Mapping Reductions

Part II

-and-

Complexity Theory

Announcements

- Casual CS Dinner for Women Studying Computer Science is tomorrow night: **Thursday, March 7** at **6PM** in **Gates 219**!
- RSVP through the email link sent out earlier this week.

Announcements

- Problem Set 7 due tomorrow at 12:50PM with a late day.
  - This is a hard deadline – no submissions will be accepted after 12:50PM so that we can release solutions early.

Recap from Last Time

Mapping Reductions

- A function $f : \Sigma_1^* \to \Sigma_2^*$ is called a **mapping reduction** from A to B iff
  - For any $w \in \Sigma_1^*$, $w \in A$ iff $f(w) \in B$.
  - f is a computable function.
- Intuitively, a mapping reduction from A to B says that a computer can transform any instance of A into an instance of B such that the answer to B is the answer to A.

Why Mapping Reducibility Matters

- **Theorem**: If $B \in \mathbf{R}$ and $A \leq_{\mathrm{M}} B$, then $A \in \mathbf{R}$.
- **Theorem**: If $B \in \mathbf{RE}$ and $A \leq_{\mathrm{M}} B$, then $A \in \mathbf{RE}$.
- **Theorem**: If $B \in$ co-$\mathbf{RE}$ and $A \leq_{\mathrm{M}} B$, then $A \in$ co-$\mathbf{RE}$.
- *Intuitively:* $A \leq_{\mathrm{M}} B$ means "A is not harder than B."

Why Mapping Reducibility Matters

- **Theorem**: If $A \notin \mathbf{R}$ and $A \leq_{\mathrm{M}} B$, then $B \notin \mathbf{R}$.
- **Theorem**: If $A \notin \mathbf{RE}$ and $A \leq_{\mathrm{M}} B$, then $B \notin \mathbf{RE}$.
- **Theorem**: If $A \notin \text{co-}\mathbf{RE}$ and $A \leq_{\mathrm{M}} B$, then $B \notin \text{co-}\mathbf{RE}$.
- *Intuitively:* $A \leq_{\mathrm{M}} B$ means "B is at at least as hard as A."

Why Mapping Reducibility Matters

If this one is "easy" (R, RE, co-RE)…

$$A \leq_{\mathrm{M}} B$$

… then this one is "easy" (R, RE, co-RE) too.

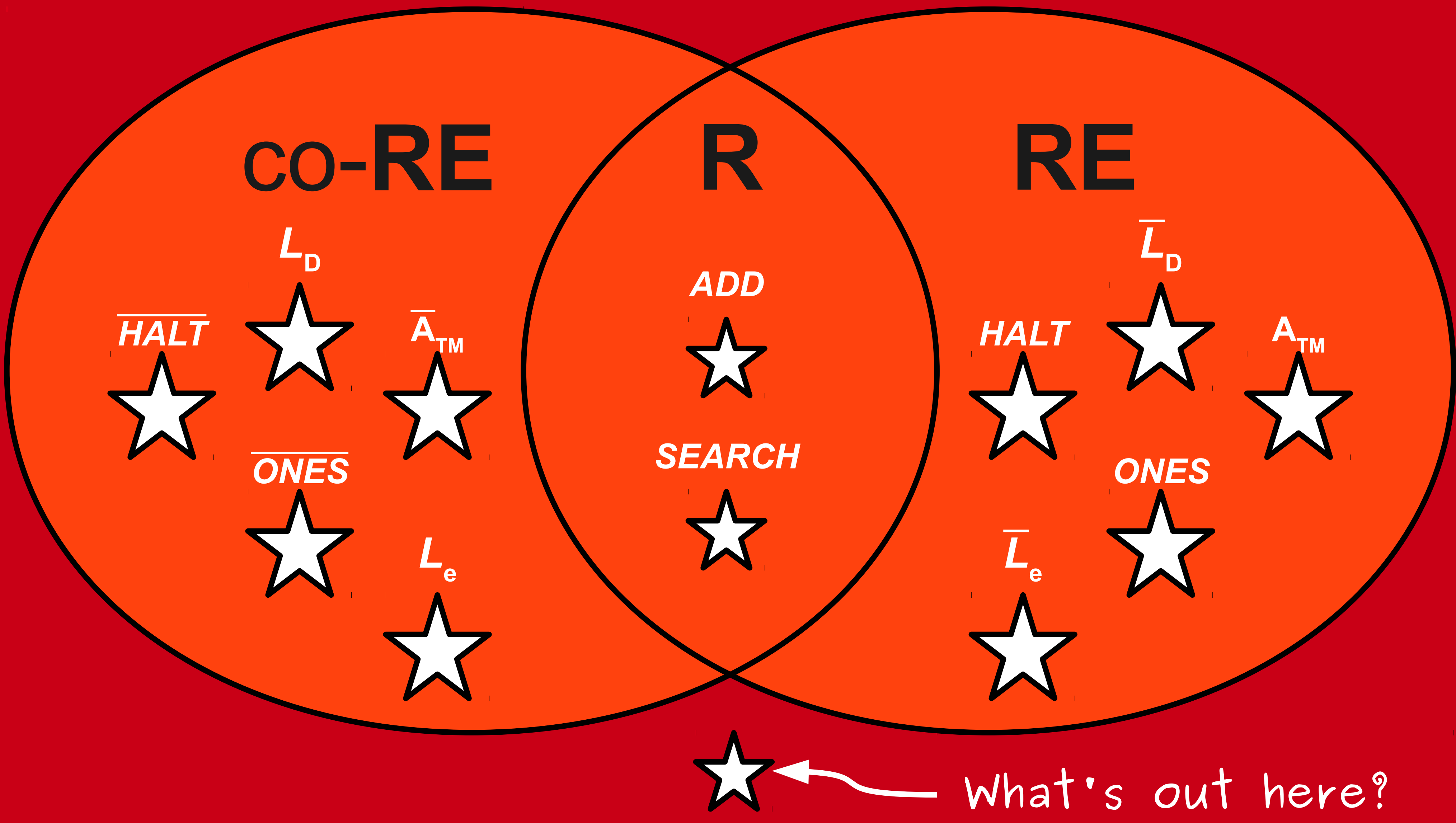
Why Mapping Reducibility Matters

If this one is "hard" (not **R**, not **RE**, or not co-**RE**)…

$$A \leq_{\mathrm{M}} B$$

… then this one is "hard" (not **R**, not **RE**, or not co-**RE**) too.

The Limits of Computability



An Extremely Hard Problem

- Recall: All regular languages are also **RE**.
- This means that some TMs accept regular languages and some TMs do not.
- Let $\text{REGULAR}_{\text{TM}}$ be the language of all TM descriptions that accept regular languages:

$$\mathbf{REGULAR_{TM}} = \{ \langle M \rangle \mid \mathscr{L}(M) \text{ is regular } \}$$

$\text{REGULAR}_{\text{TM}} \notin \mathbf{RE}$

- It turns out that $\text{REGULAR}_{\text{TM}}$ is unrecognizable, meaning that there is no computer program that can confirm that another TM's language is regular!
- To do this, we'll do a reduction from L_{D} and prove that $L_{\text{D}} \leq_{\text{M}} \text{REGULAR}_{\text{TM}}$.

$L_D \leq_M \text{REGULAR}_{TM}$

- We want to find a computable function f such that

$$\langle M \rangle \in L_D \quad \text{iff} \quad f(\langle M \rangle) \in \text{REGULAR}_{TM}.$$

- We need to choose N such that $f(\langle M \rangle) = \langle N \rangle$ for some TM N. Then

$$\langle M \rangle \in L_D \quad \text{iff} \quad f(\langle M \rangle) \in \text{REGULAR}_{TM}$$

$$\langle M \rangle \in L_D \quad \text{iff} \quad \langle N \rangle \in \text{REGULAR}_{TM}$$

$$\langle M \rangle \notin \mathscr{L}(M) \quad \text{iff} \quad \mathscr{L}(N) \text{ is regular.}$$

- Question: How do we pick N?

$L_D \leq_M \text{REGULAR}_{TM}$

- We want to construct some N out of M such that
  - If $\langle M \rangle \in \mathscr{L}(M)$, then $\mathscr{L}(N)$ is not regular.
  - If $\langle M \rangle \notin \mathscr{L}(M)$, then $\mathscr{L}(N)$ is regular.
- One option: choose two languages, one regular and one nonregular, then construct N so its language switches from regular to nonregular based on whether $\langle M \rangle \notin \mathscr{L}(M)$.
  - If $\langle M \rangle \in \mathscr{L}(M)$, then $\mathscr{L}(N) = \{ \mathtt{0}^n\mathtt{1}^n \mid n \in \mathbb{N} \}$
  - If $\langle M \rangle \notin \mathscr{L}(M)$, then $\mathscr{L}(N) = \emptyset$

The Reduction

- We want to build N from M such that
  - If $\langle M \rangle \in \mathscr{L}(M)$, then $\mathscr{L}(N) = \{ \mathtt{0}^n\mathtt{1}^n \mid n \in \mathbb{N} \}$
  - If $\langle M \rangle \notin \mathscr{L}(M)$, then $\mathscr{L}(N) = \emptyset$
- Here is one way to do this:

  N = “On input w:

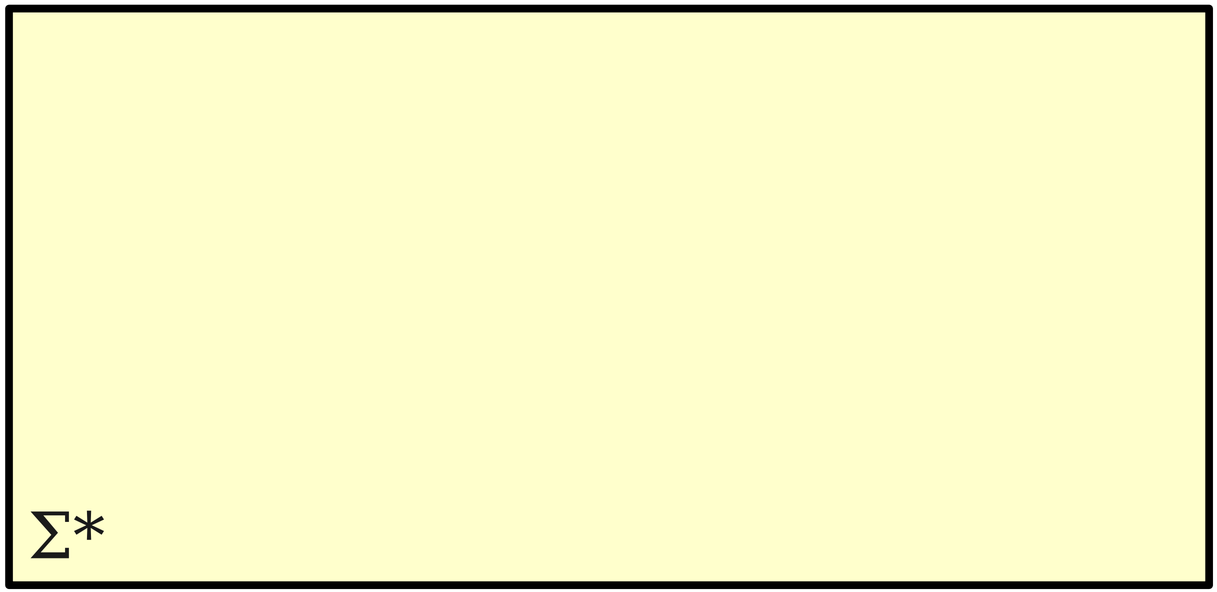
  - If w does not have the form $\mathtt{0}^n\mathtt{1}^n$, then N rejects w.
  - Run M on $\langle M \rangle$.
  - If M accepts $\langle M \rangle$, then N accepts w.
  - If M rejects $\langle M \rangle$, then N rejects w.”

Graphically

N = "On input w:

- If w does not have the form $\texttt{0}^n\texttt{1}^n$, then N rejects w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

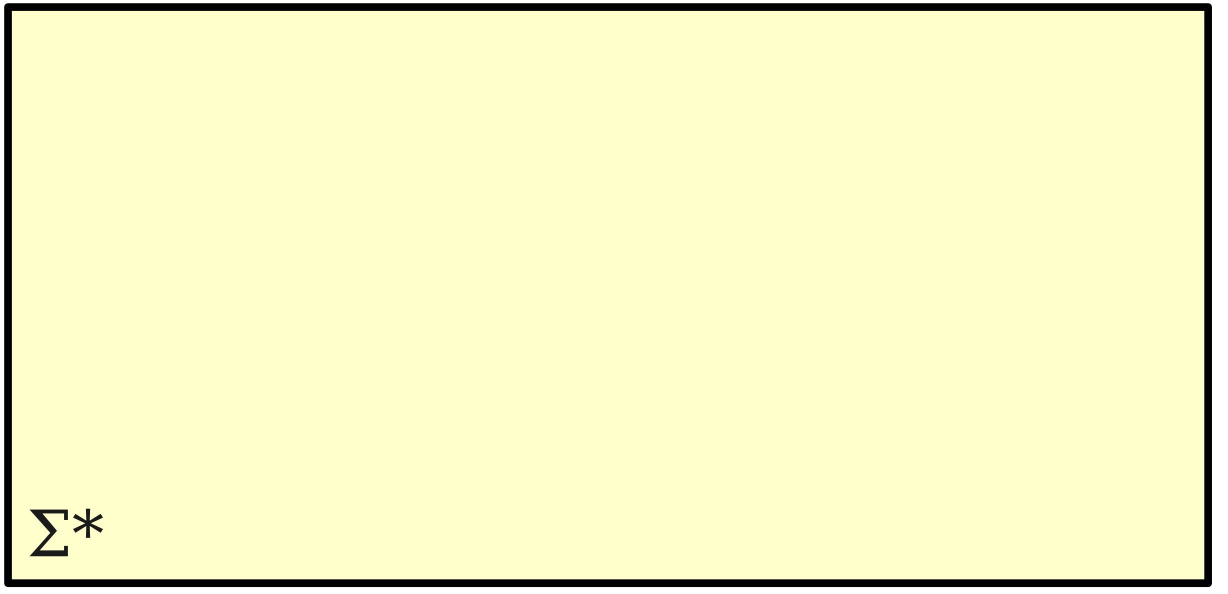
Graphically



N = "On input w:

- If w does not have the form $\mathtt{0}^n\mathtt{1}^n$, then N rejects w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

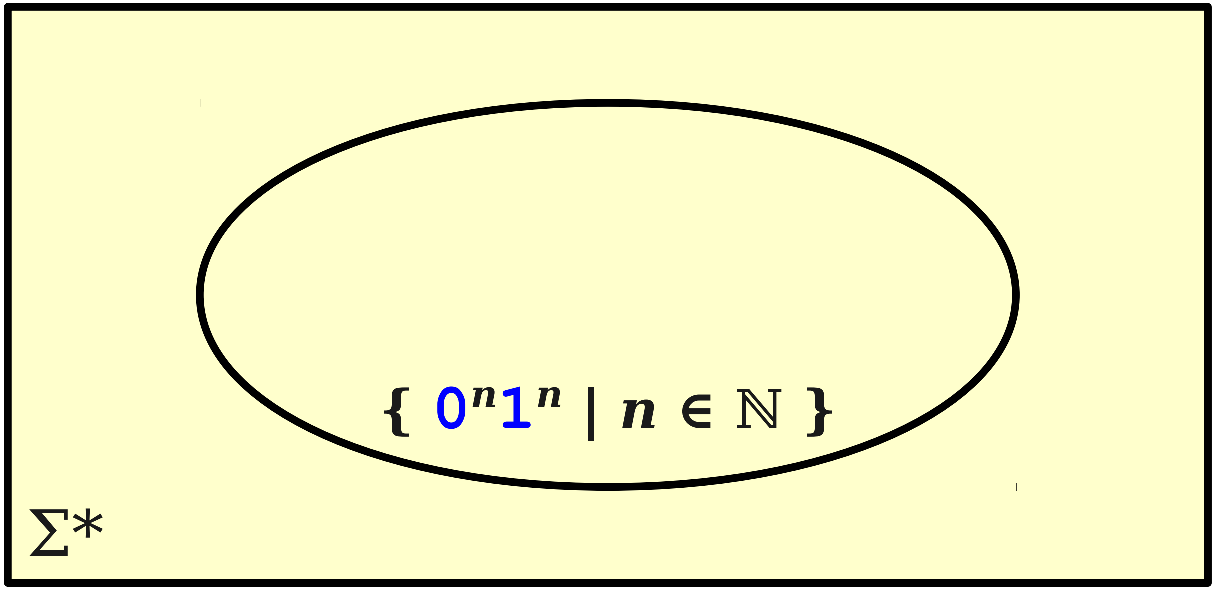
Graphically



N = "On input w:

- If w does not have the form $\texttt{0}^n\texttt{1}^n$, then N rejects w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

