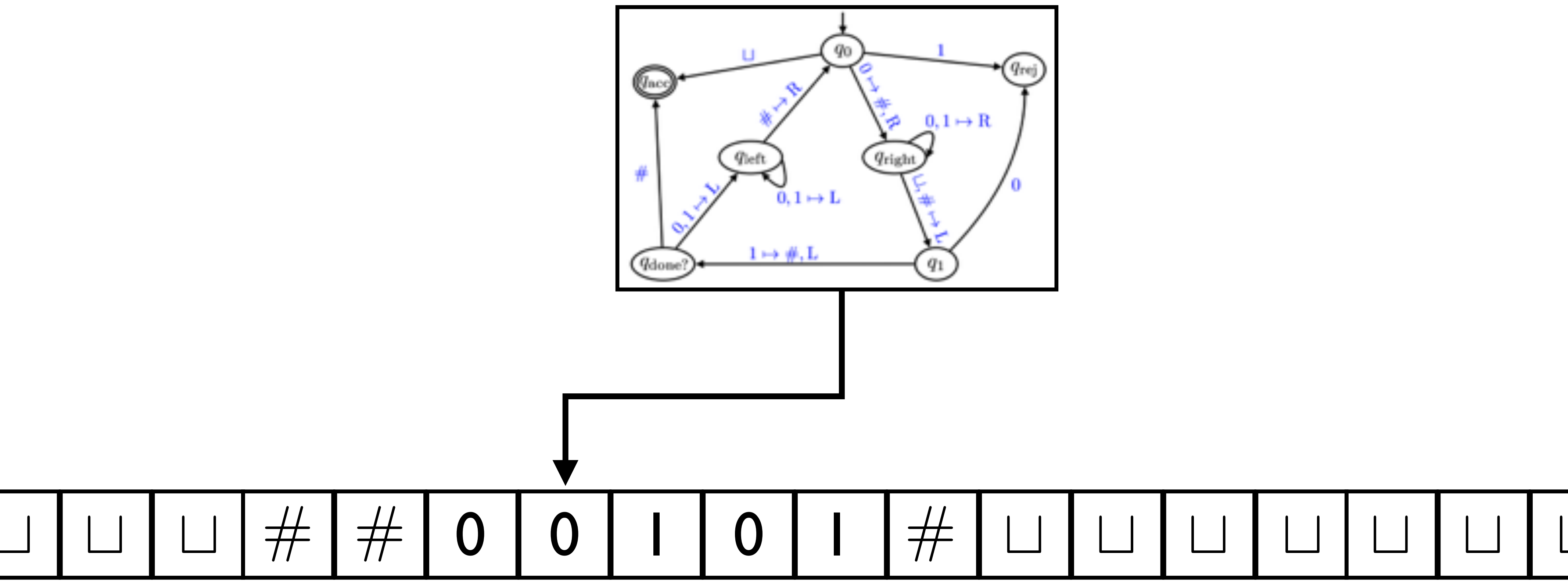
Input: 00001011

Turing machine that decides 0^n1^n



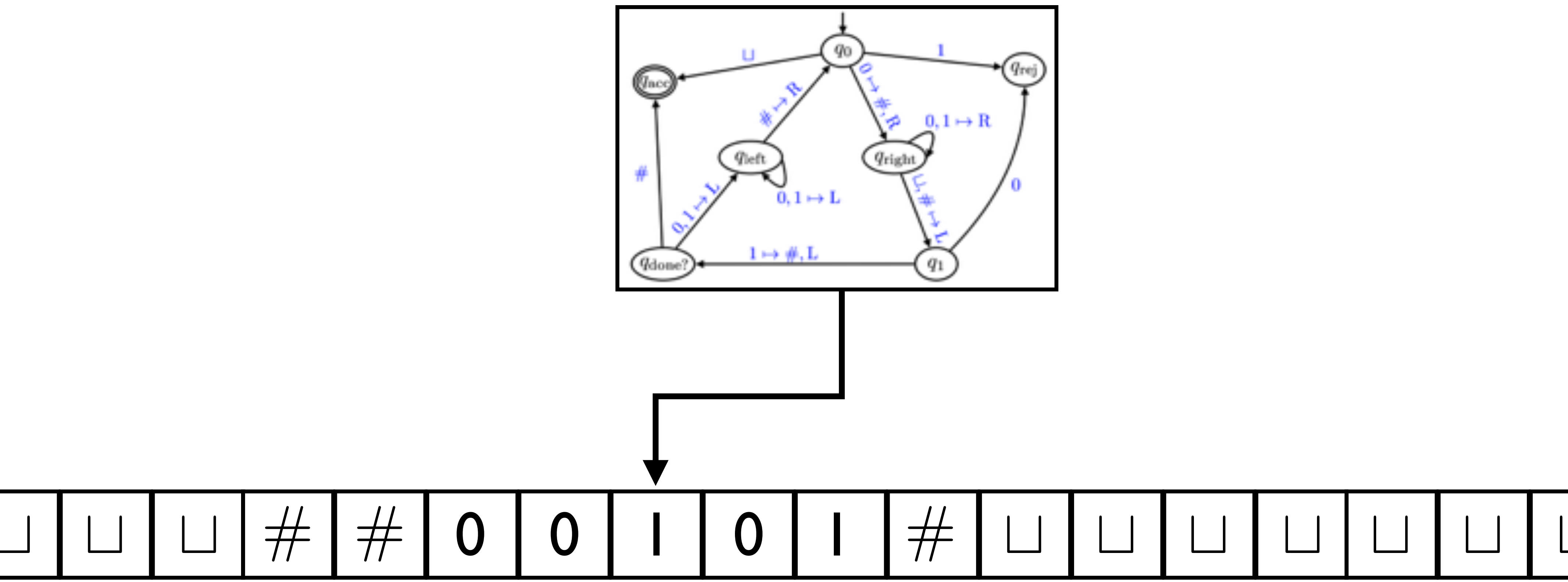
Input: 00001011

Turing machine that decides 0^n1^n



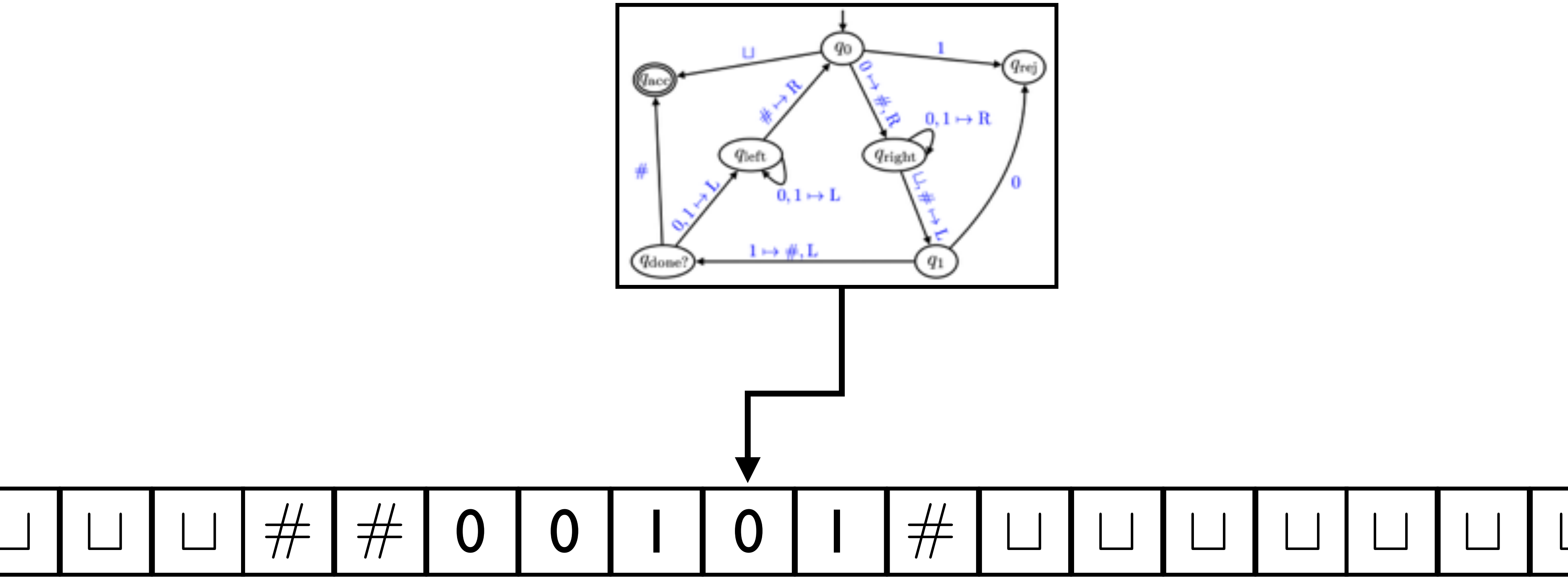
Input: 00001011

Turing machine that decides 0^n1^n



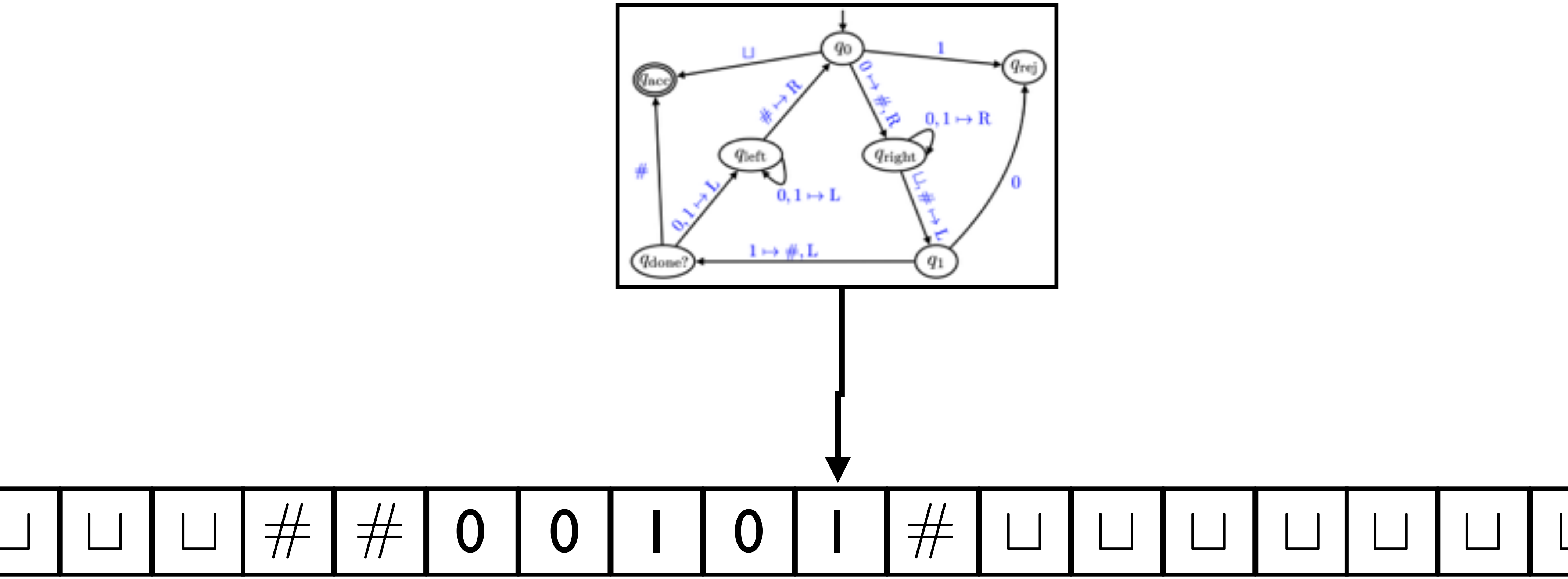
Input: 00001011

Turing machine that decides 0^n1^n



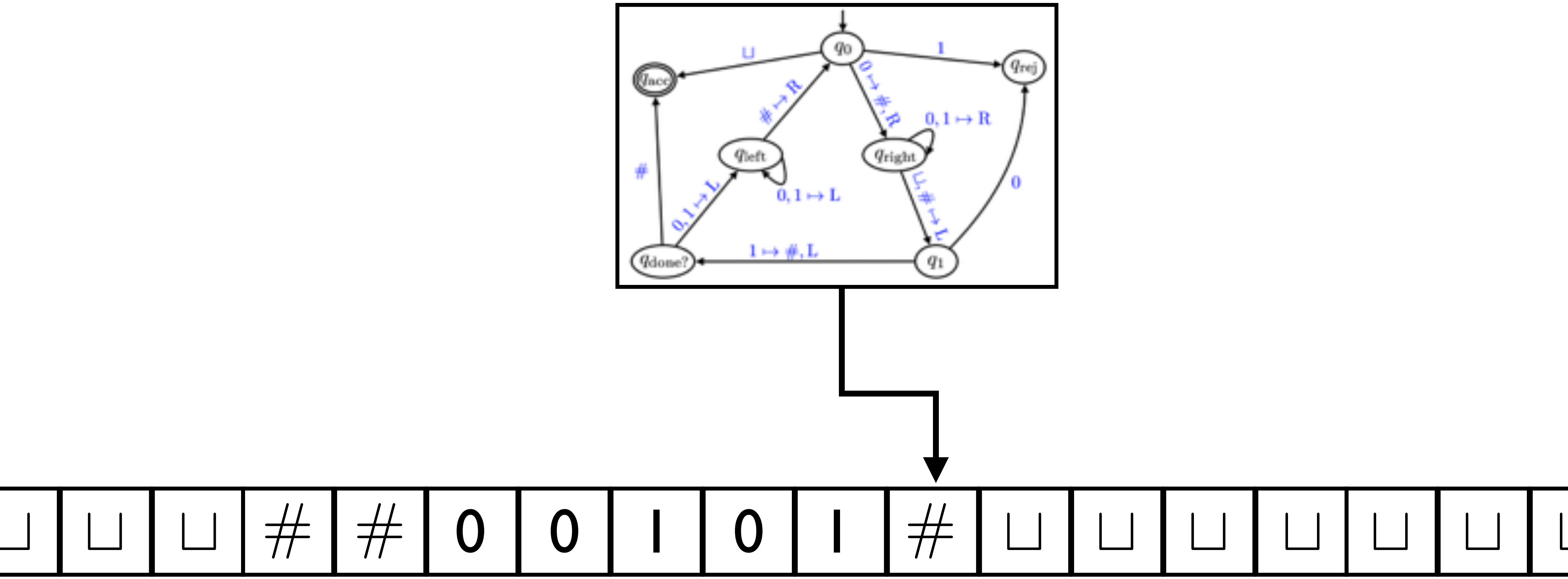
Input: 00001011

Turing machine that decides 0^n1^n



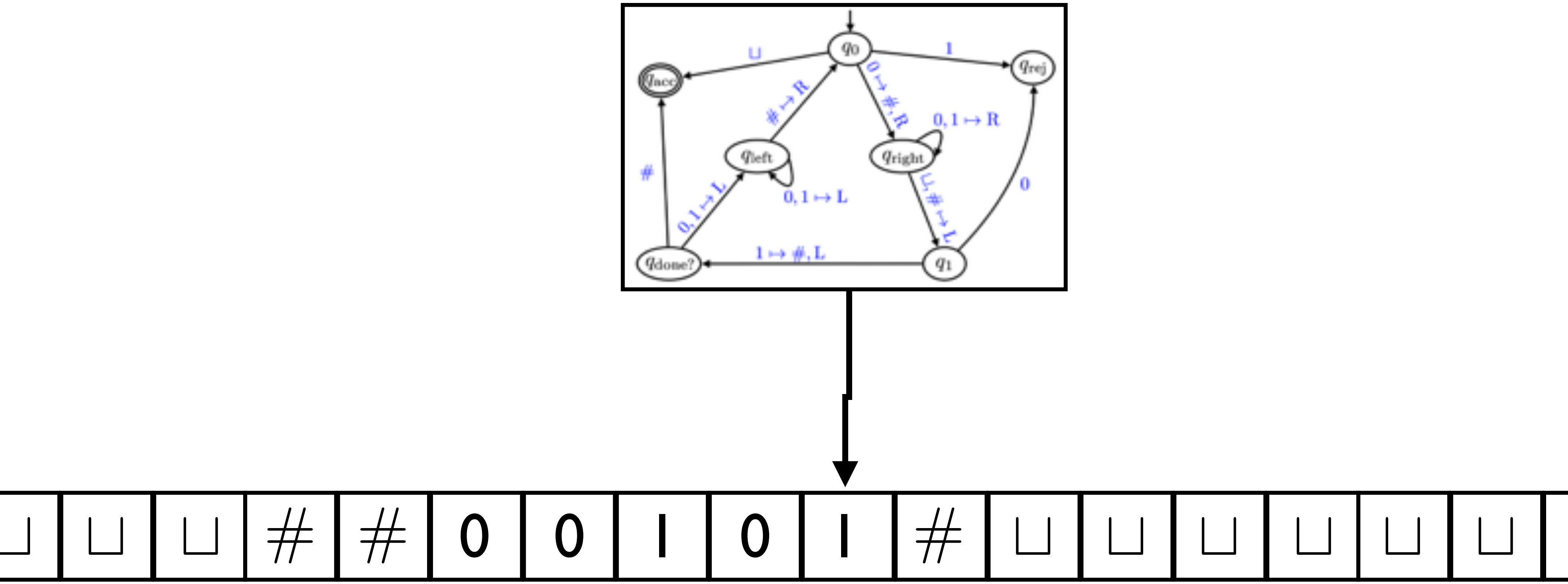