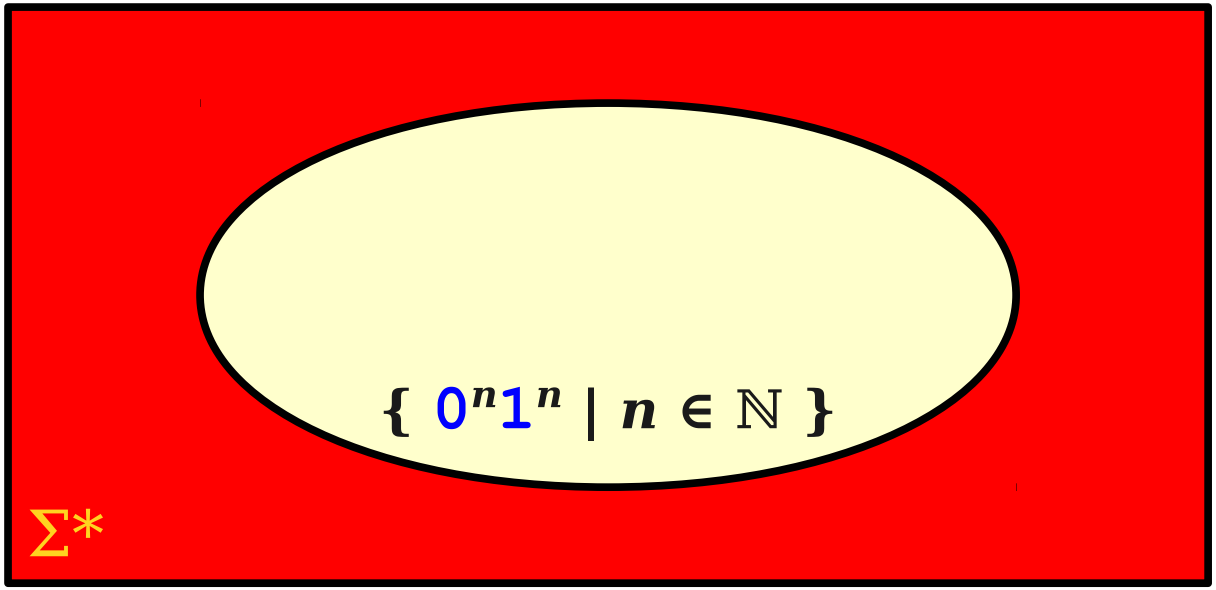
Graphically



N = "On input w:

- $\boxed{\text{If } w \text{ does not have the form } 0^n1^n\text{, then } N \text{ rejects } w.}$
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

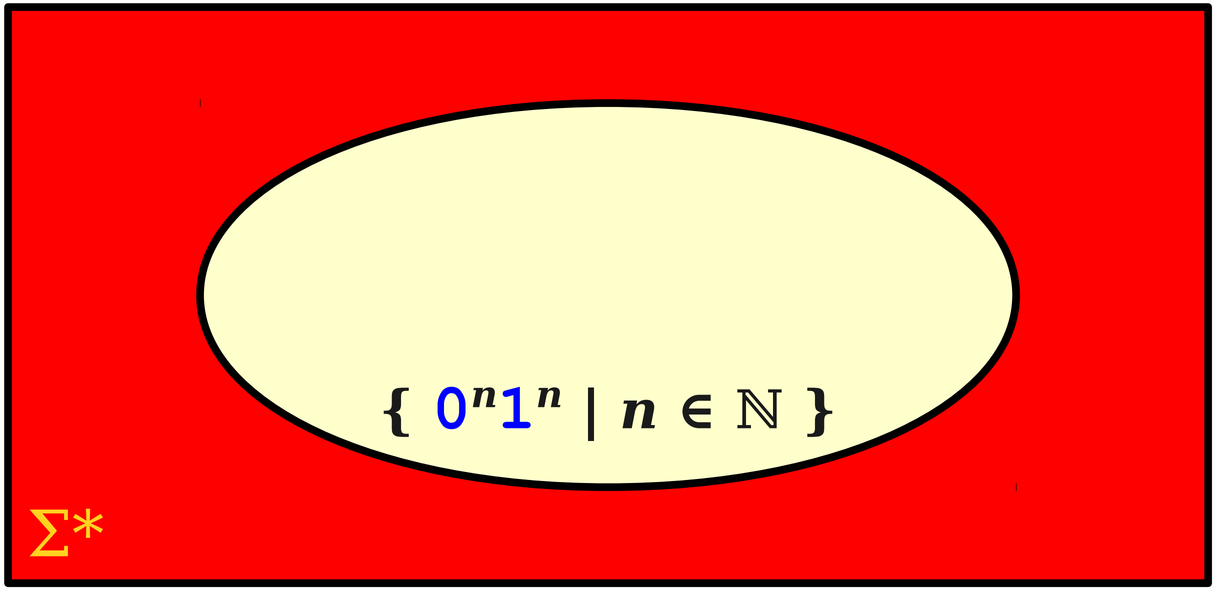
Graphically



N = "On input w:

- If w does not have the form 0^n1^n, then N rejects w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

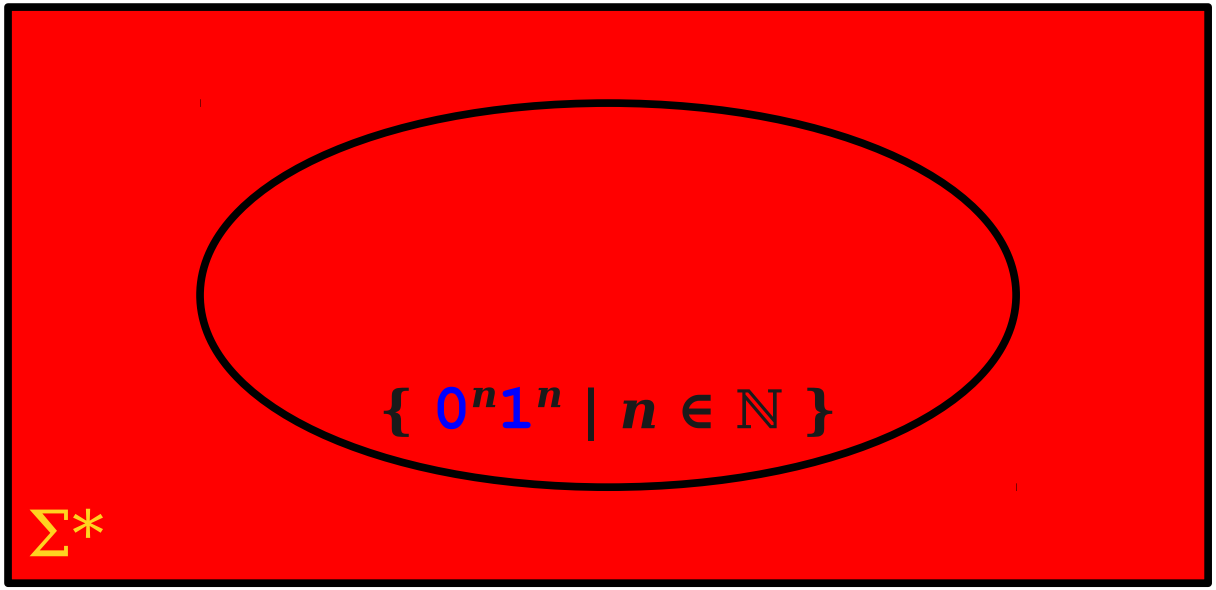
Graphically



N = "On input w:

- If w does not have the form 0^n1^n, then N rejects w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

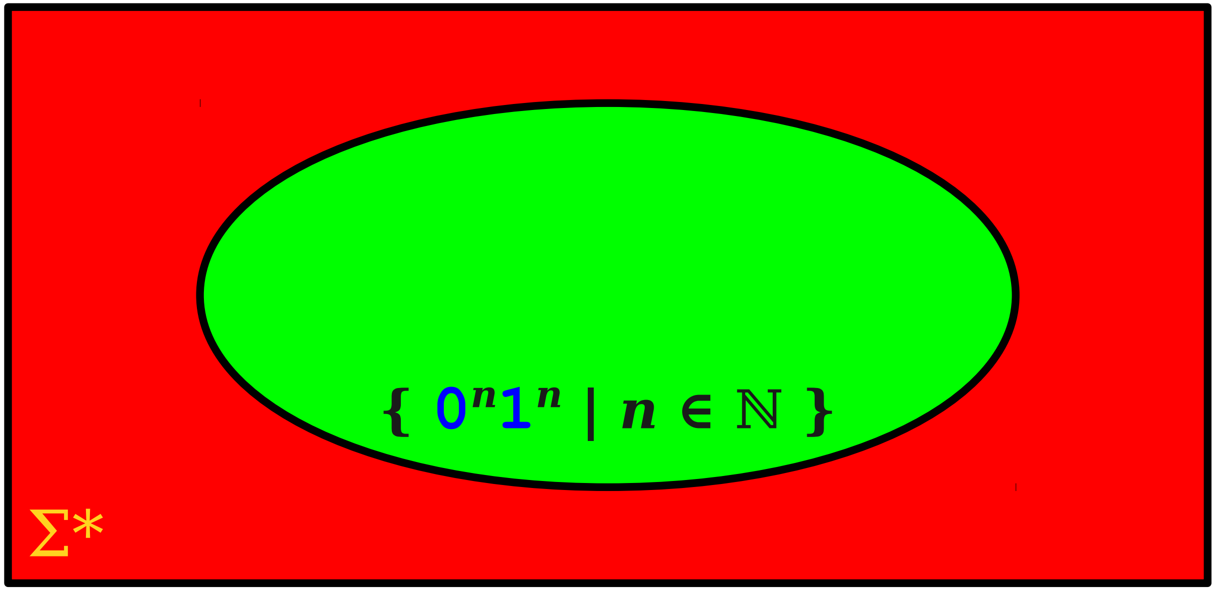
Graphically



N = "On input w:

- If w does not have the form 0^n1^n, then N rejects w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

Graphically



N = "On input w:

- If w does not have the form 0^n1^n, then N rejects w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

Theorem: $L_D \leq_M \mathrm{REGULAR}_{TM}$.

Proof: We exhibit a mapping reduction from L_D to $\mathrm{REGULAR}_{TM}$.

For any TM M, let $f(\langle M\rangle) = \langle N\rangle$, where N is defined in terms of M as follows:

N = "On input w:

If w does not have the form 0^n1^n, N rejects w.

Run M on $\langle M\rangle$.

If M accepts $\langle M\rangle$, N accepts w.

If M rejects $\langle M\rangle$, N rejects w."

We claim f is computable and omit the details from this proof. We further claim that $\langle M\rangle \in L_D$ iff $f(\langle M\rangle) \in \mathrm{REGULAR}_{TM}$. To see this, note that $f(\langle M\rangle) = \langle N\rangle \in \mathrm{REGULAR}_{TM}$ iff $\mathscr{L}(N)$ is regular. We claim that $\mathscr{L}(N)$ is regular iff $\langle M\rangle \notin \mathscr{L}(M)$. To see this, note that if $\langle M\rangle \notin \mathscr{L}(M)$, then N never accepts any strings. Thus $\mathscr{L}(N) = \emptyset$, which is regular. Otherwise, if $\langle M\rangle \in \mathscr{L}(M)$, then N accepts all strings of the form 0^n1^n, so we have that $\mathscr{L}(M) = \{ 0^n1^n \mid n \in \mathbb{N} \}$, which is not regular. Finally, $\langle M\rangle \notin \mathscr{L}(\langle M\rangle)$ iff $\langle M\rangle \in L_D$. Thus $\langle \mathrm{M}\rangle \in L_D$ iff $f(\langle M\rangle) \in \mathrm{REGULAR}_{TM}$, so f is a mapping reduction from L_D to $\mathrm{REGULAR}_{TM}$. Therefore, $L_D \leq_M \mathrm{REGULAR}_{TM}$. ■

$\text{REGULAR}_{\text{TM}} \notin \text{co-}\mathbf{RE}$

- Not only is $\text{REGULAR}_{\text{TM}} \notin \mathbf{RE}$, but $\text{REGULAR}_{\text{TM}} \notin \text{co-}\mathbf{RE}$.
- Before proving this, take a minute to think about just how ridiculously hard this problem is.
  - No computer can confirm that an arbitrary TM has a regular language.
  - No computer can confirm that an arbitrary TM has a nonregular language.
  - This is vastly beyond the limits of what computers could ever hope to solve.

$\overline{L}_D \leq_M \text{REGULAR}_{TM}$

- To prove that REGULAR_{TM} is not co-**RE**, we will prove that $\overline{L}_D \leq_M \text{REGULAR}_{TM}$.
- Since $\overline{L}_D$ is not co-**RE**, this proves that REGULAR_{TM} is not co-**RE** either.
- Goal: Find a function f such that

$$\langle M \rangle \in \overline{L}_D \quad \textbf{iff} \quad f(\langle M \rangle) \in \text{REGULAR}_{TM}$$

- Let $f(\langle M \rangle) = \langle N \rangle$ for some TM N. Then we want

$$\langle M \rangle \in \overline{L}_D \quad \textbf{iff} \quad \langle N \rangle \in \text{REGULAR}_{TM}$$

$$\langle M \rangle \in \mathscr{L}(M) \quad \textbf{iff} \quad \mathscr{L}(N) \textbf{ is regular}$$

- Question: How do we pick machine N?

$\overline{L}_\mathrm{D} \leq_\mathrm{M} \mathrm{REGULAR}_\mathrm{TM}$

- We want to construct some N out of M such that
  - If $\langle M \rangle \in \mathscr{L}(M)$, then $\mathscr{L}(N)$ is regular.
  - If $\langle M \rangle \notin \mathscr{L}(M)$, then $\mathscr{L}(N)$ is not regular.
- One option: choose two languages, one regular and one nonregular, then construct N so its language switches from regular to nonregular based on whether $\langle M \rangle \in \mathscr{L}(M)$.
  - If $\langle M \rangle \in \mathscr{L}(M)$, then $\mathscr{L}(N) = \Sigma^*$.
  - If $\langle M \rangle \notin \mathscr{L}(M)$, then $\mathscr{L}(N) = \{ \mathtt{0}^n \mathtt{1}^n \mid n \in \mathbb{N} \}$

$\overline{L}_{\mathrm{D}} \leq_{\mathrm{M}} \mathrm{REGULAR}_{\mathrm{TM}}$

- We want to build N from M such that
  - If $\langle M \rangle \in \mathscr{L}(M)$, then $\mathscr{L}(N) = \Sigma^*$
  - If $\langle M \rangle \notin \mathscr{L}(M)$, then $\mathscr{L}(N) = \{\ \mathtt{0}^n\mathtt{1}^n \mid n \in \mathbb{N}\ \}$
- Here is one way to do this:

N = "On input w:

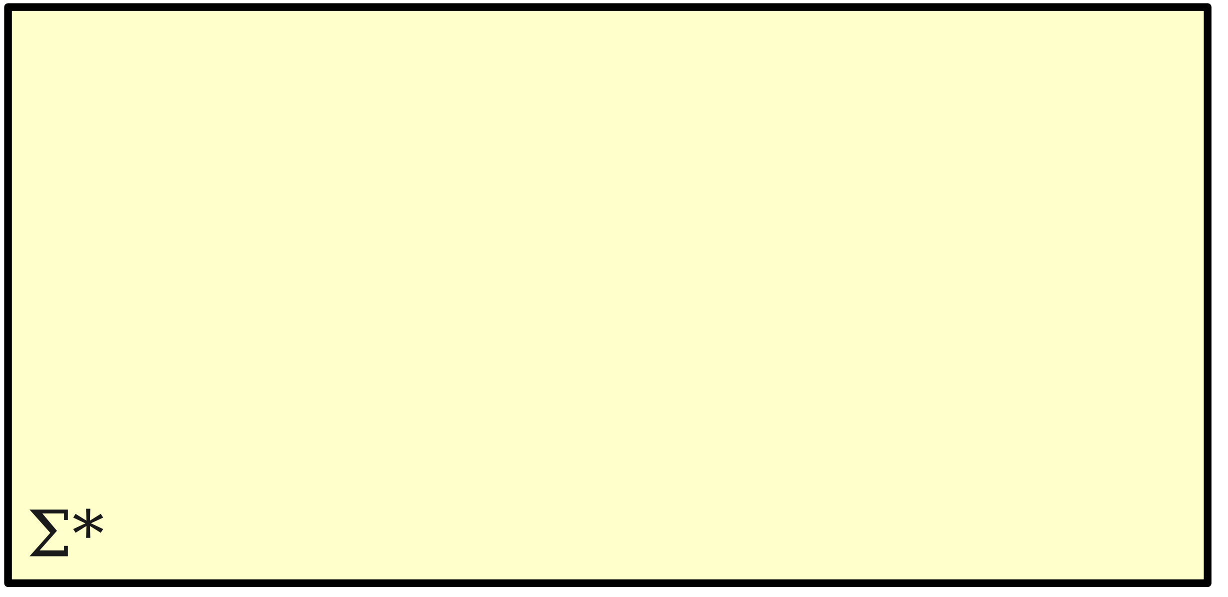
- If w has the form $\mathtt{0}^n\mathtt{1}^n$, then N accepts w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

Graphically

N = "On input w:

- If w has the form $\mathtt{0}^n\mathtt{1}^n$, then N accepts w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