Input: 00001011

Turing machine that decides 0^n1^n



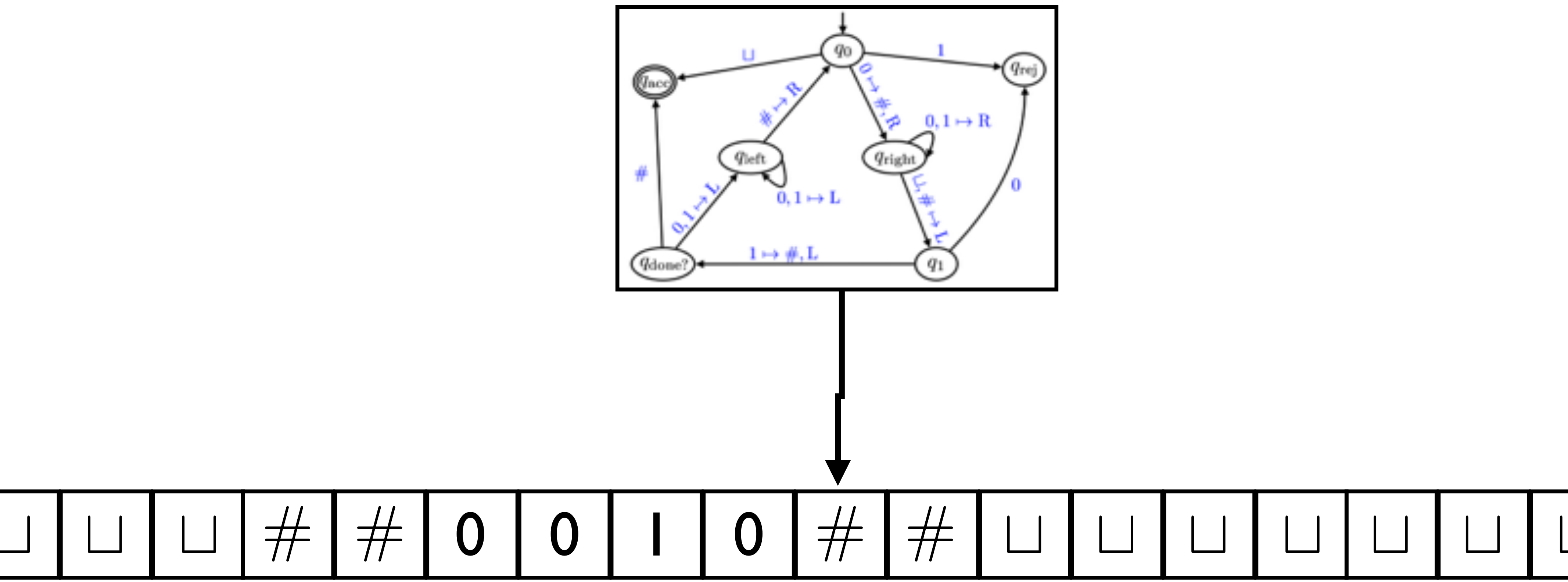
Input: 00001011

Turing machine that decides 0^n1^n



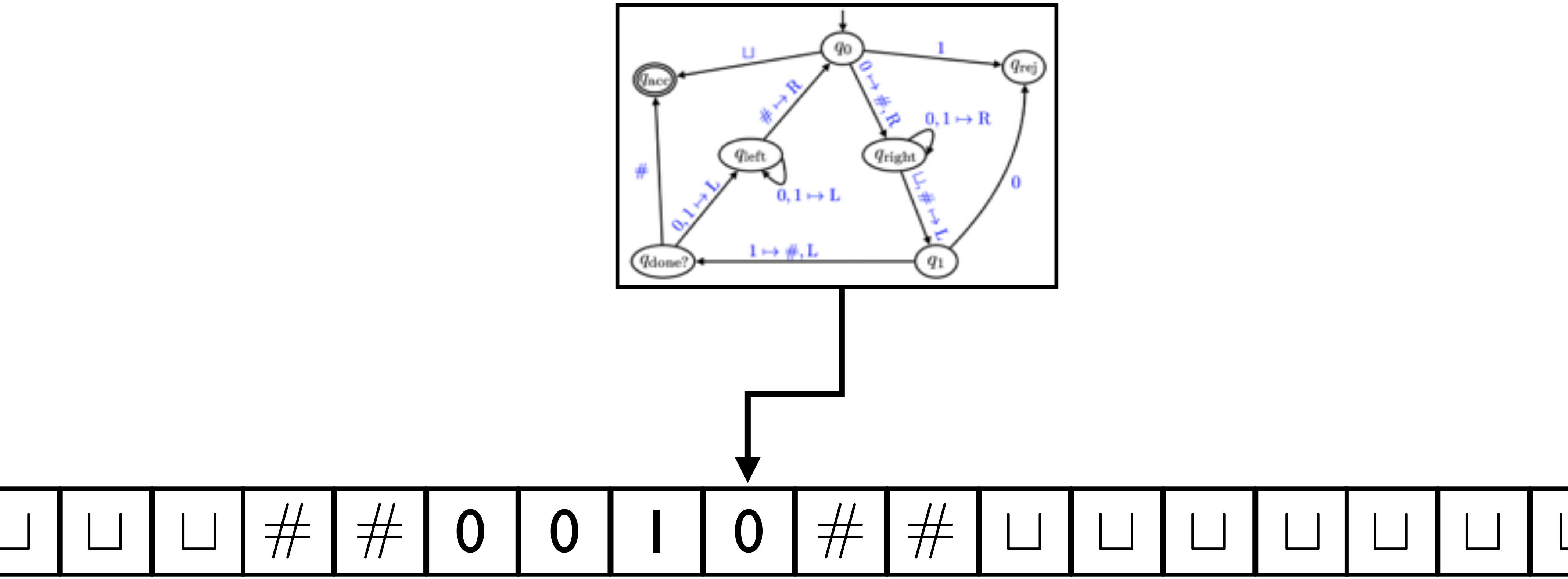
Input: 00001011

Turing machine that decides 0^n1^n



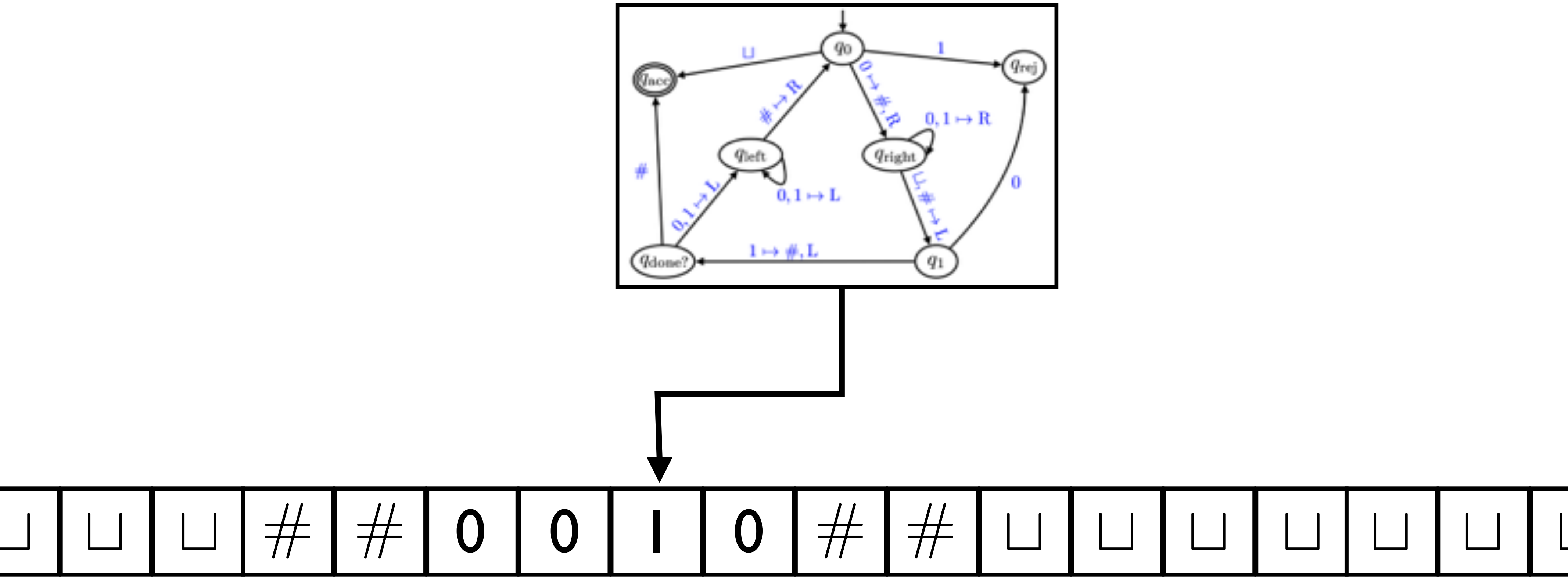
Input: 00001011

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



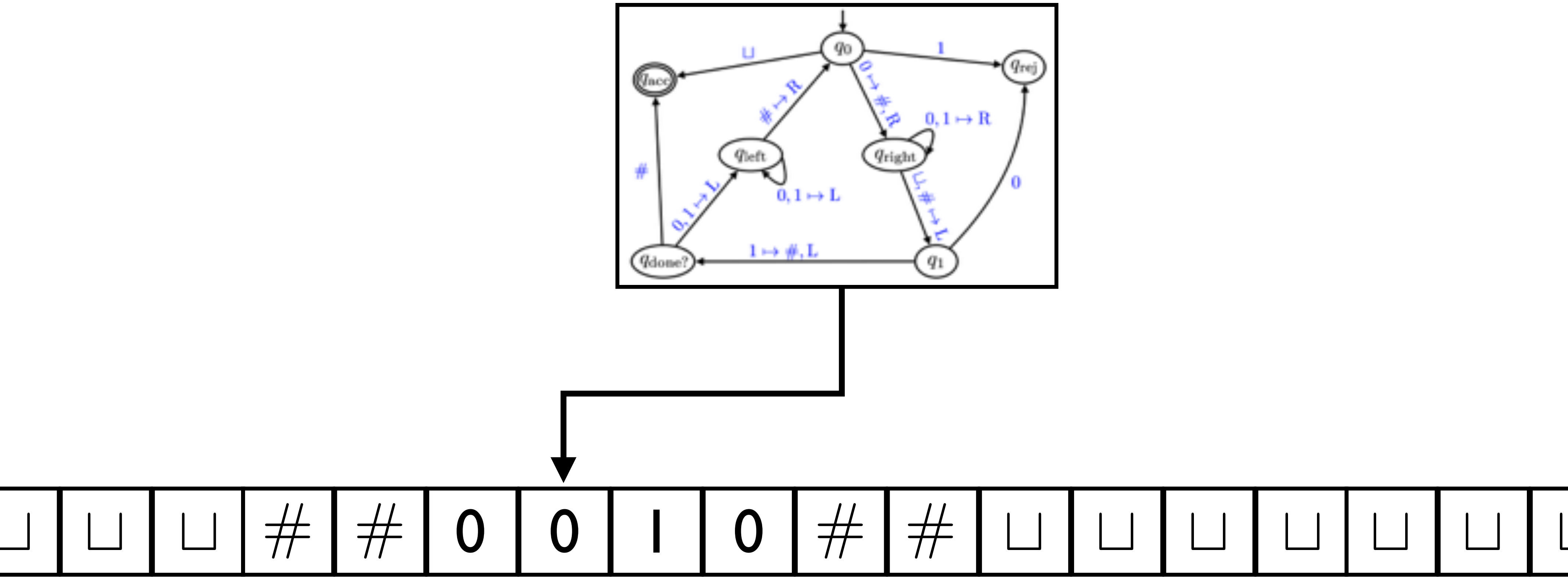
Input: 00001011

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