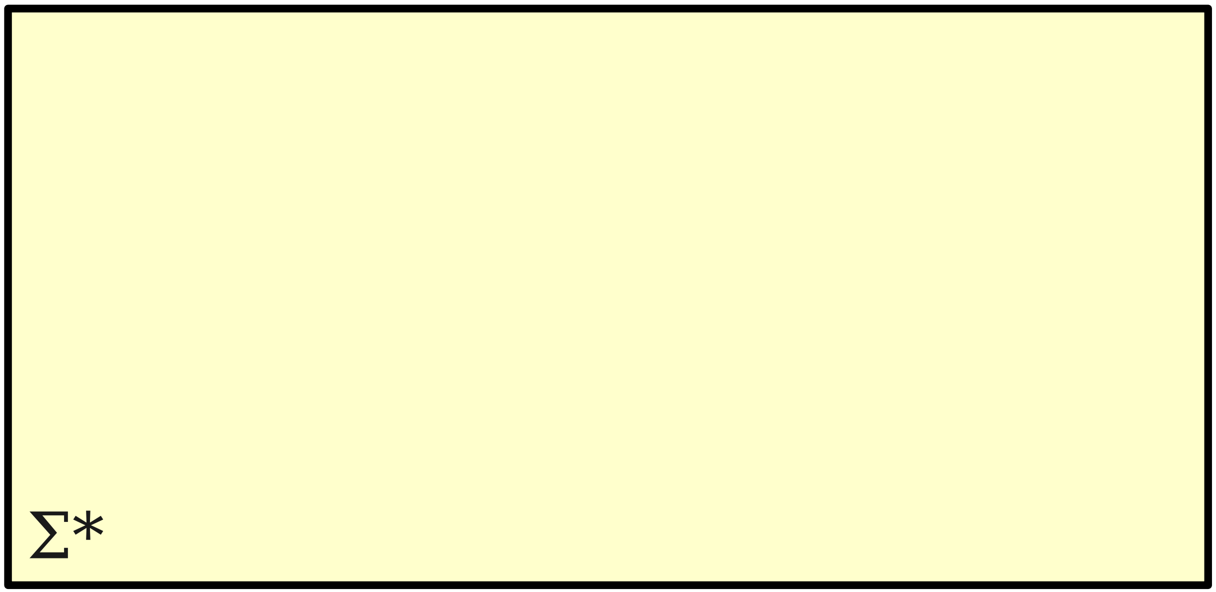
Graphically



N = "On input w:

- If w has the form $\mathtt{0}^n\mathtt{1}^n$, then N accepts w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

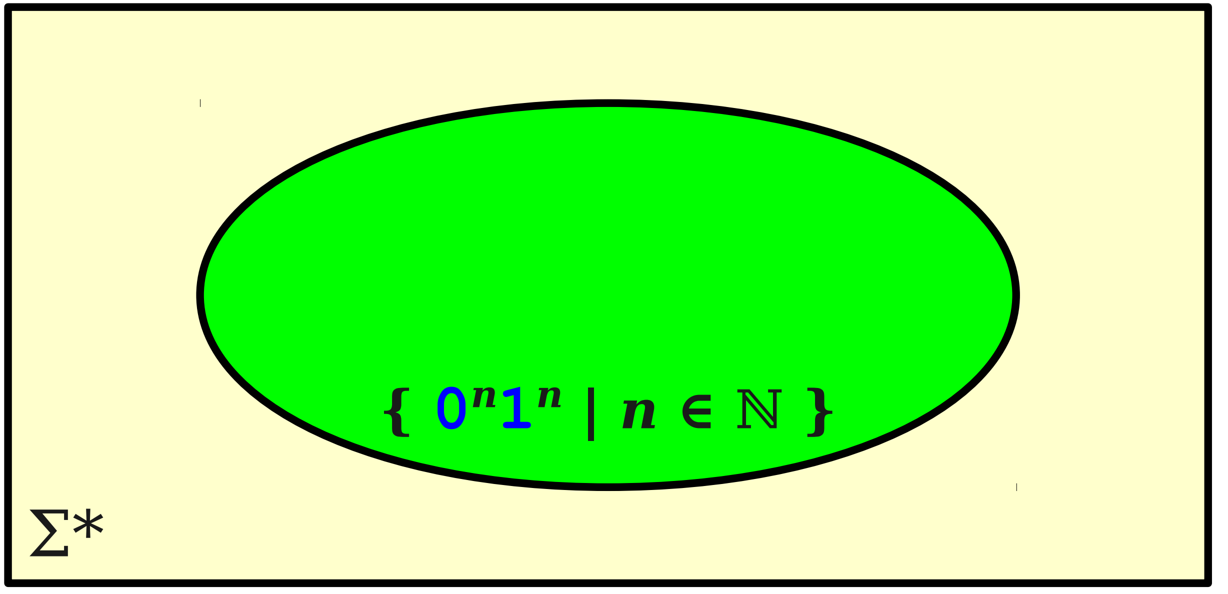
Graphically



N = "On input w:

- If w has the form 0^n1^n, then N accepts w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

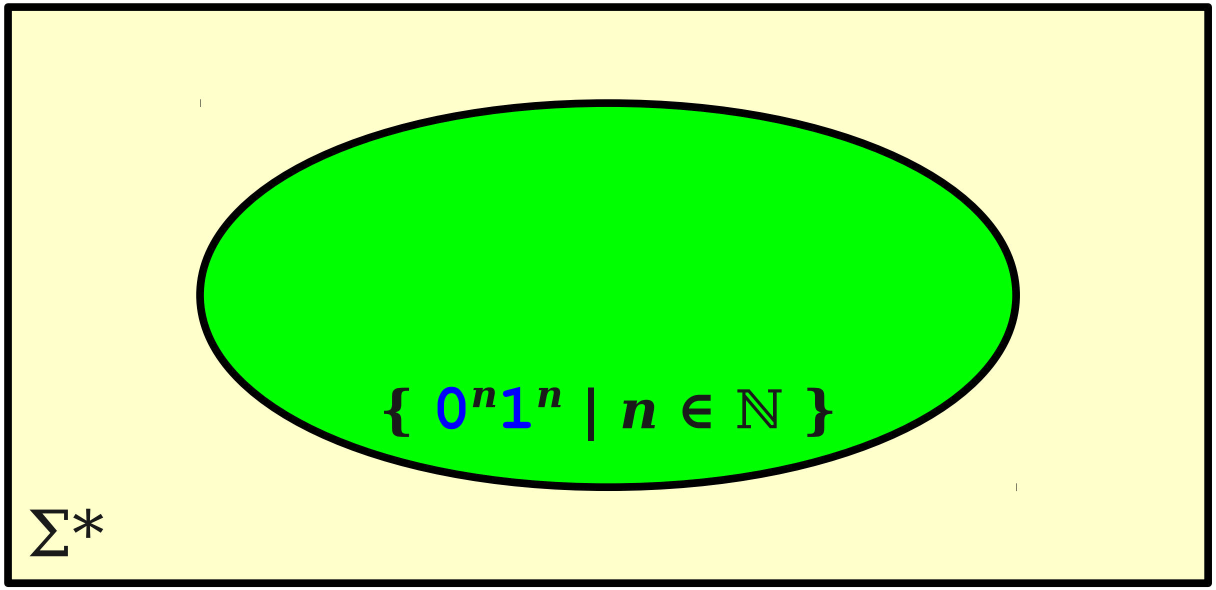
Graphically



N = "On input w:

- If w has the form 0^n1^n, then N accepts w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

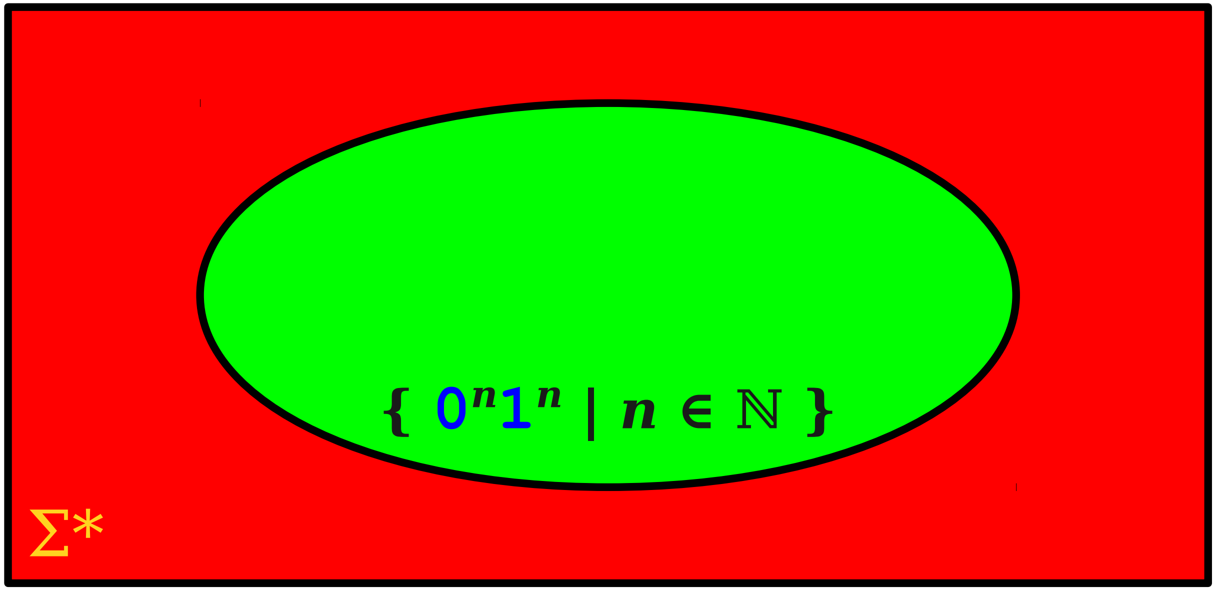
Graphically



N = "On input w:

- If w has the form 0^n1^n, then N accepts w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

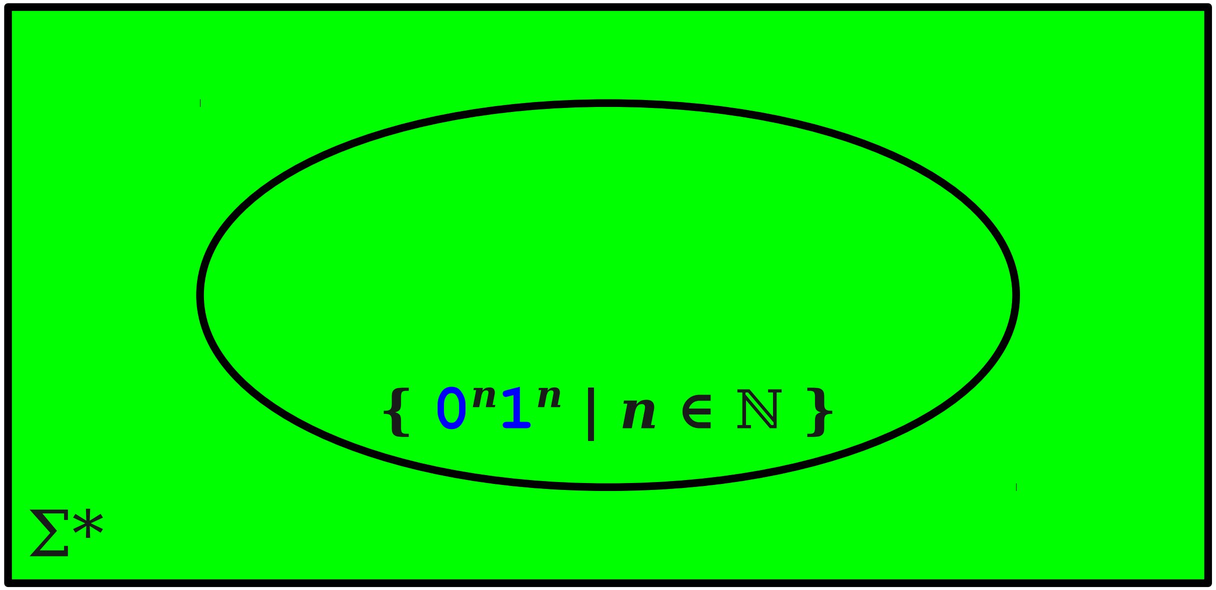
Graphically



N = "On input w:

- If w has the form 0^n1^n, then N accepts w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

Graphically



N = "On input w:

- If w has the form 0^n1^n, then N accepts w.
- Run M on $\langle M \rangle$.
- If M accepts $\langle M \rangle$, then N accepts w.
- If M rejects $\langle M \rangle$, then N rejects w."

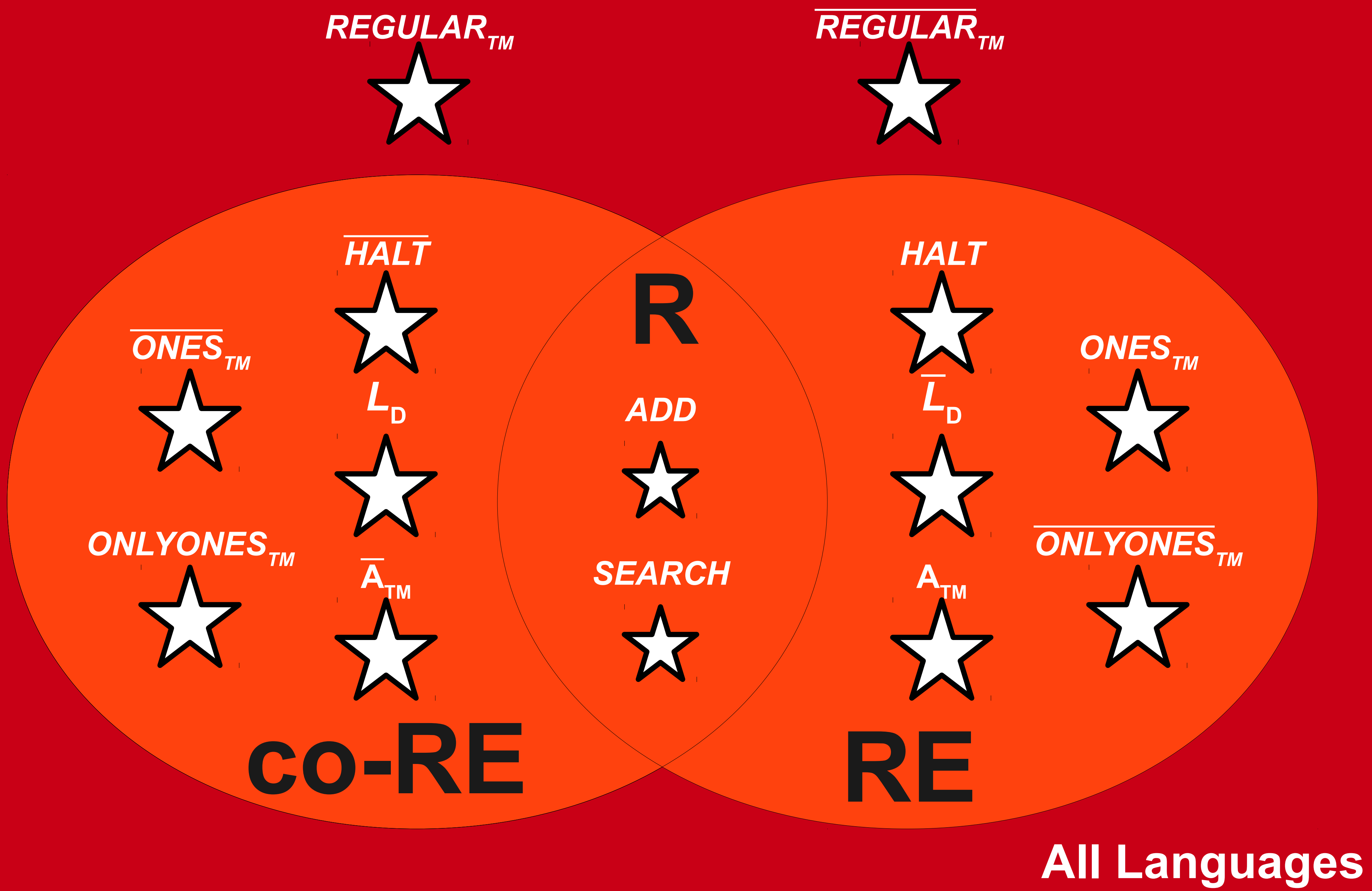
Theorem: $\overline{L}_D \leq_M \text{REGULAR}_{TM}$.

Proof: We exhibit a mapping reduction from $\overline{L}_D$ to REGULAR_{TM}. For any TM M, let $f(\langle M \rangle) = \langle N \rangle$, where N is defined from M as follows:

N = "On input w:
If w has the form 0^n1^n, then N accepts w.
Run M on $\langle M \rangle$.
If M accepts $\langle M \rangle$, then N accepts w.
If M rejects $\langle M \rangle$, then N rejects w."

We state without proof that f is computable. We further claim that $\langle M \rangle \in \overline{L}_D$ iff $f(\langle M \rangle) \in \text{REGULAR}_{TM}$. Note $f(\langle M \rangle) = \langle N \rangle$ and $\langle N \rangle \in \text{REGULAR}_{TM}$ iff $\mathscr{L}(N)$ is regular. We claim that $\mathscr{L}(N)$ is regular iff $\langle M \rangle \in \mathscr{L}(M)$. To see this, note that if $\langle M \rangle \in \mathscr{L}(M)$, then N accepts all strings, either because that string is of the form 0^n1^n or because M eventually accepts $\langle M \rangle$. Thus $\mathscr{L}(N) = \Sigma^*$, which is regular. Otherwise, if $\langle M \rangle \notin \mathscr{L}(M)$, then N only accepts strings of the form 0^n1^n, so $\mathscr{L}(N) = \{ 0^n1^n \mid n \in \mathbb{N} \}$, which is not regular. Finally, $\langle M \rangle \in \mathscr{L}(\langle M \rangle)$ iff $\langle M \rangle \in \overline{L}_D$. Thus $\langle M \rangle \in \overline{L}_D$ iff $f(\langle M \rangle) \in \text{REGULAR}_{TM}$, so f is a mapping reduction from $\overline{L}_D$ to REGULAR_{TM}. Therefore, $\overline{L}_D \leq_M \text{REGULAR}_{TM}$. ■

The Limits of Computability



Why All This Matters

The Limits of Computability

What problems can be solved by a computer?

What problems can be
solved **efficiently** by a computer?

Where We've Been

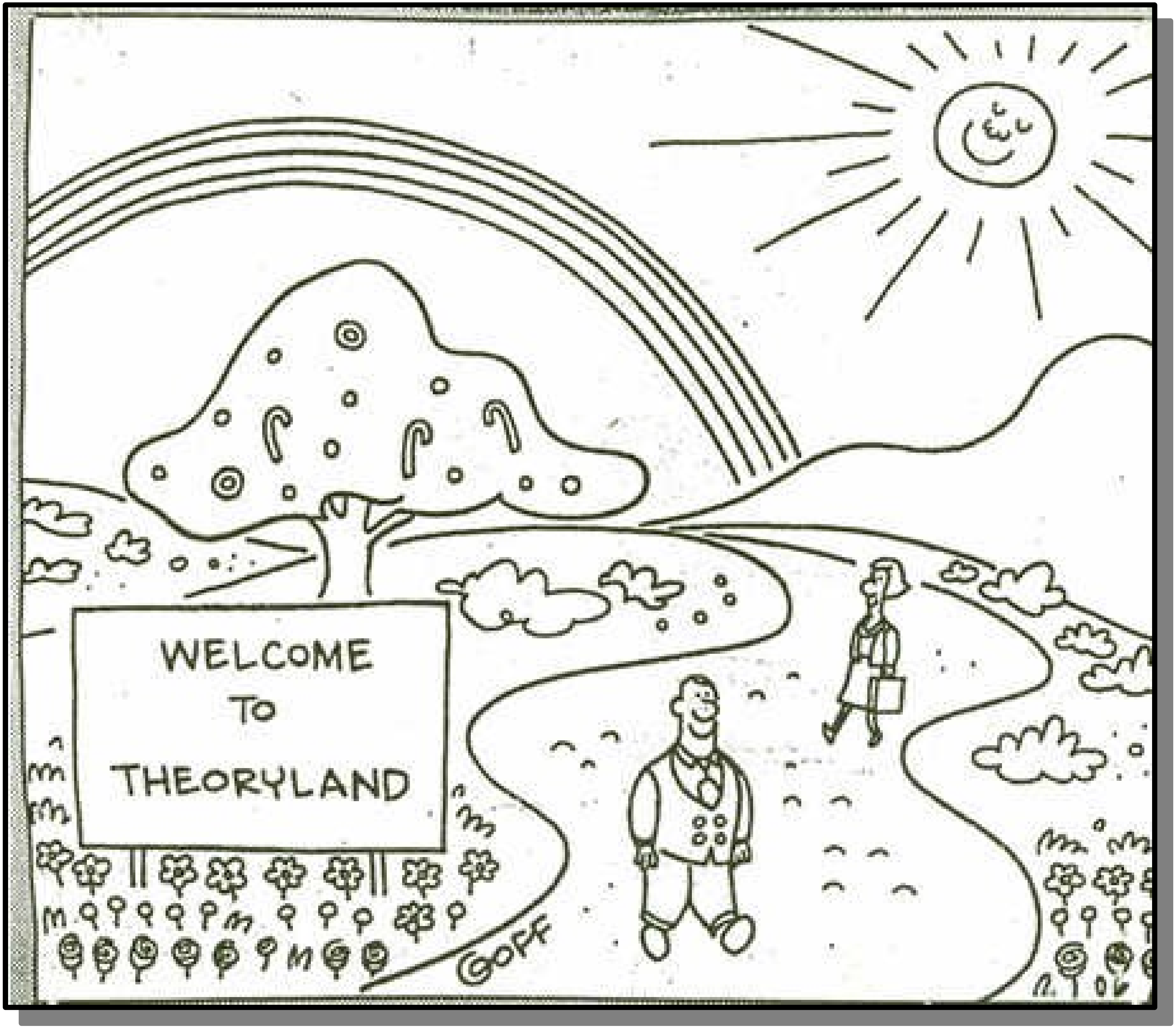
- The class **R** represents problems that can be solved by a computer.
- The class **RE** represents problems where “yes” answers can be verified by a computer.
- The class co-**RE** represents problems where “no” answers can be verified by a computer.
- The mapping reduction can be used to find connections between problems.

Where We're Going

- The class **P** represents problems that can be solved *efficiently* by a computer.
- The class **NP** represents problems where “yes” answers can be verified *efficiently* by a computer.
- The class co-**NP** represents problems where “no” answers can be verified *efficiently* by a computer.
- The *polynomial-time* mapping reduction can be used to find connections between problems.

It may be that since one is customarily concerned with existence, […] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-finite* algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”


WELCOME
TO
THEORYLAND
GOFF

It may be that since one is customarily concerned with existence, […] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-finite* algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”

It may be that since one is customarily concerned with existence, […] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-finite* algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”

It may be that since one is customarily concerned with existence, [...] decidability, and so forth, one is not inclined to take seriously the question of the existence of a *better-than-decidable* algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”

A Decidable Problem

- **Presburger arithmetic** is a logical system for reasoning about arithmetic.
  - $\forall x.\ x + 1 \neq 0$
  - $\forall x.\ \forall y.\ (x + 1 = y + 1 \to x = y)$
  - $\forall x.\ x + 0 = x$
  - $\forall x.\ \forall y.\ (x + y) + 1 = x + (y + 1)$
  - $\forall x.\ ((P(0) \land \forall y.\ (P(y) \to P(y + 1))) \to \forall x.\ P(x)$
- Given a statement, it is decidable whether that statement can be proven from the laws of Presburger arithmetic.
- Any Turing machine that decides whether a statement in Presburger arithmetic is true or false has to move the tape head at least $2^{2^{cn}}$ times on some inputs of length n (for some fixed constant c).

For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$

For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$
$$2^{2^1} = 4$$

For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$
$$2^{2^1} = 4$$
$$2^{2^2} = 16$$

For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$
$$2^{2^1} = 4$$
$$2^{2^2} = 16$$
$$2^{2^3} = 256$$

For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$

$$2^{2^1} = 4$$

$$2^{2^2} = 16$$

$$2^{2^3} = 256$$

$$2^{2^4} = 65536$$

For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$

$$2^{2^1} = 4$$

$$2^{2^2} = 16$$

$$2^{2^3} = 256$$

$$2^{2^4} = 65536$$

$$2^{2^5} = 18446744073709551616$$

For Reference

- Assume $c = 1$.

$$2^{2^0} = 2$$

$$2^{2^1} = 4$$

$$2^{2^2} = 16$$

$$2^{2^3} = 256$$

$$2^{2^4} = 65536$$

$$2^{2^5} = 18446744073709551616$$

$$2^{2^6} = 340282366920938463463374607431768211456$$

The Limits of Decidability

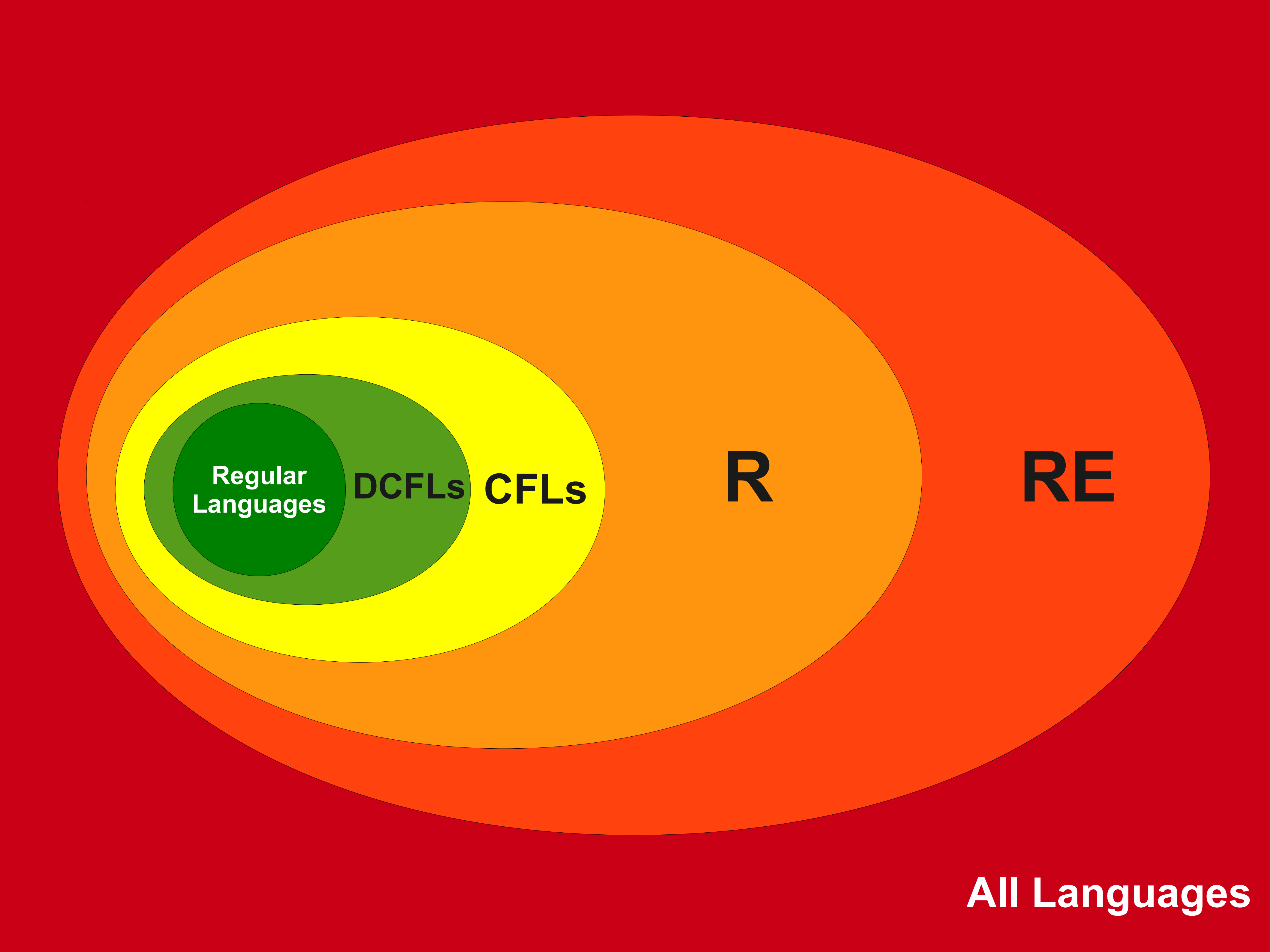
- The fact that a problem is decidable does not mean that it is *feasibly* decidable.
- In **computability theory**, we ask the question

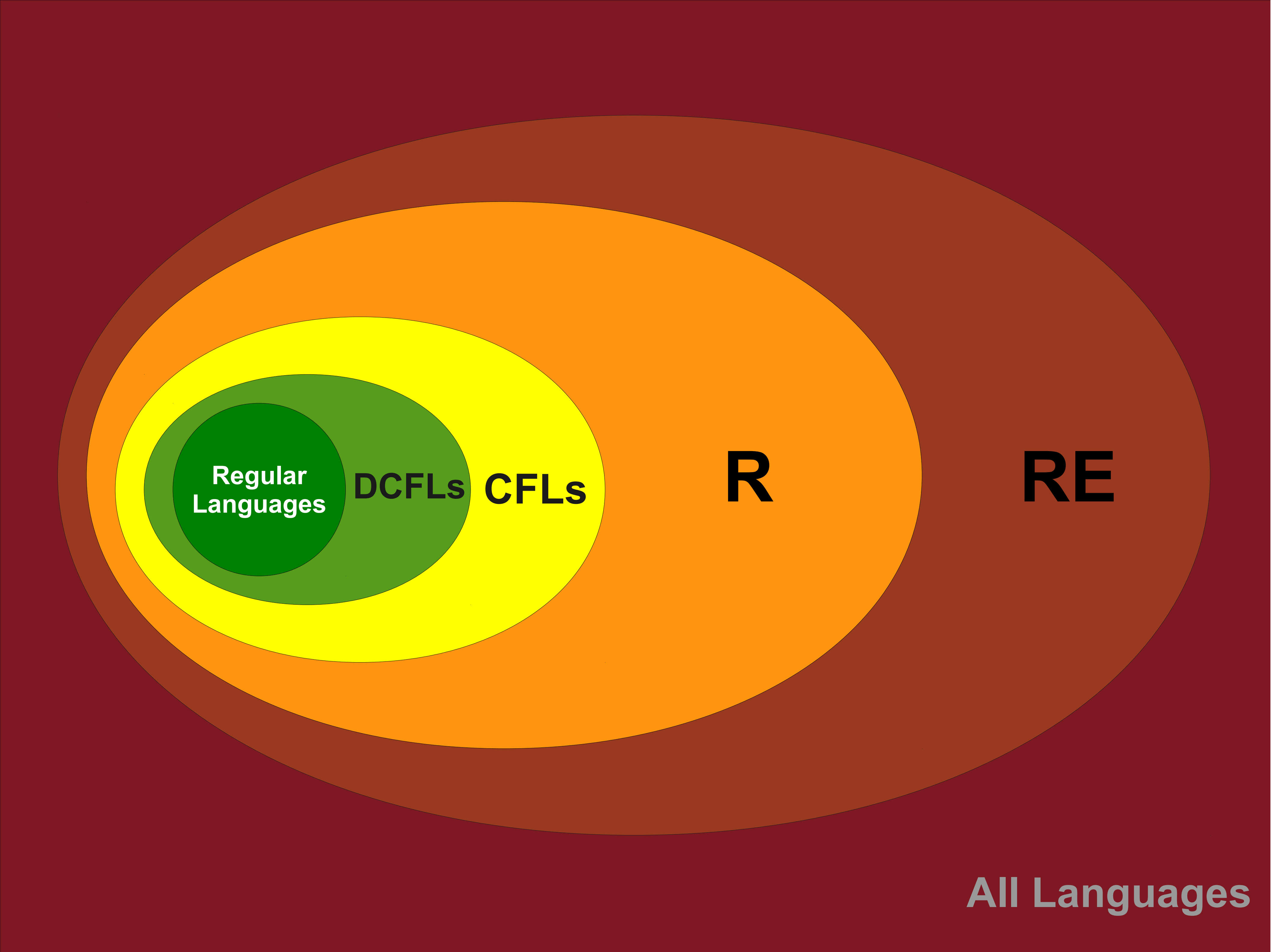
Is it **possible** to solve problem L?