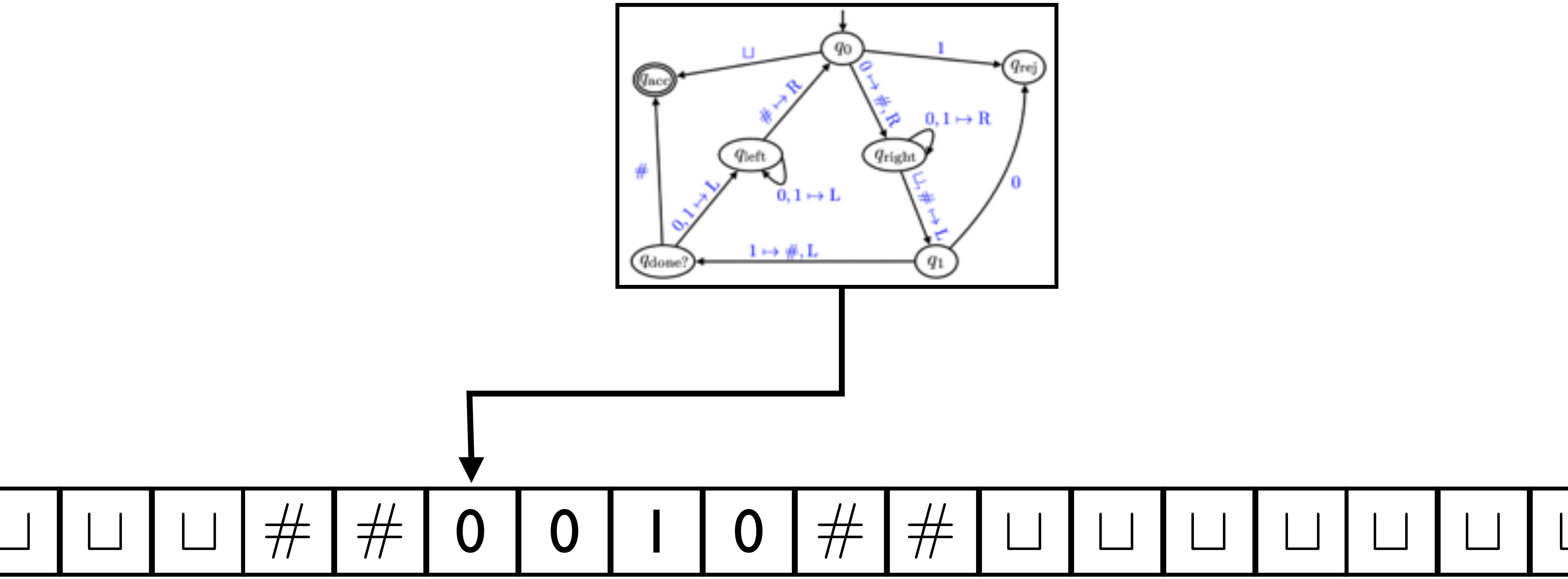
Input: 00001011

Turing machine that decides 0^n1^n



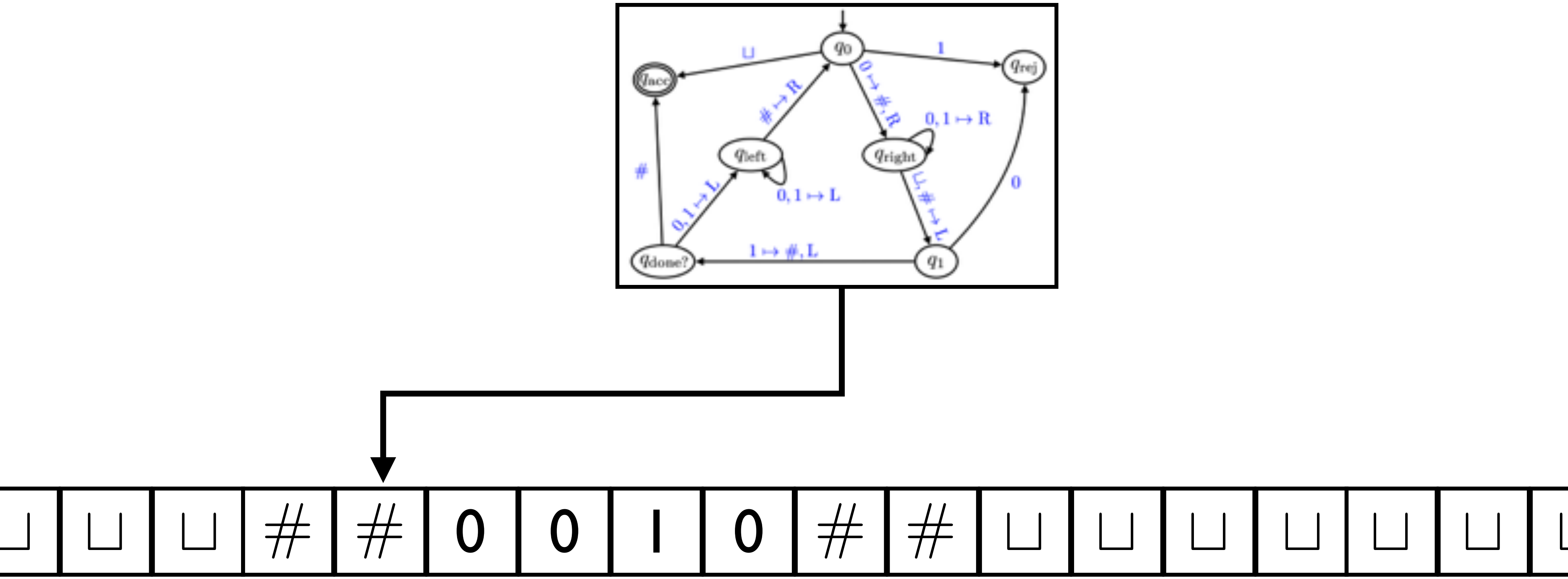
Input: 00001011

Turing machine that decides 0^n1^n



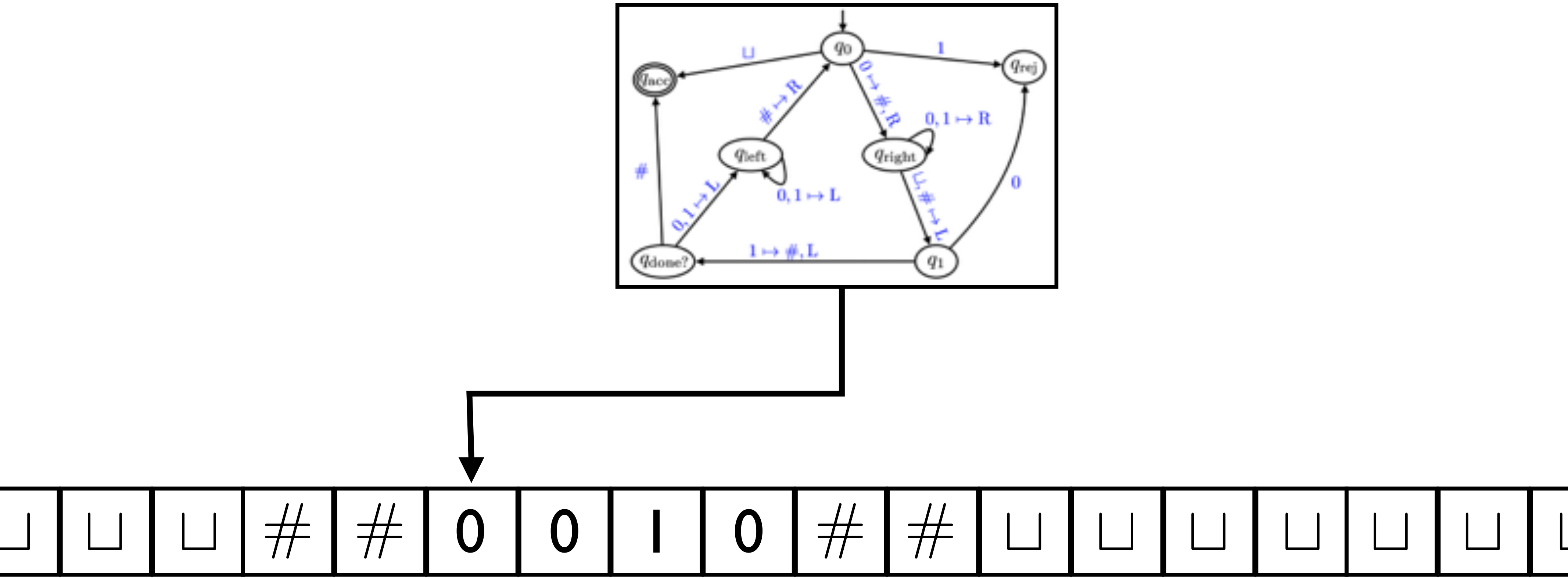
Input: 00001011

Turing machine that decides 0^n1^n



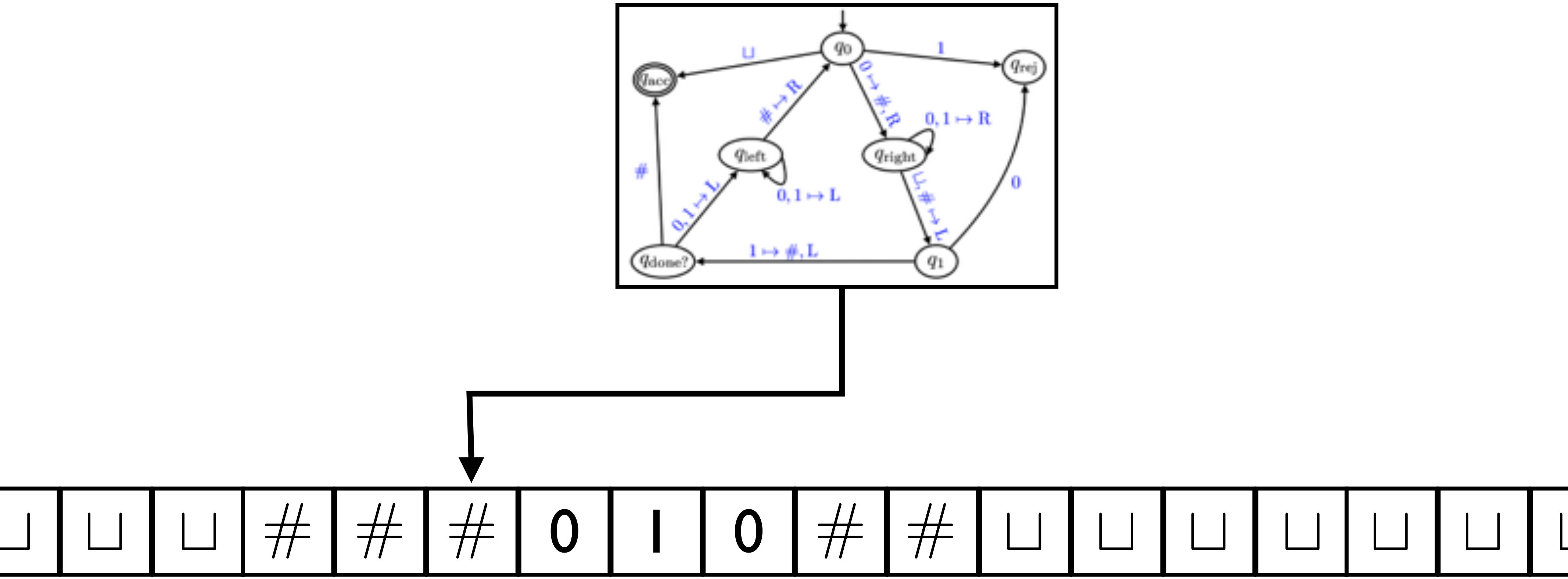
Input: 00001011

Turing machine that decides 0^n1^n



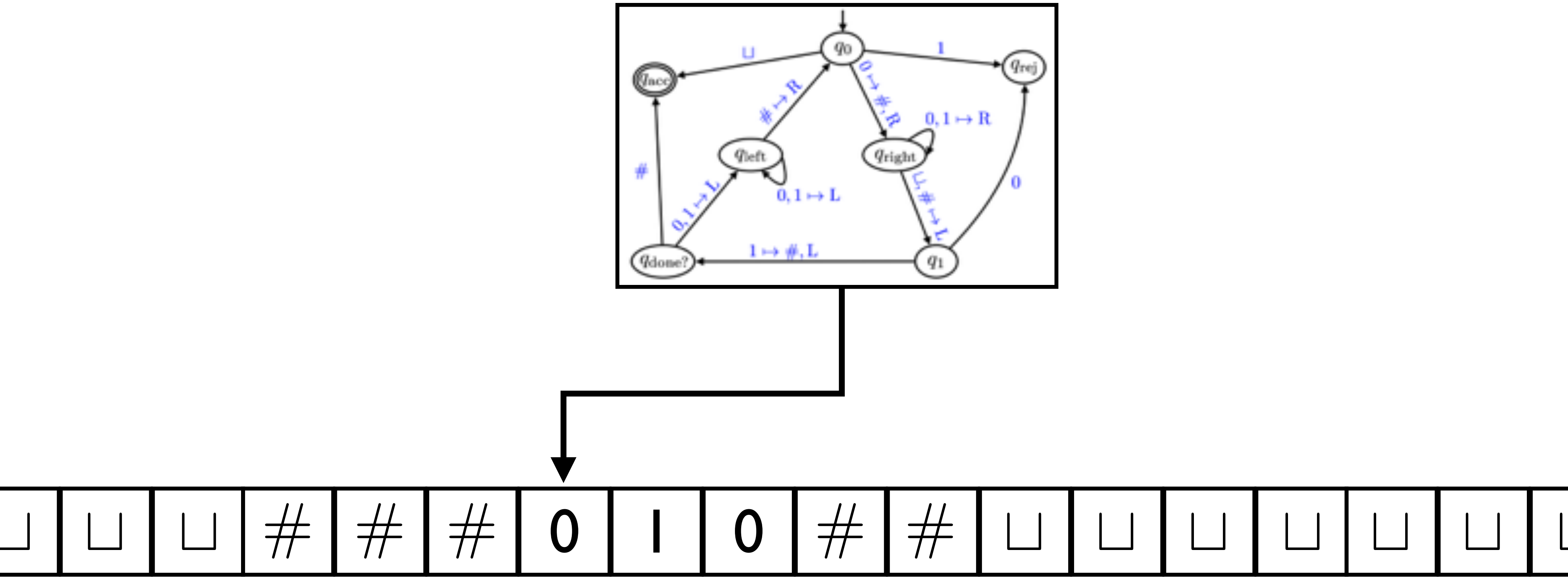
Input: 00001011

Turing machine that decides 0^n1^n



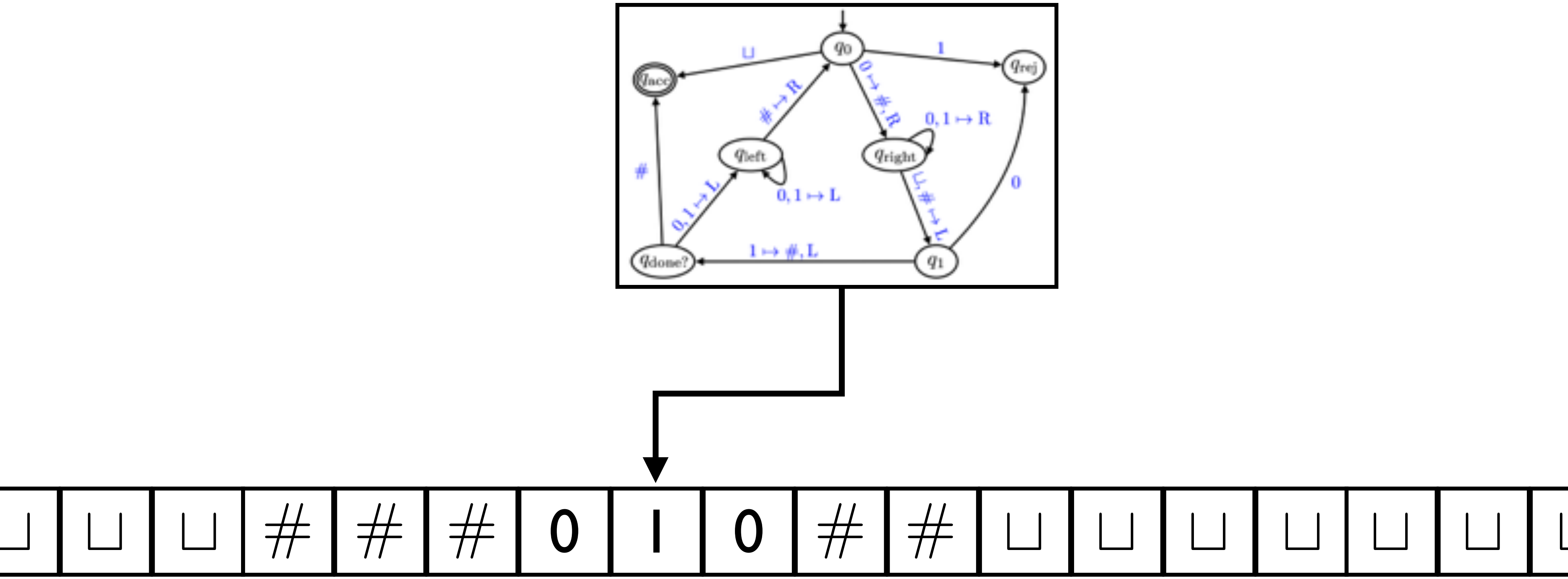