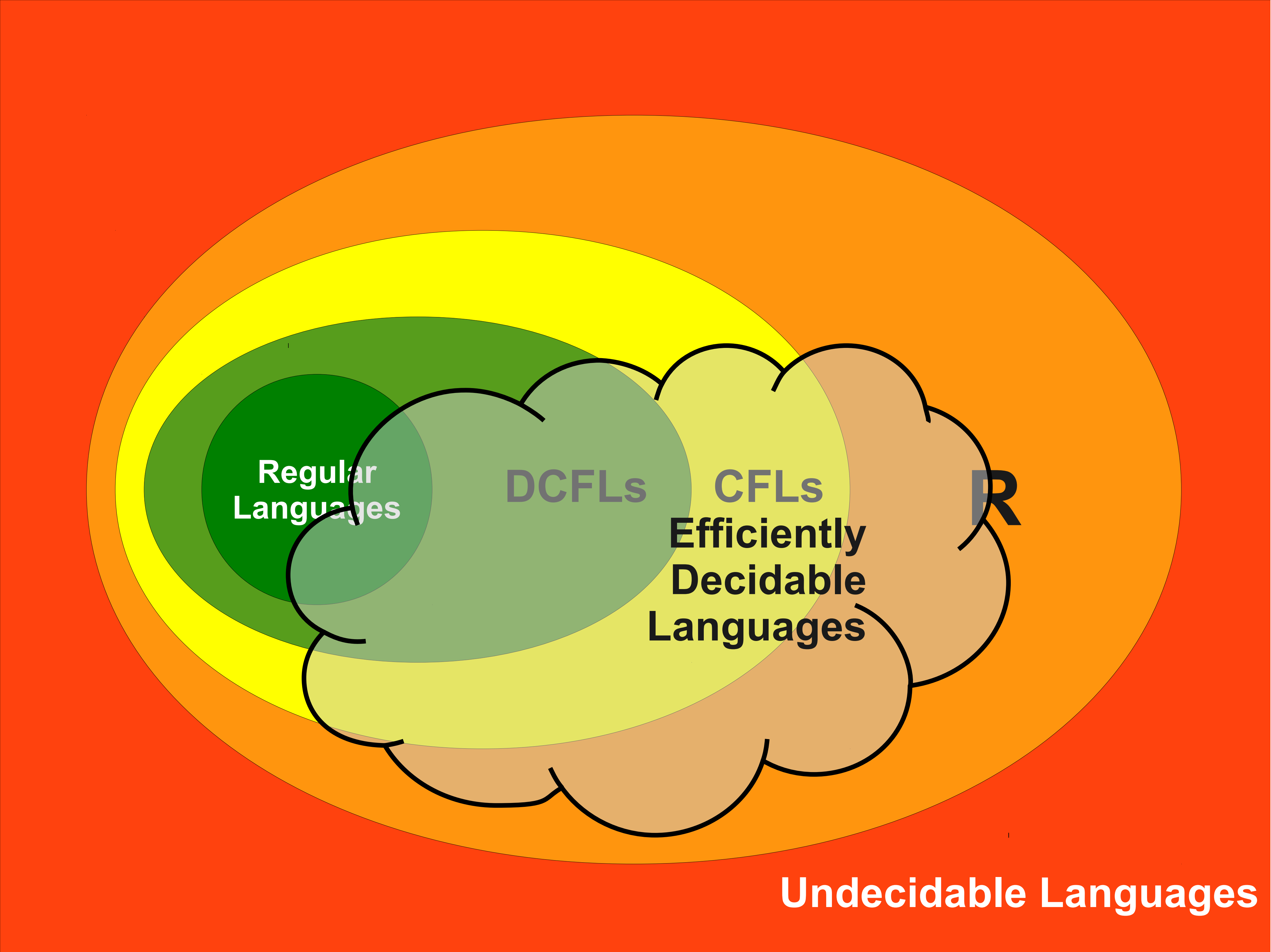
- In **complexity theory**, we ask the question

Is it possible to solve problem L **efficiently**?

- In the remainder of this course, we will explore this question in more detail.


Regular Languages
DCFLs
CFLs
R
RE
All Languages


Regular Languages
DCFLs
CFLs
R
RE
All Languages


Regular Languages
DCFLs
CFLs
R
Efficiently Decidable Languages
Undecidable Languages

The Setup

- In order to study computability, we needed to answer these questions:
  - What is “computation?”
  - What is a “problem?”
  - What does it mean to “solve” a problem?
- To study complexity, we need to answer these questions:
  - What does “complexity” even mean?
  - What is an “efficient” solution to a problem?

Measuring Complexity

- Suppose that we have a decider D for some language L.
- How might we measure the complexity of D?

Measuring Complexity

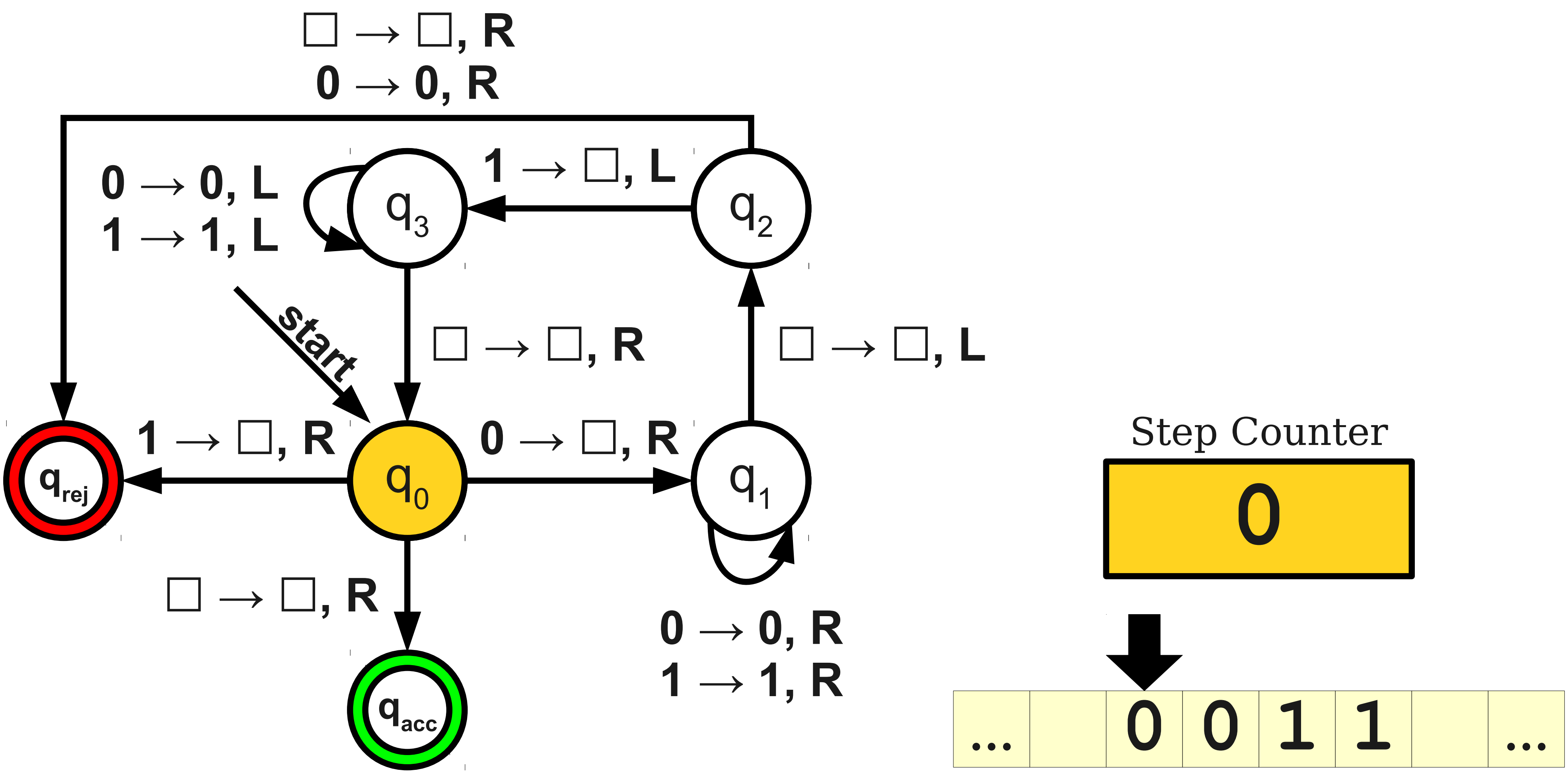
- Suppose that we have a decider D for some language L.
- How might we measure the complexity of D?
  - Number of states.
  - Size of tape alphabet.
  - Size of input alphabet.
  - Amount of tape required.
  - Number of steps required.
  - Number of times a given state is entered.
  - Number of times a given symbol is printed.
  - Number of times a given transition is taken.
  - (Plus a whole lot more…)

Measuring Complexity

- Suppose that we have a decider D for some language L.
- How might we measure the complexity of D?
  - Number of states.
  - Size of tape alphabet.
  - Size of input alphabet.
  - Amount of tape required.
  - Number of steps required.
  - Number of times a given state is entered.
  - Number of times a given symbol is printed.
  - Number of times a given transition is taken.
  - (Plus a whole lot more...)

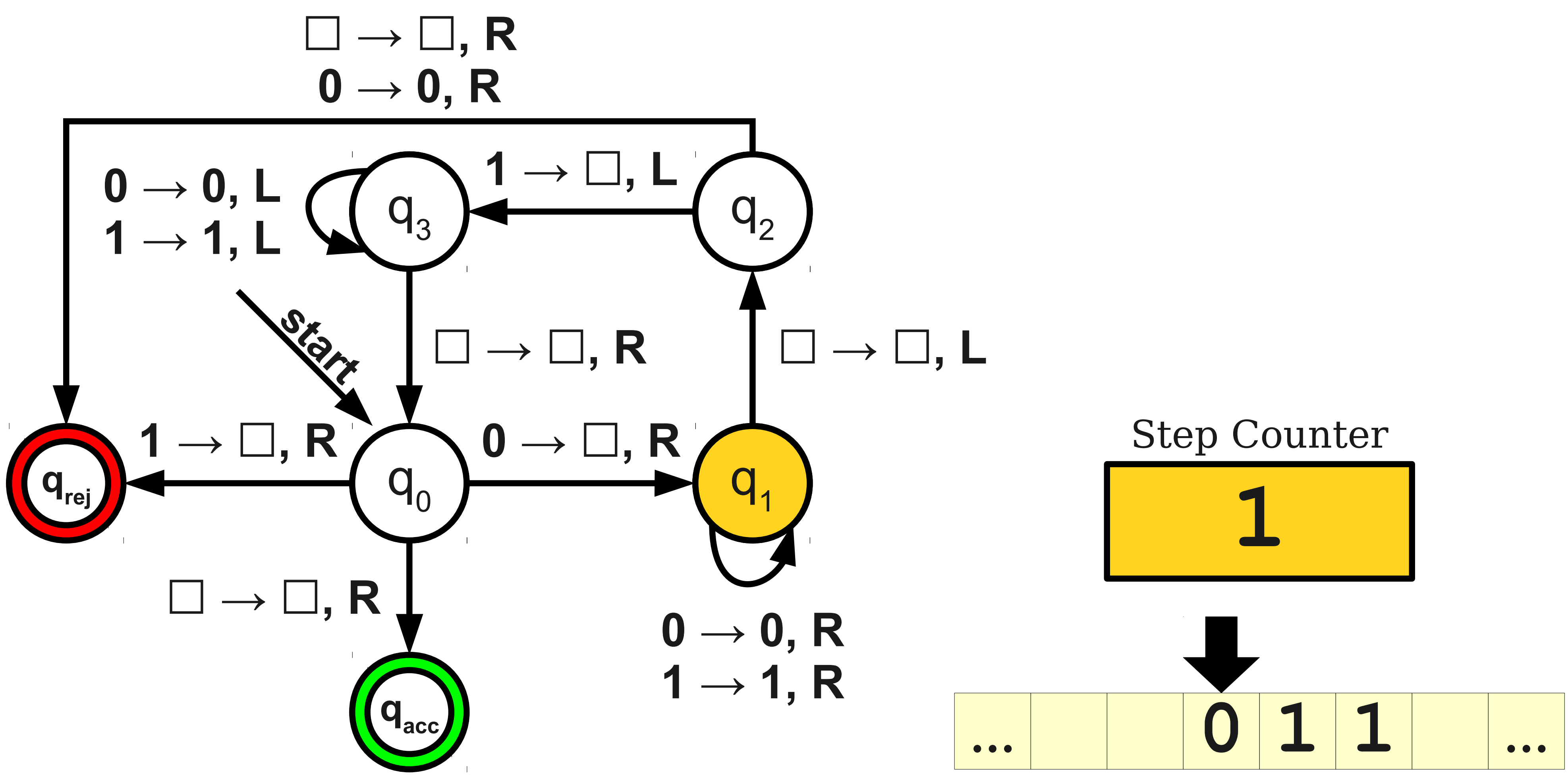
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



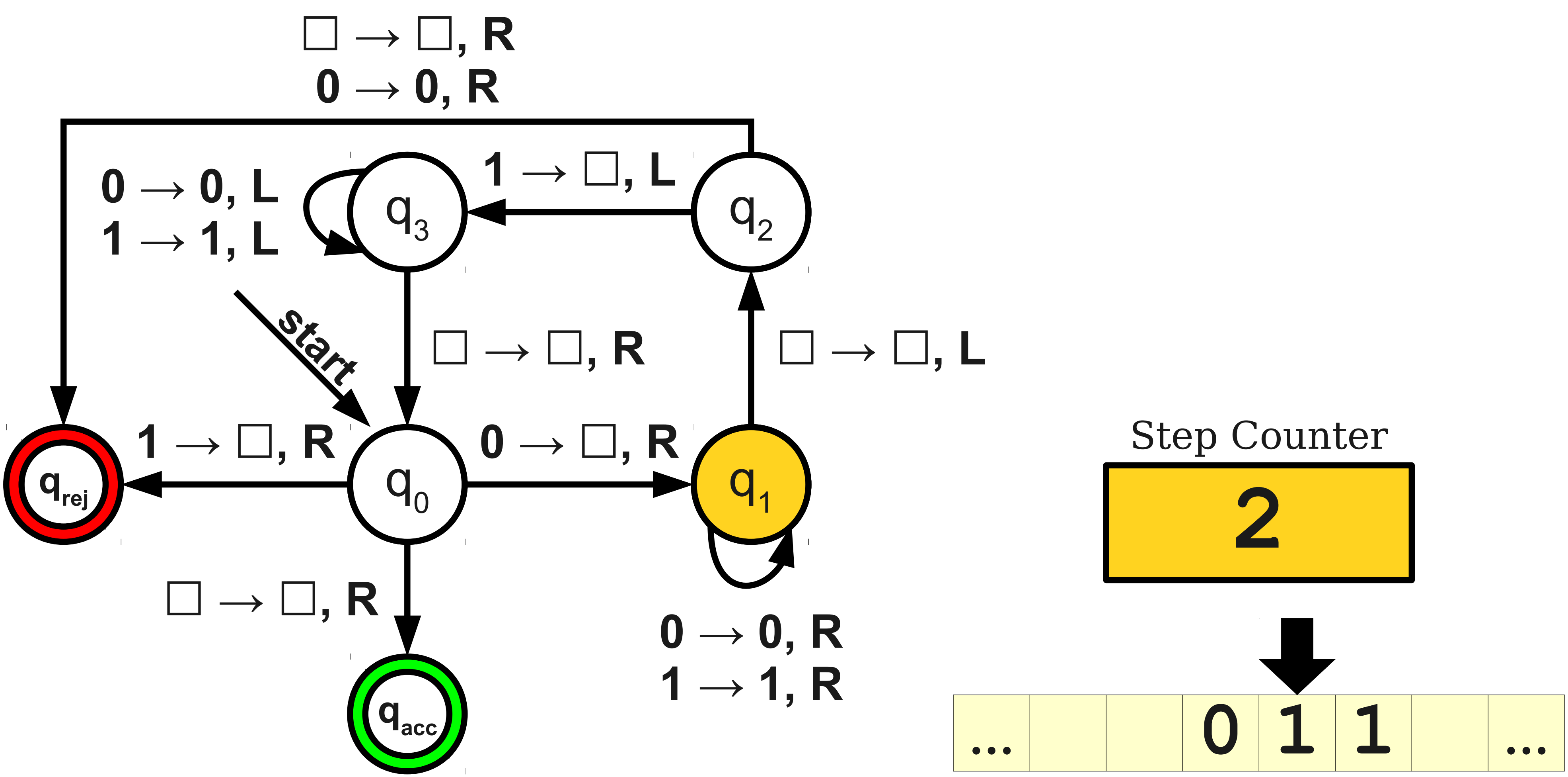
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



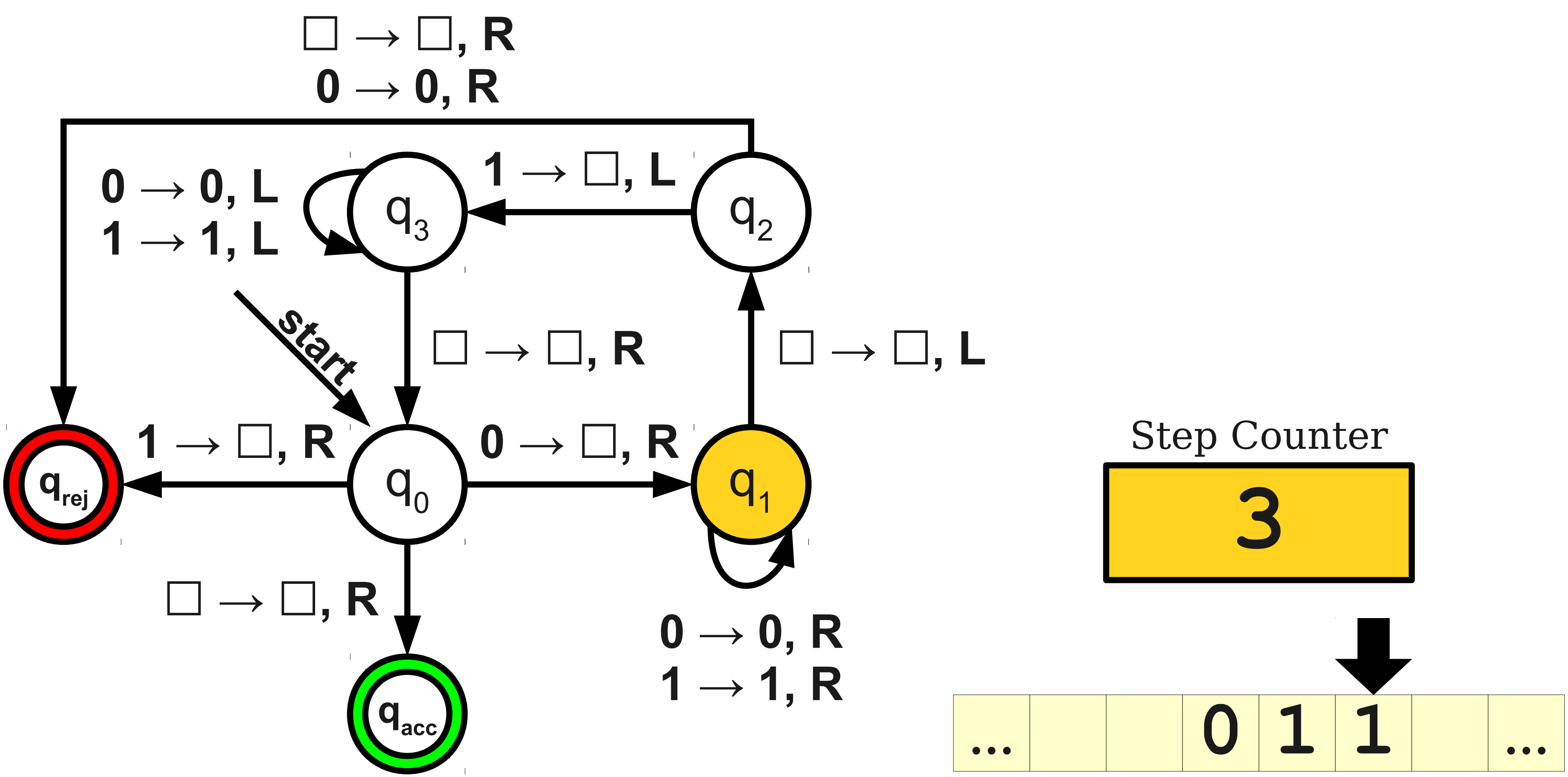
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