Input: 00001011

Turing machine that decides 0^n1^n



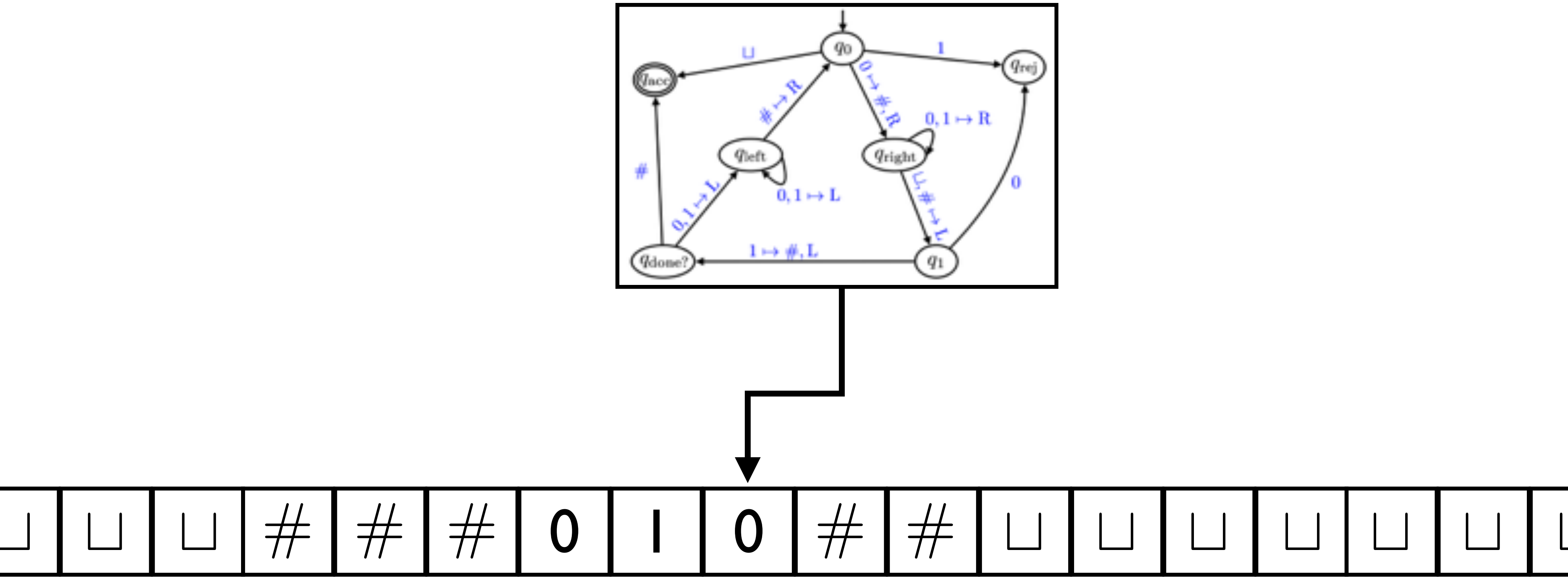
Input: 00001011

Turing machine that decides 0^n1^n



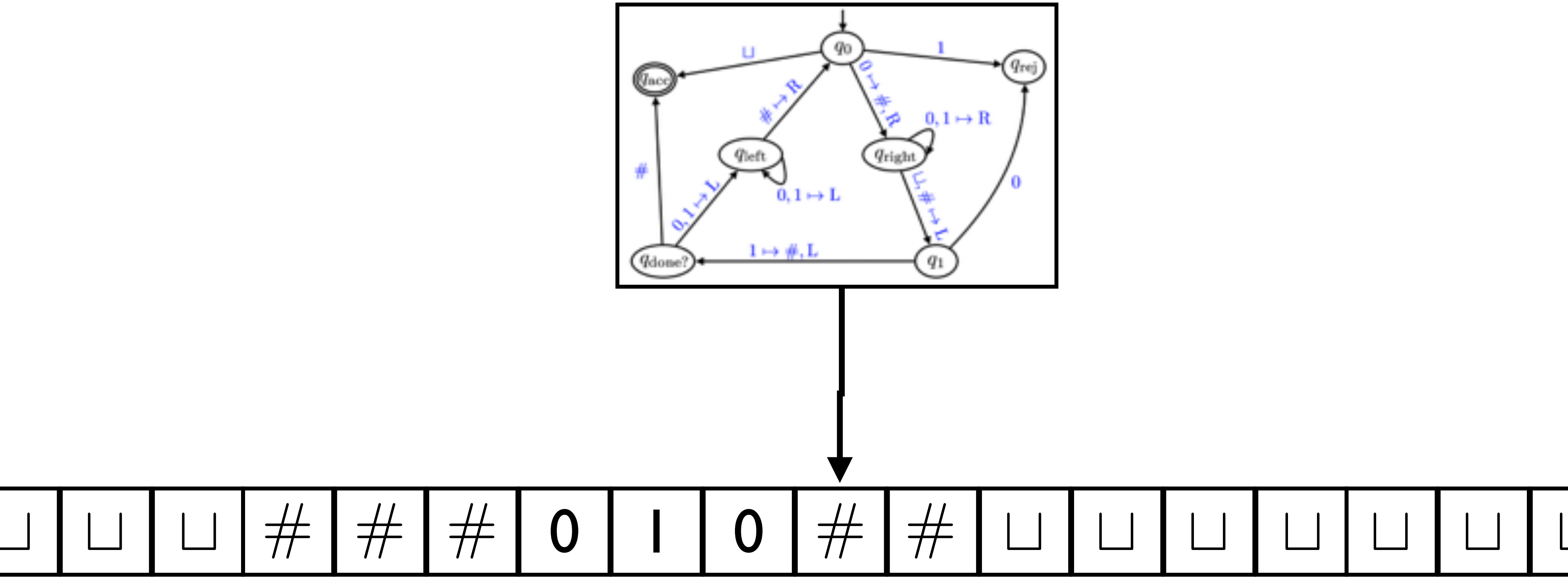
Input: 00001011

Turing machine that decides 0^n1^n



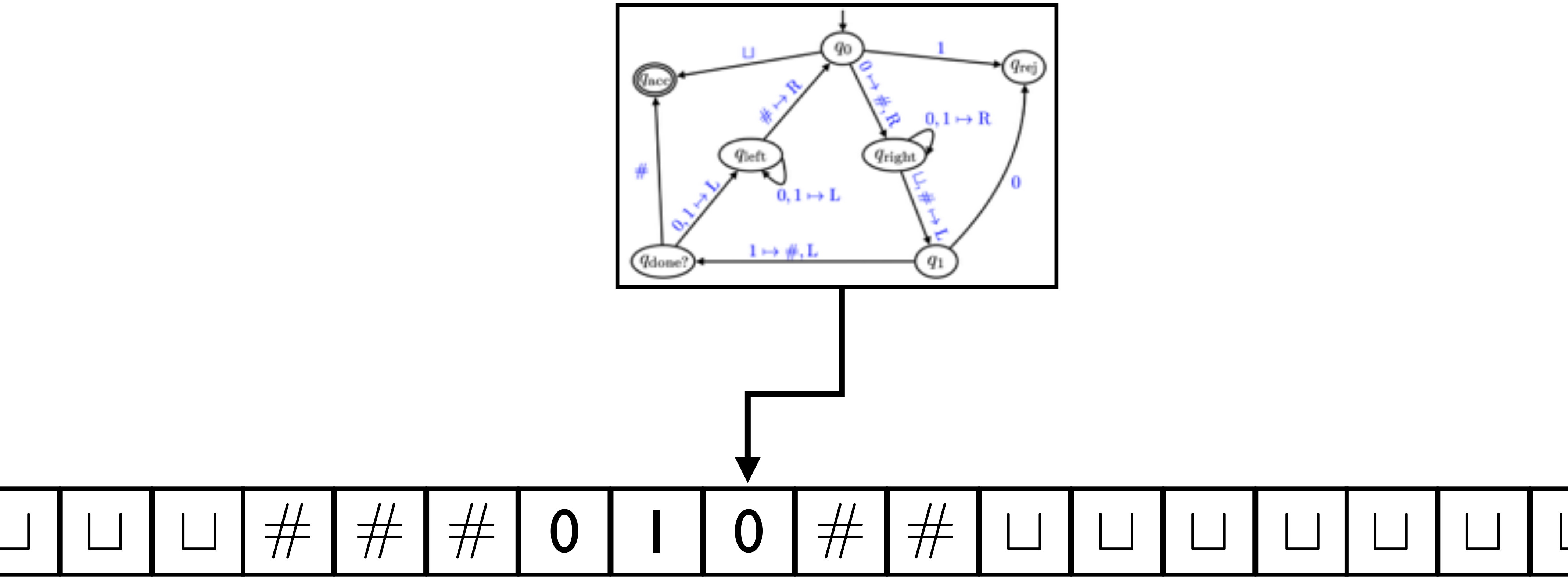
Input: 00001011

Turing machine that decides 0^n1^n



Input: 00001011

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



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 $\sqcup$ encountered

- Shift entire input string one cell to the right

- Convert input from

  $x_1 x_2 x_3 \dots x_n$ to $\sqcup x_1 \sqcup x_2 \sqcup x_3 \dots \sqcup x_n$

- Simulate a big Γ by just $\{0, 1, \sqcup\}$

- "Mark" cells. If $\Gamma = \{0, 1, \sqcup\}$, extend it to

  $$\Gamma = \{0, 1, 0^\bullet, 1^\bullet, \sqcup\}$$

- 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"

$\vdots$

- Simulate assembly language

You could prove this <u>rigorously</u> 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.

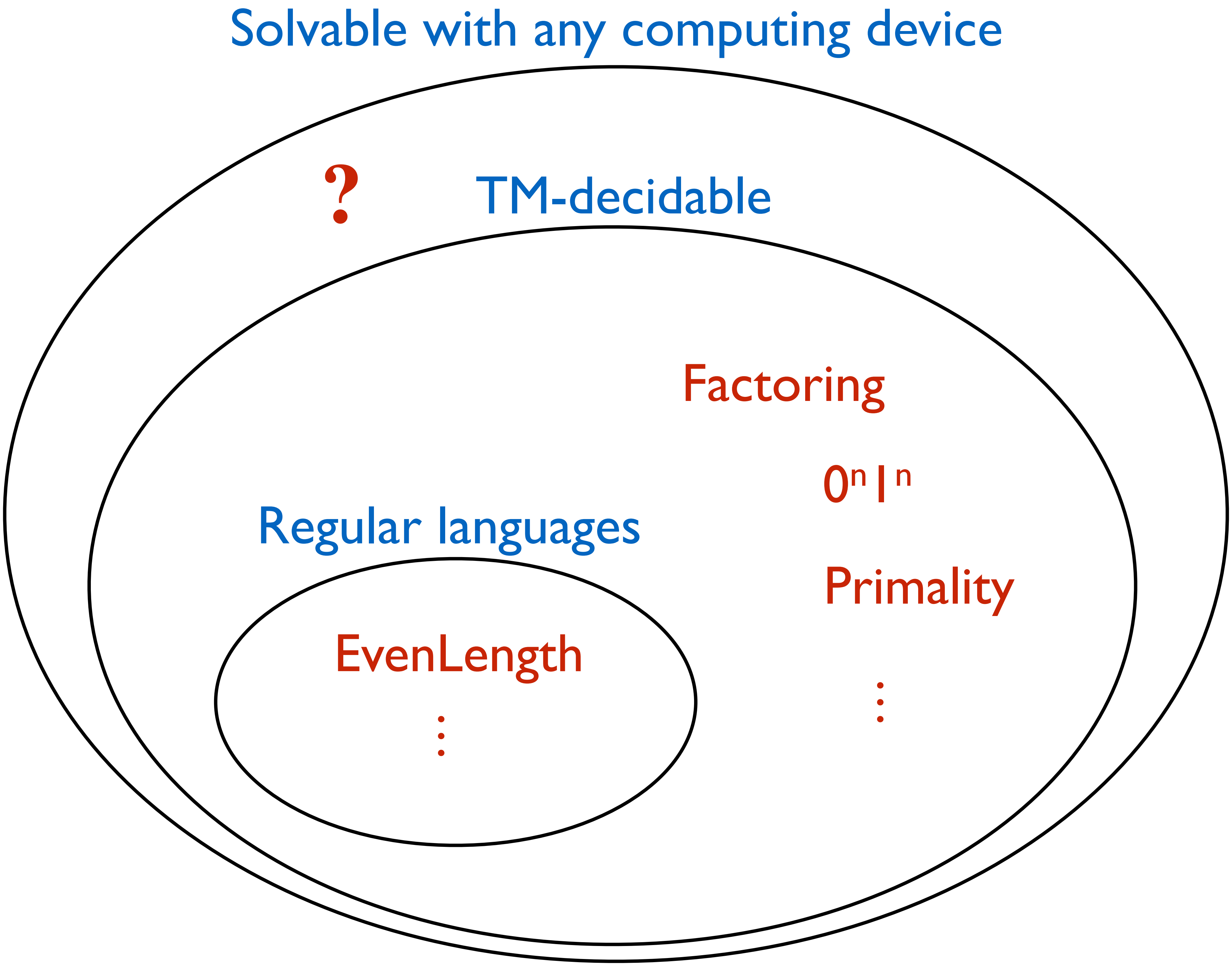
High level description

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
Factoring
$0^n 1^n$
Regular languages
Primality
EvenLength
⋮
⋮

Church-Turing Thesis

Church-Turing Thesis

The intuitive notion of "computable" is captured by functions computable by a Turing Machine.

This is **<u>not</u>** 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


230 A. M. Turing [Nov. 12,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers π, e, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. These results

† Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", *Monatshefte Math. Phys.*, 38 (1931), 173–198.

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

230 A. M. Turing [Nov. 12,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers π, e, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. These results

† Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", *Monatshefte Math. Phys.*, 38 (1931), 173–198.