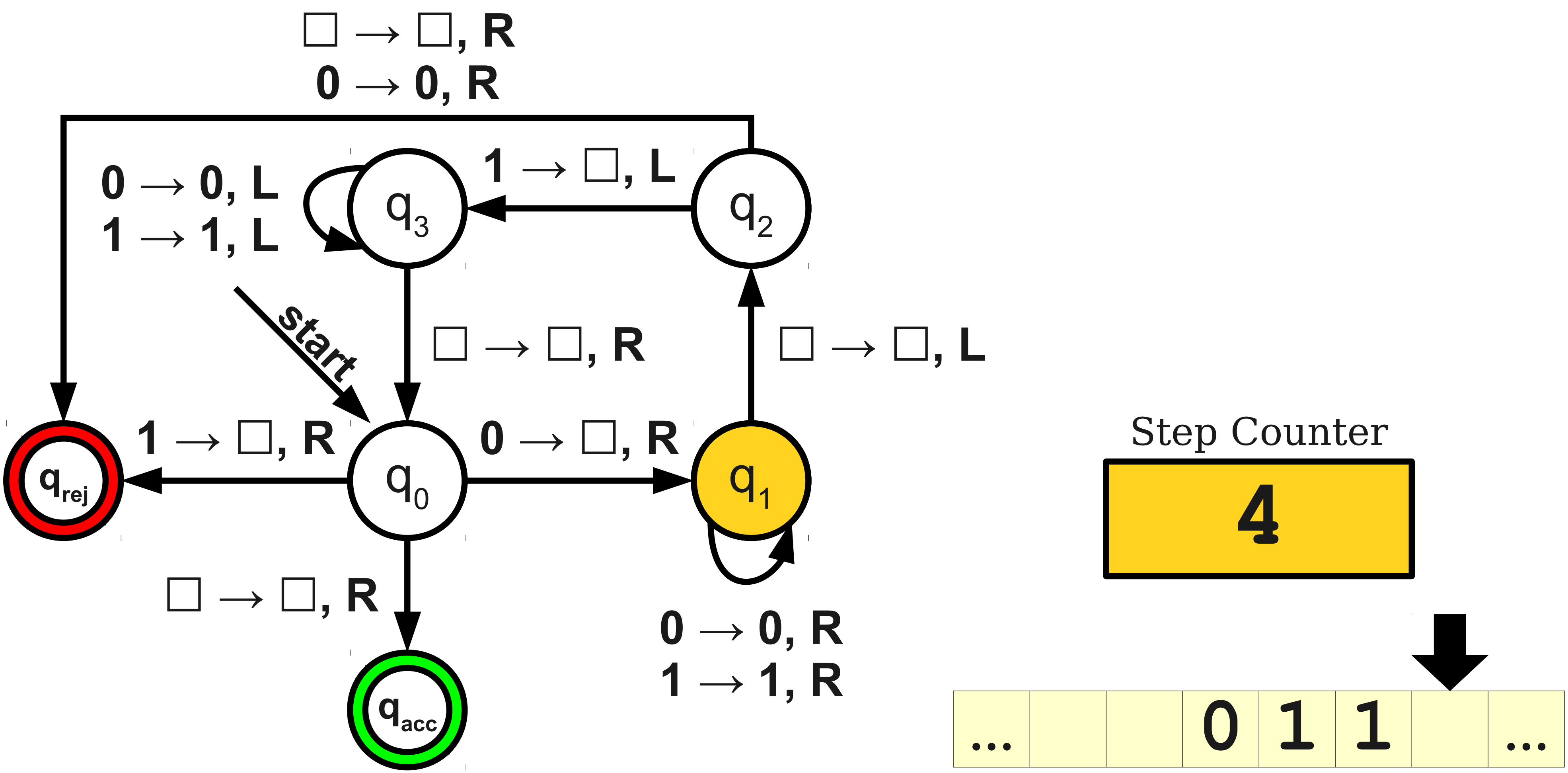
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



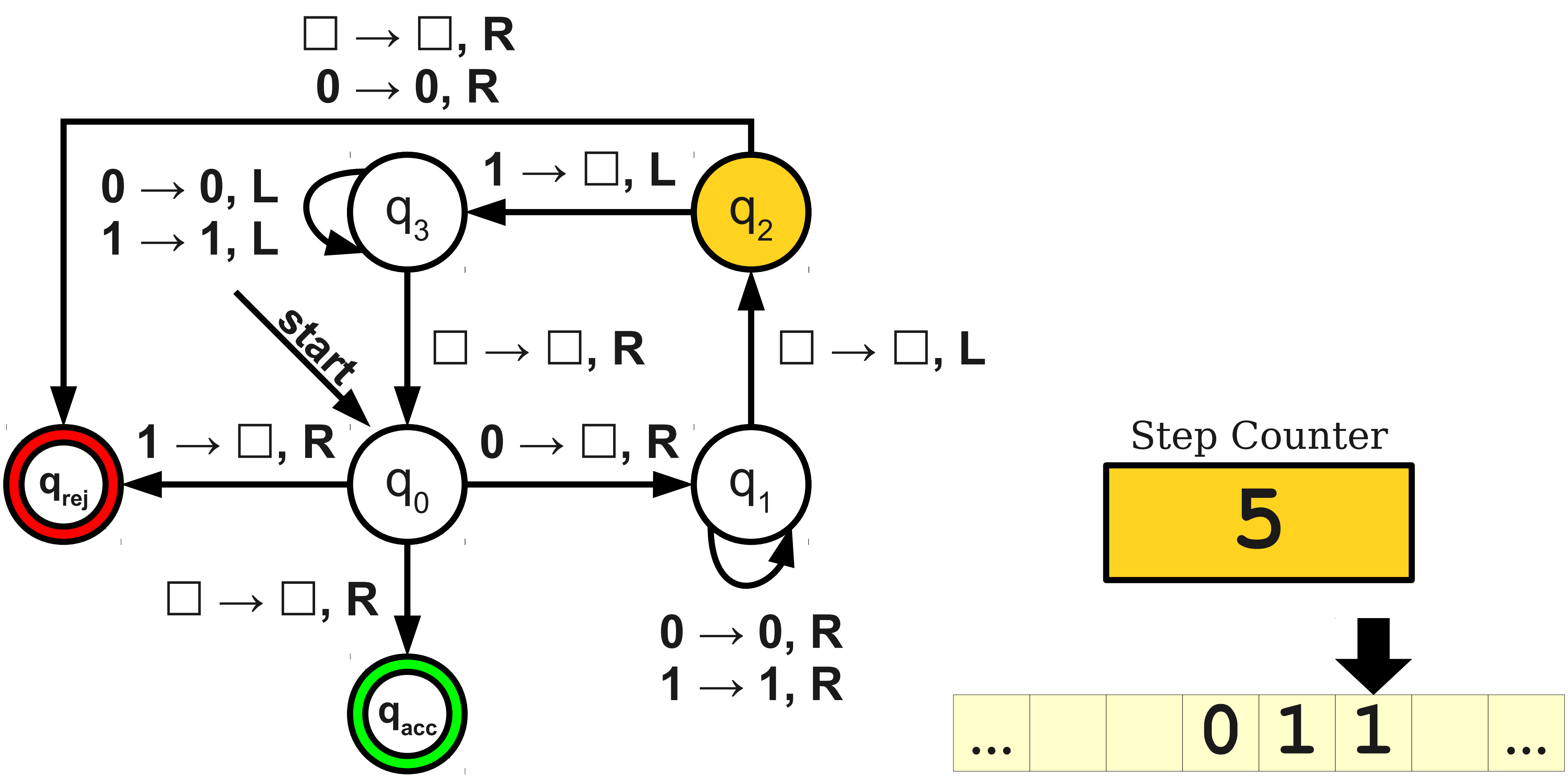
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



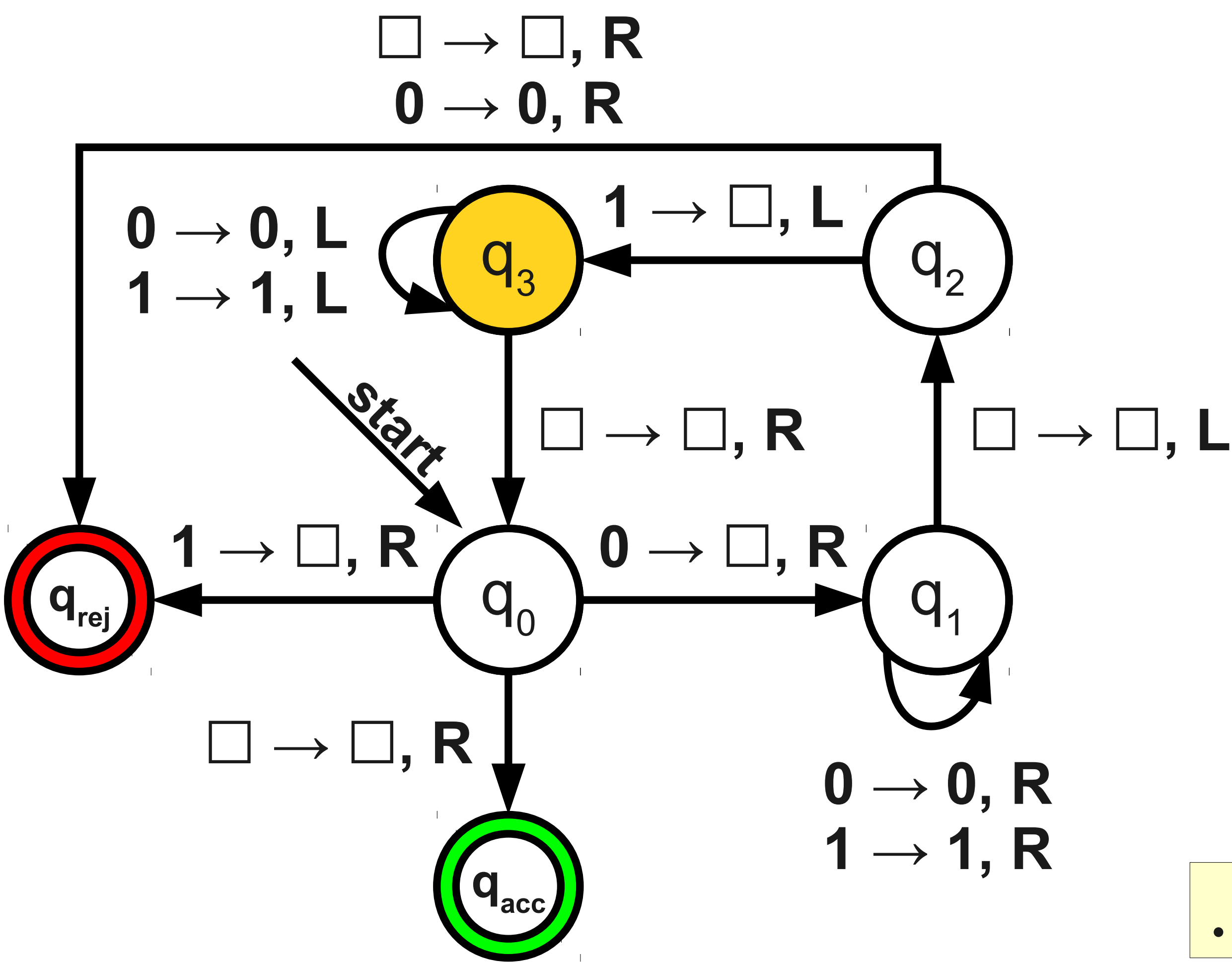
Time Complexity

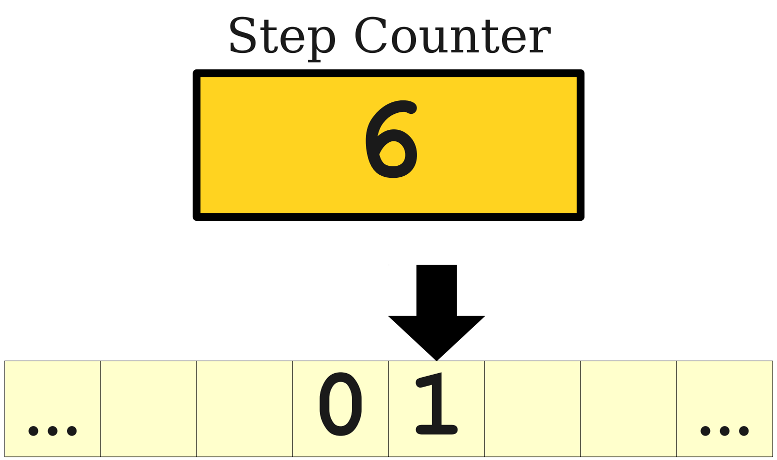
- A **step** of a Turing machine is one event where the TM takes a transition.



Time Complexity

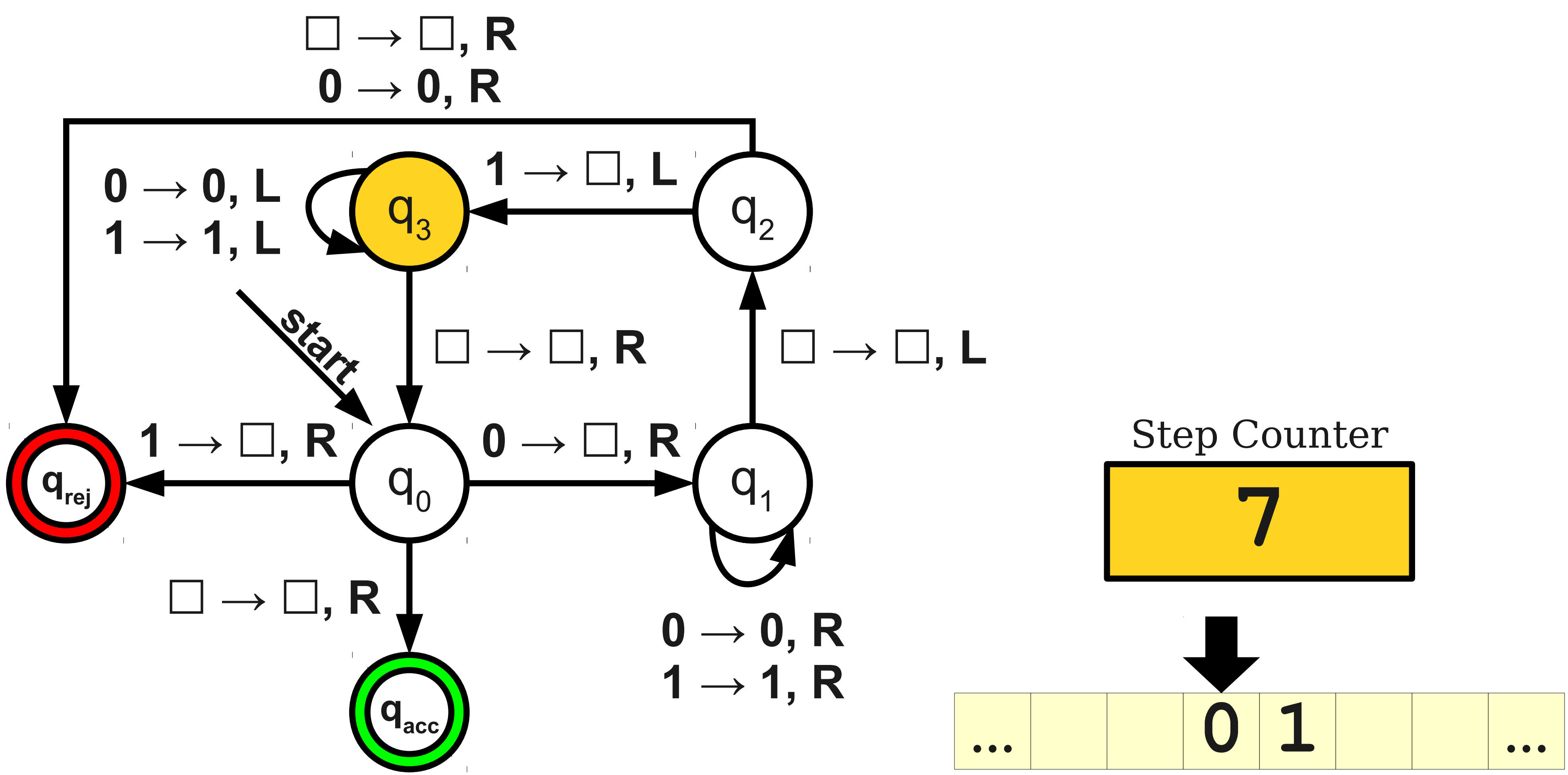
- A **step** of a Turing machine is one event where the TM takes a transition.





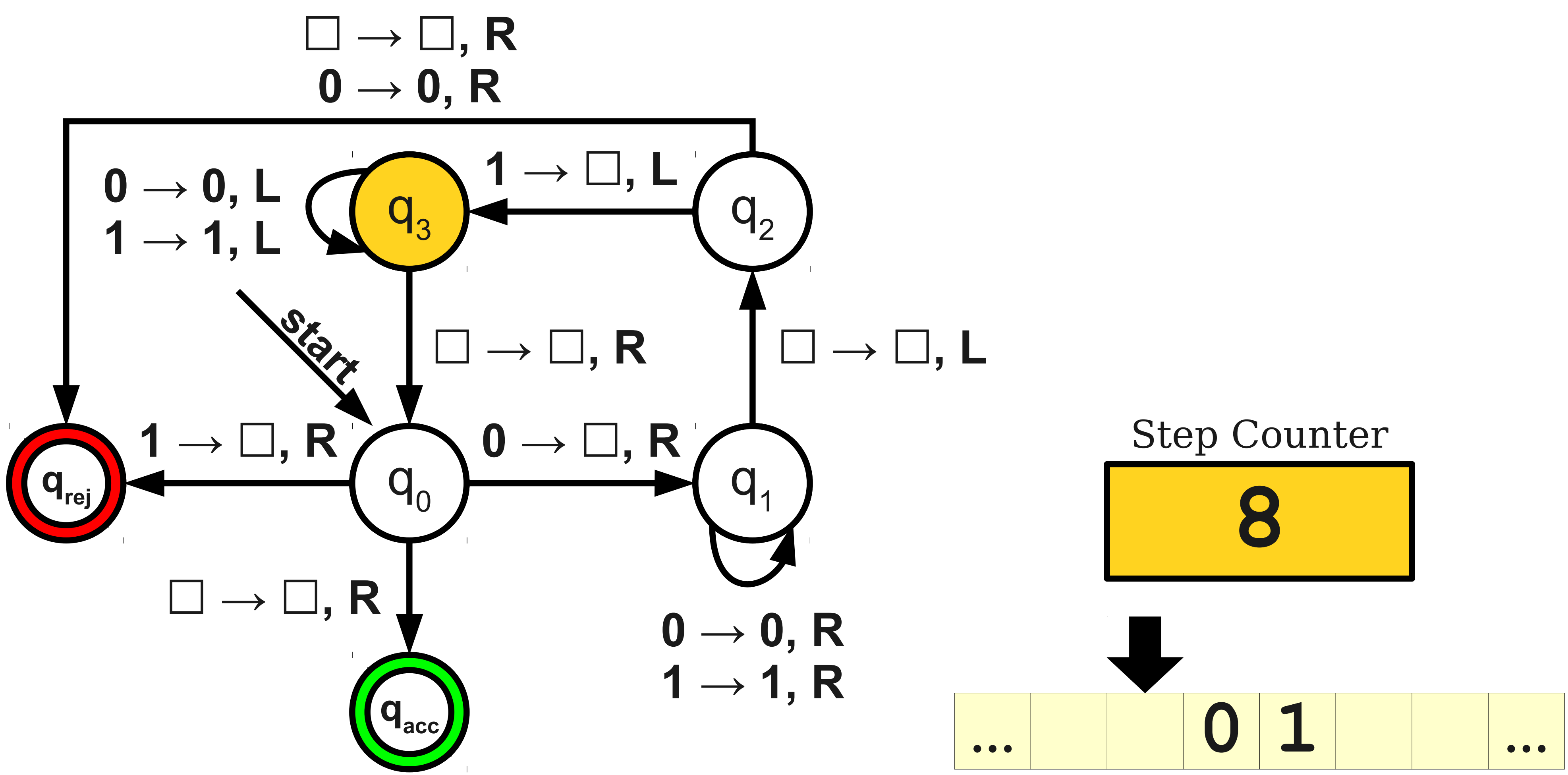
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



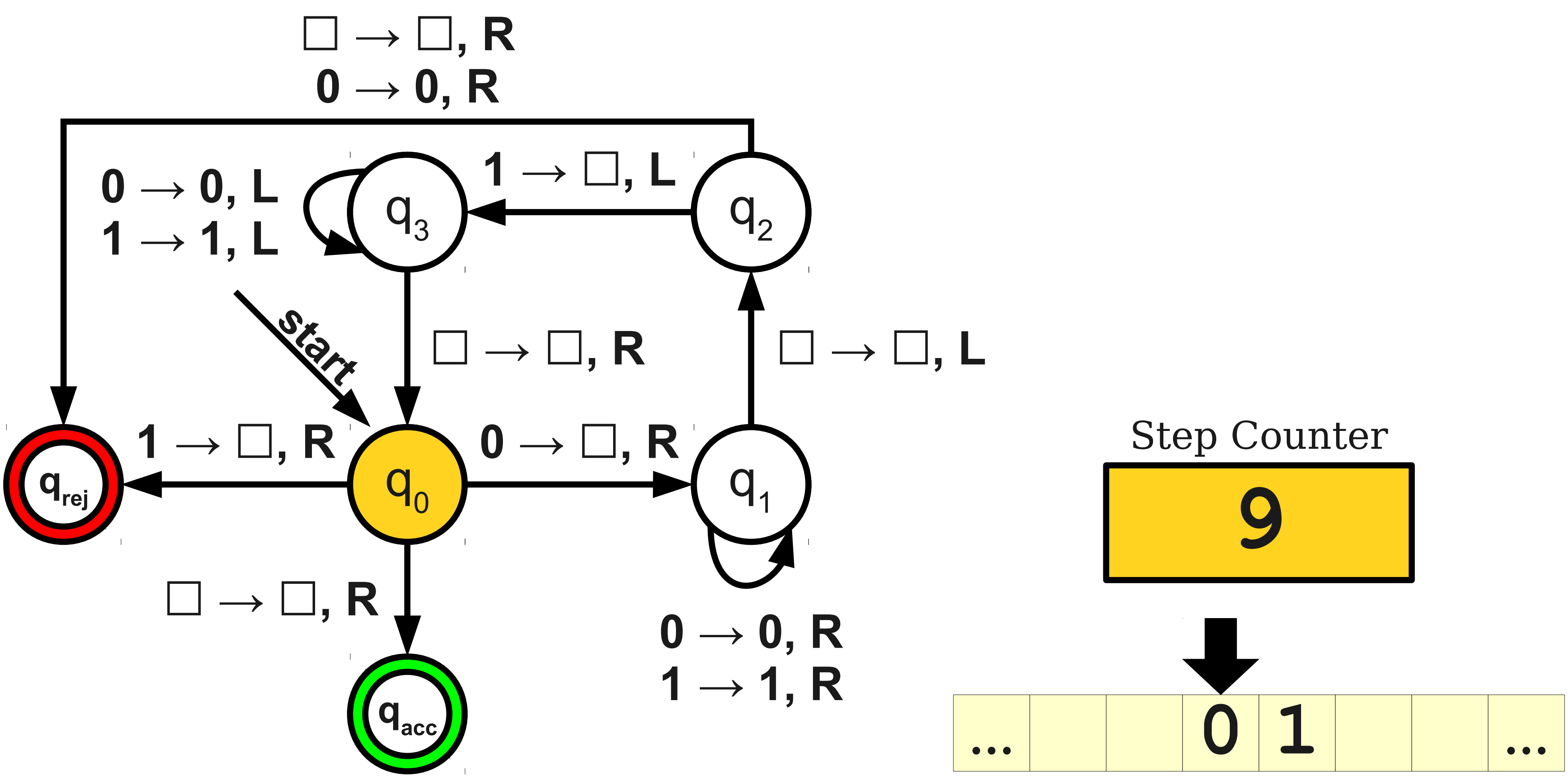
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



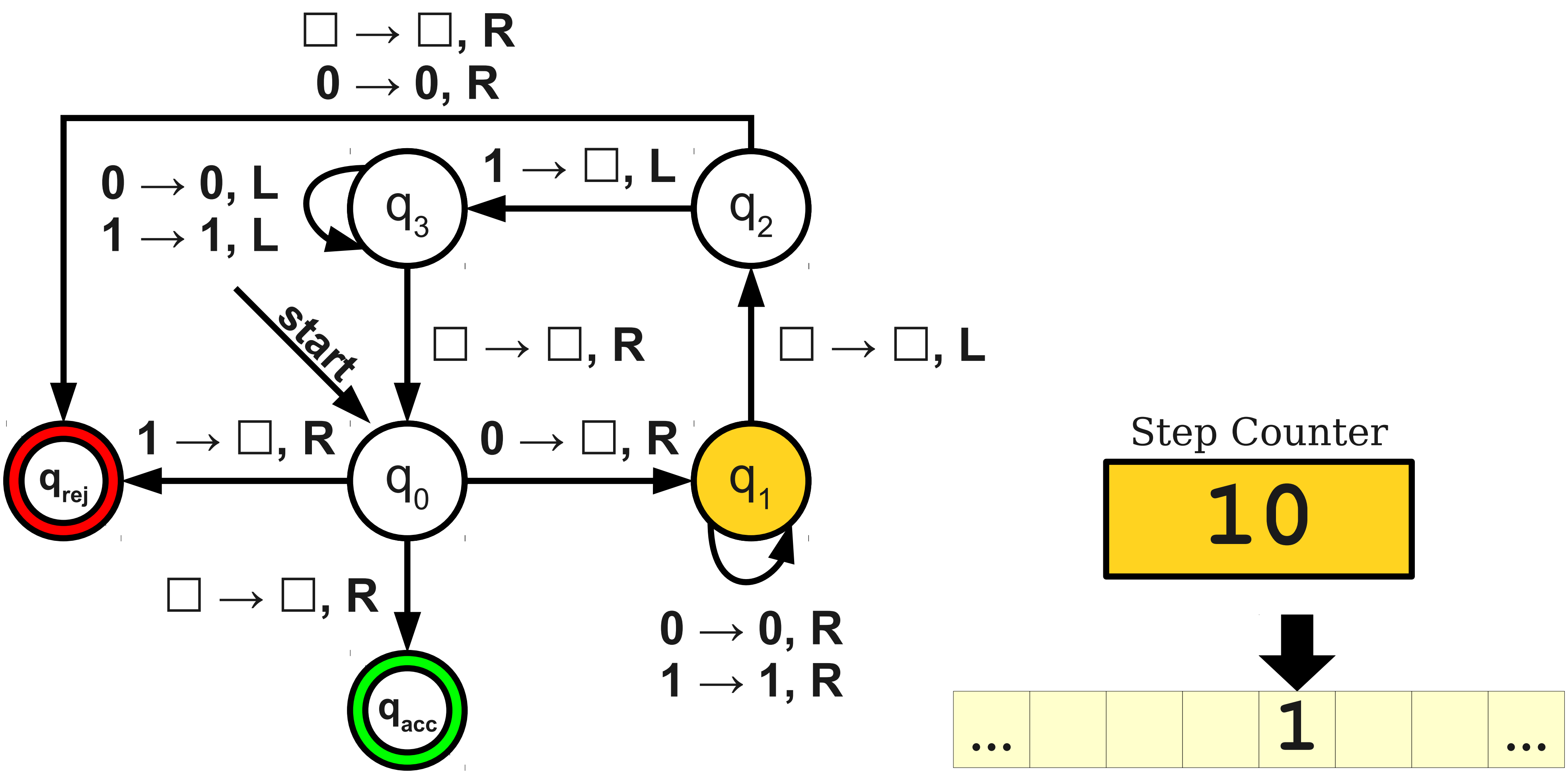
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



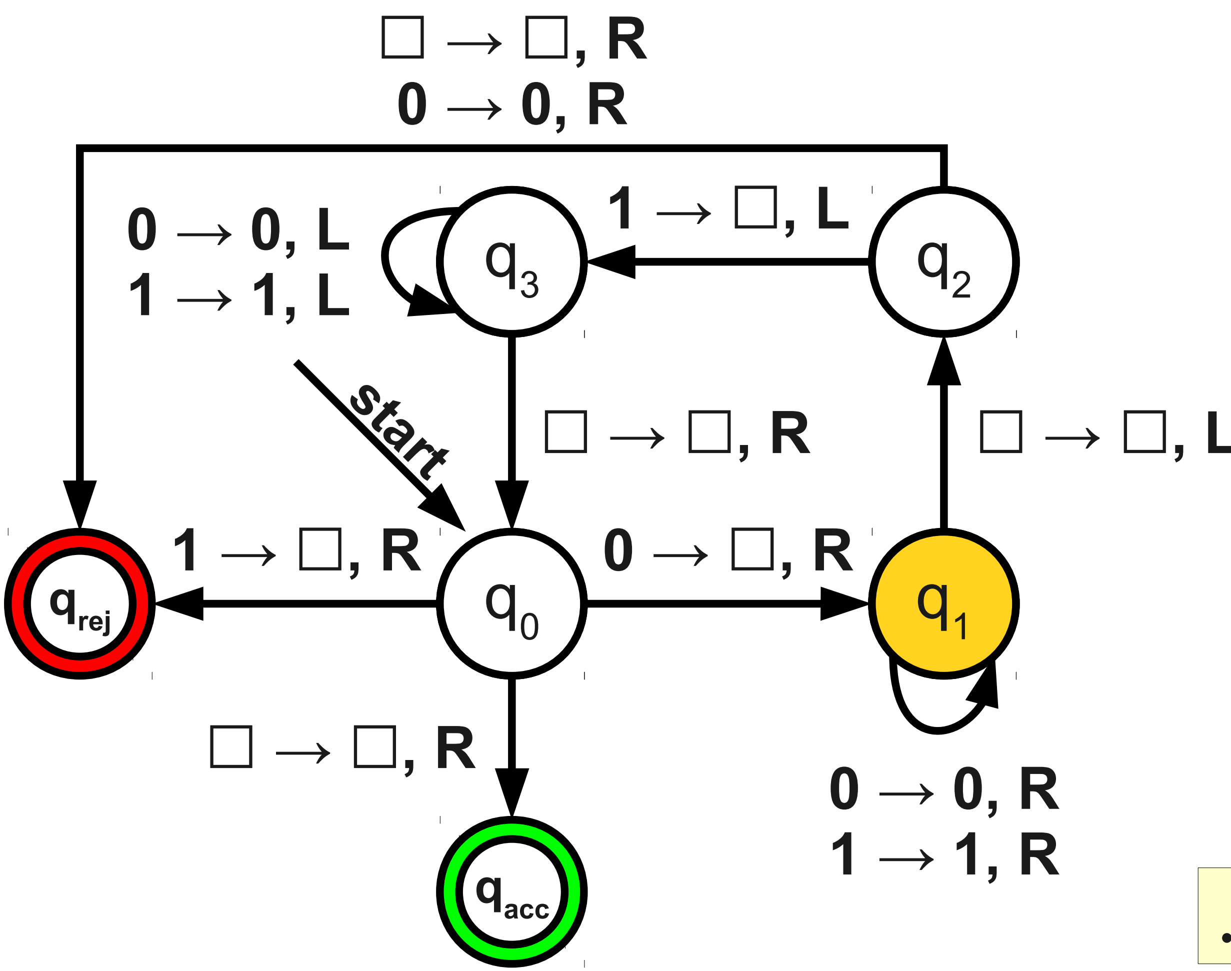
Time Complexity

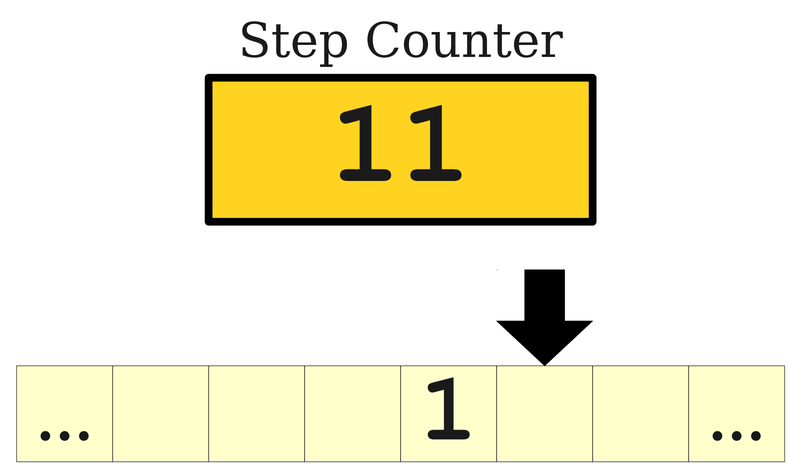
- A **step** of a Turing machine is one event where the TM takes a transition.