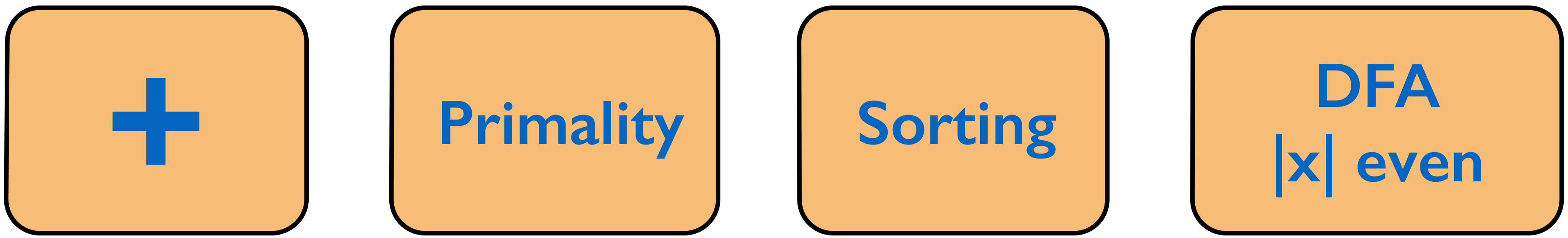
Any notion of "computation" must be able to be carried out by a "computer".

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

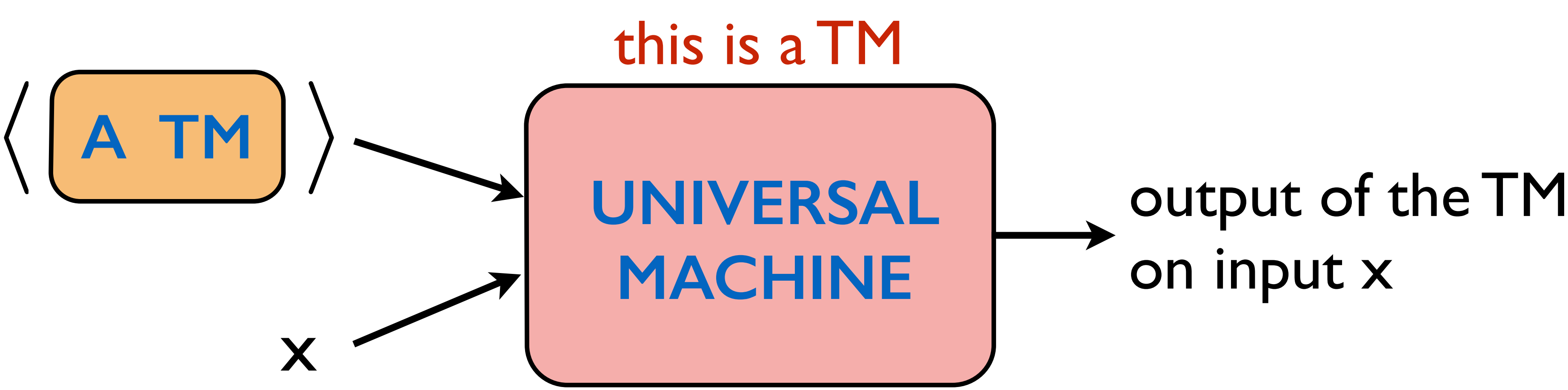
What else did Turing do in his paper?

Universal Machine

(one machine to rule them all)



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

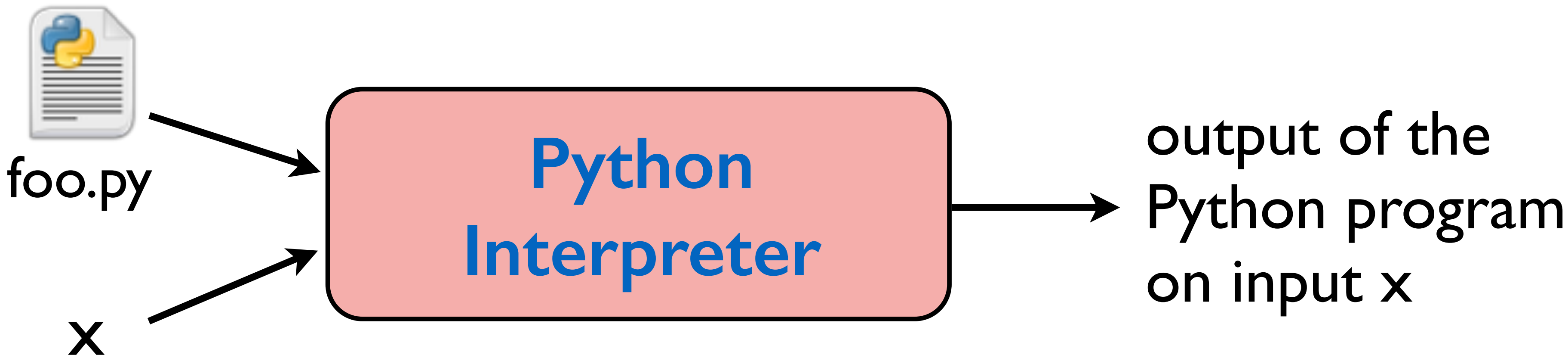


What else did Turing do in his paper?

Universal Machine

(one machine to rule them all)

This is exactly what an interpreter does.

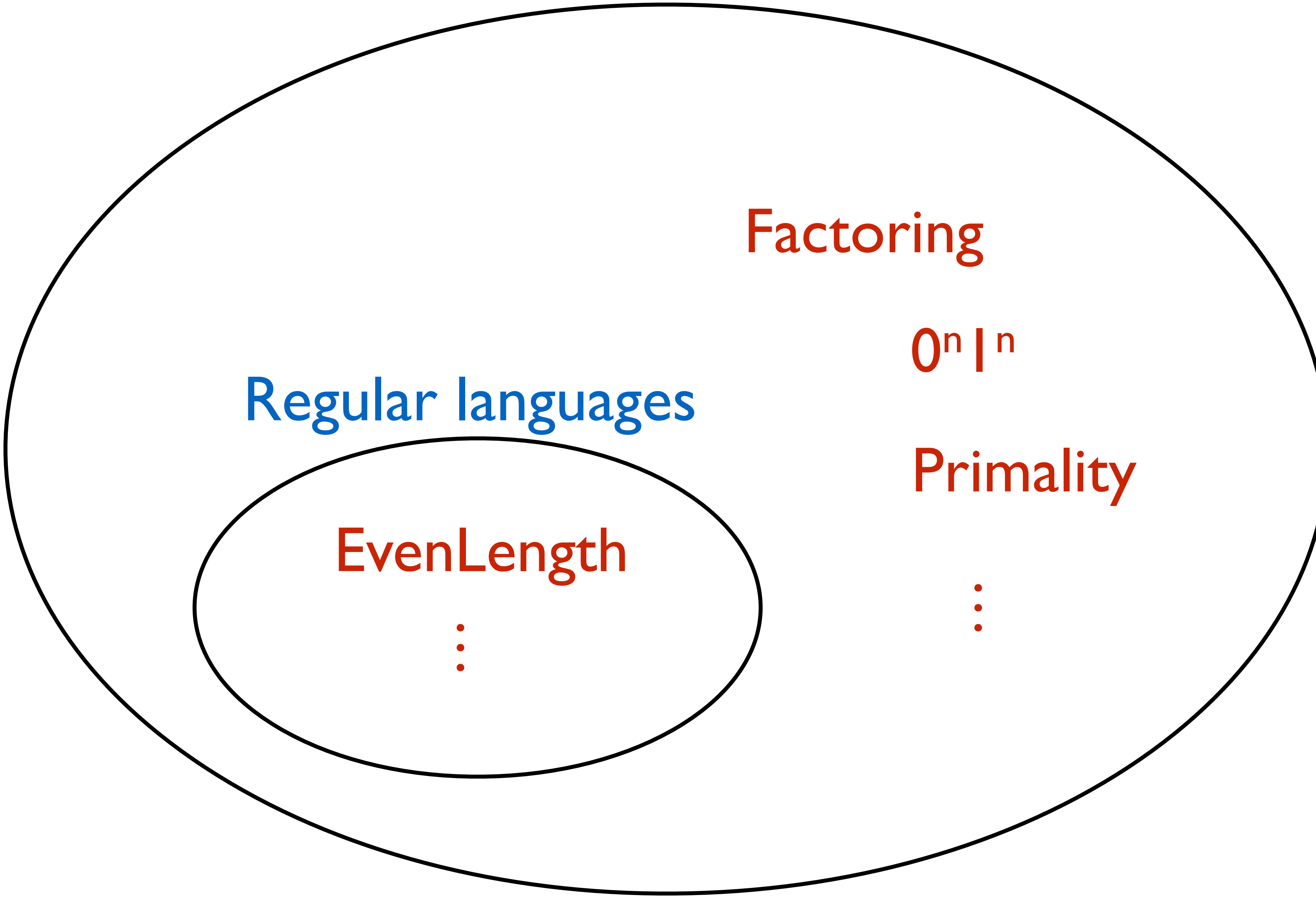


What else did Turing do in his paper?

There are languages that cannot be computed!

?

Solvable with any computing device
=
TM-decidable



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. $\neg \exists x, y, z, n \in \mathbb{N} : (n \geq 3) \wedge (x^n + y^n = z^n)$

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!