Time Complexity

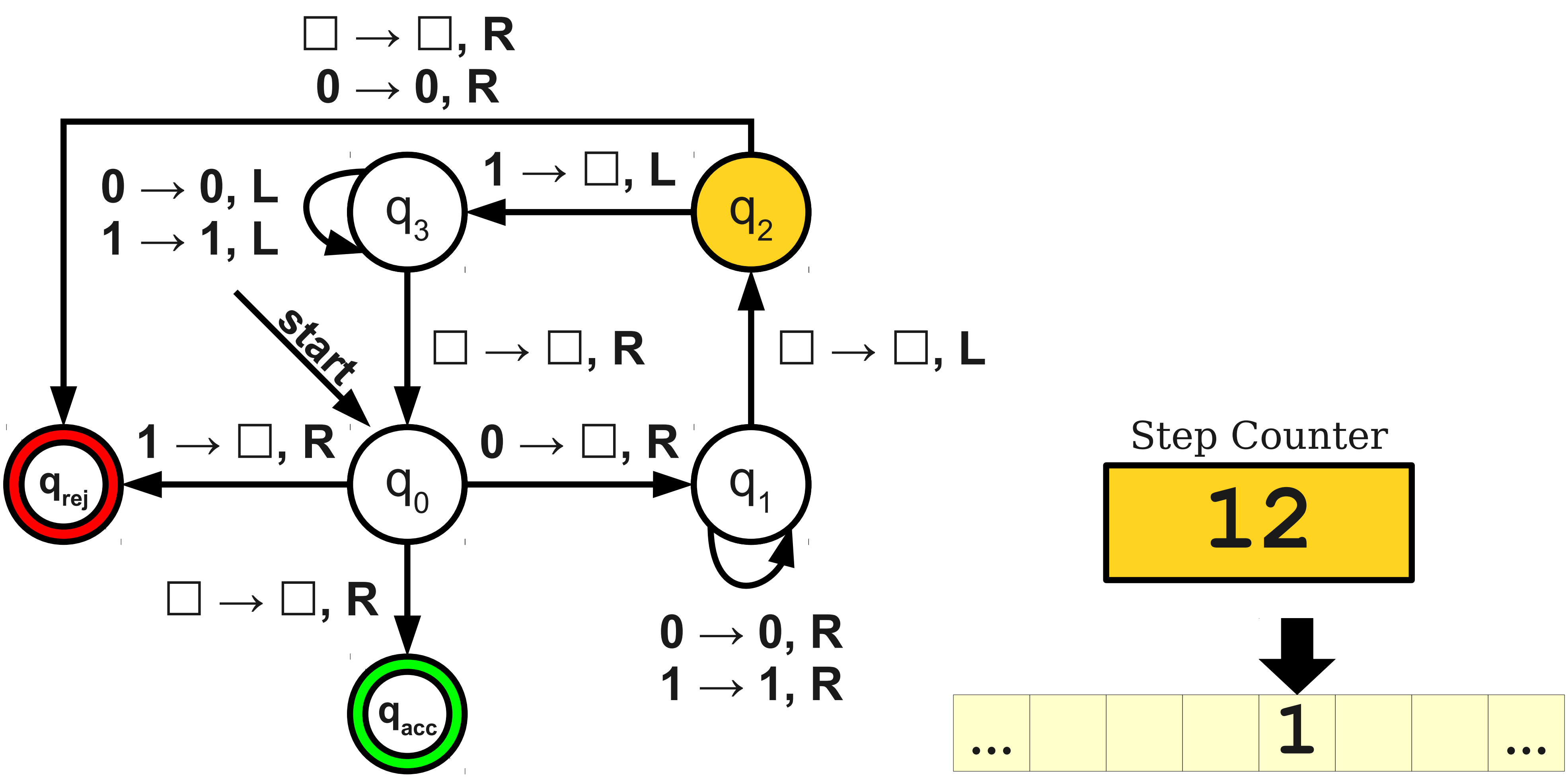
- A **step** of a Turing machine is one event where the TM takes a transition.





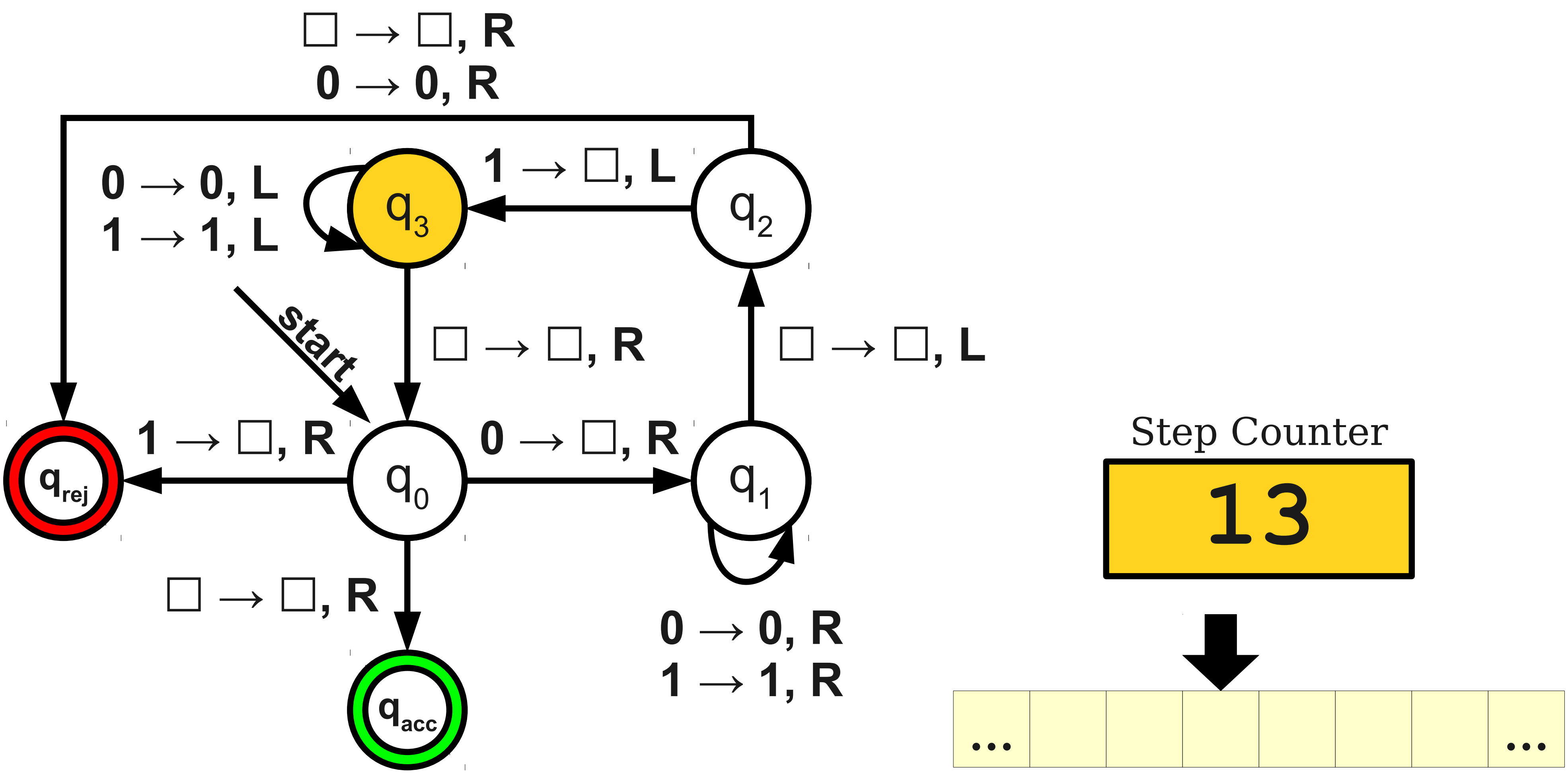
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



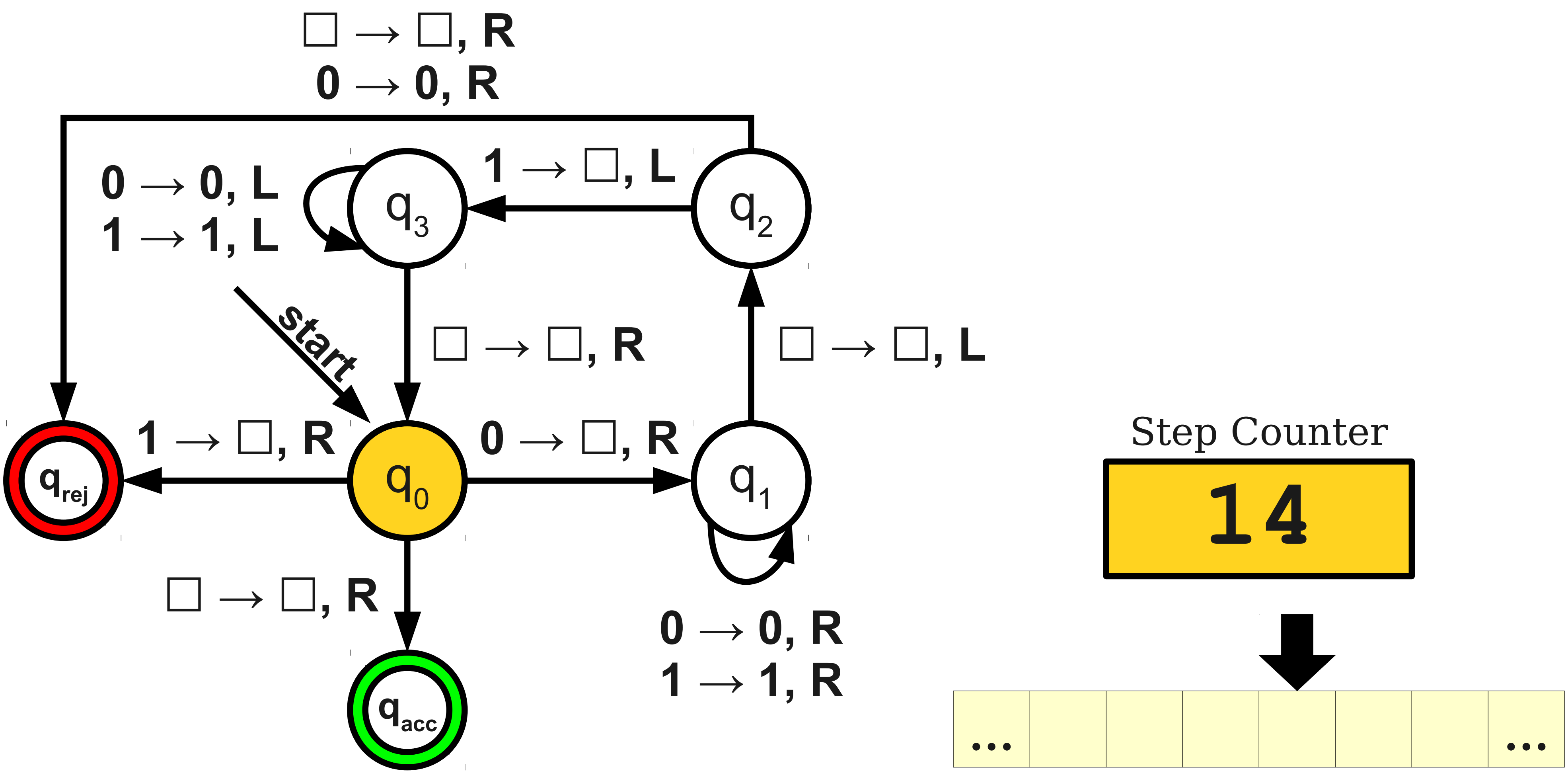
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



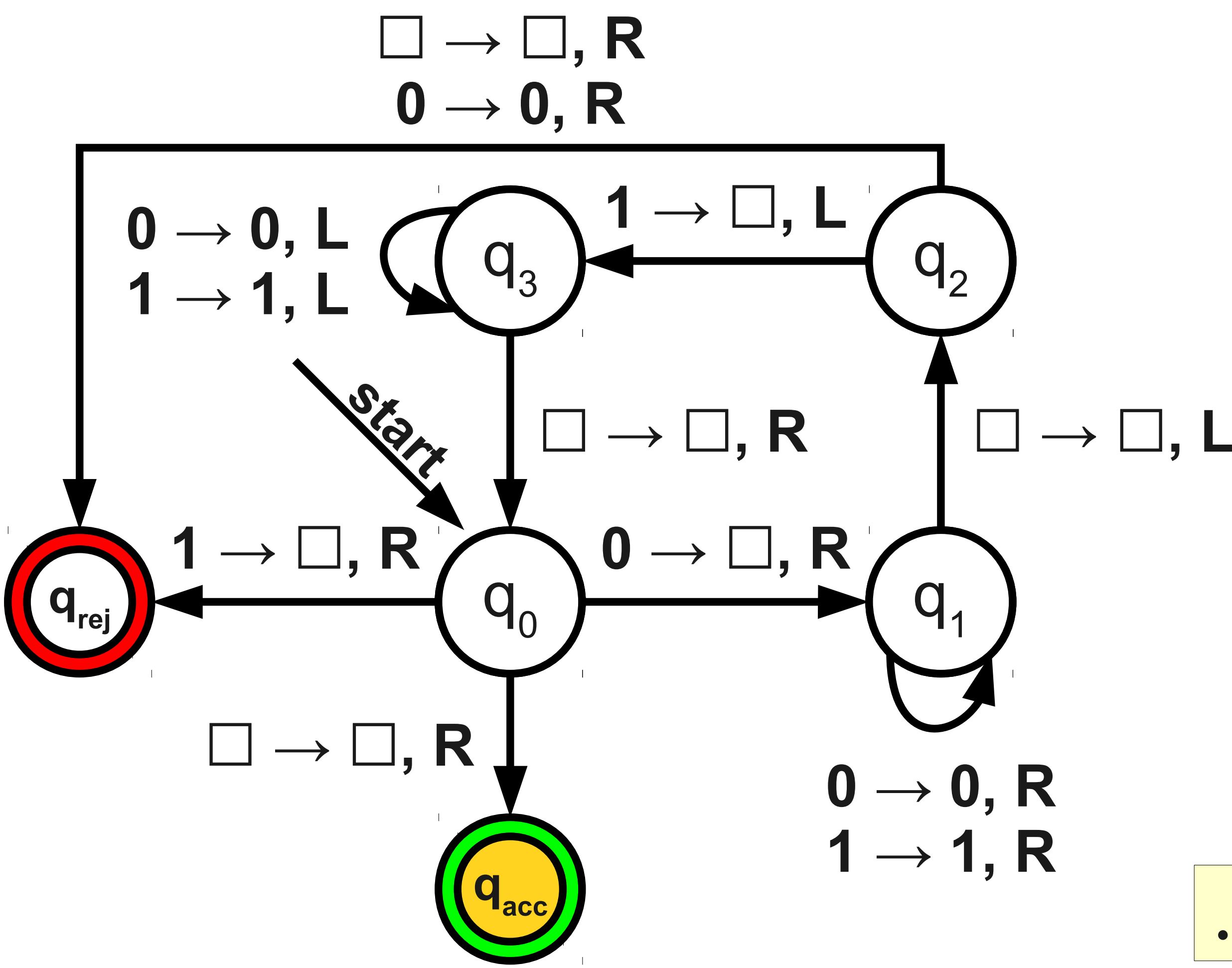
Time Complexity

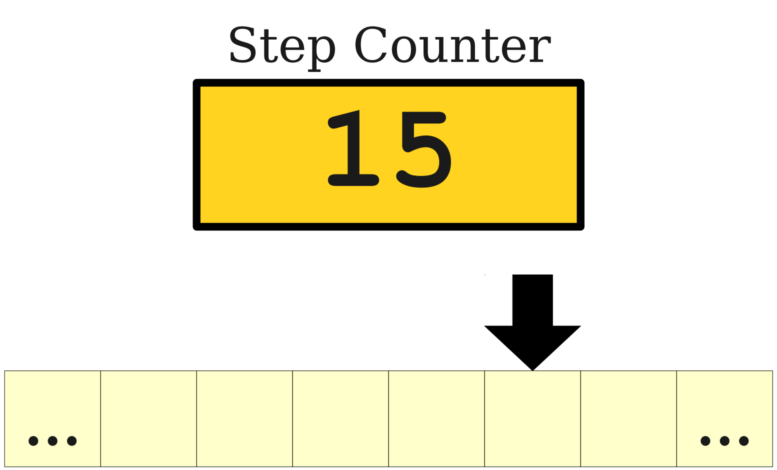
- A **step** of a Turing machine is one event where the TM takes a transition.



Time Complexity

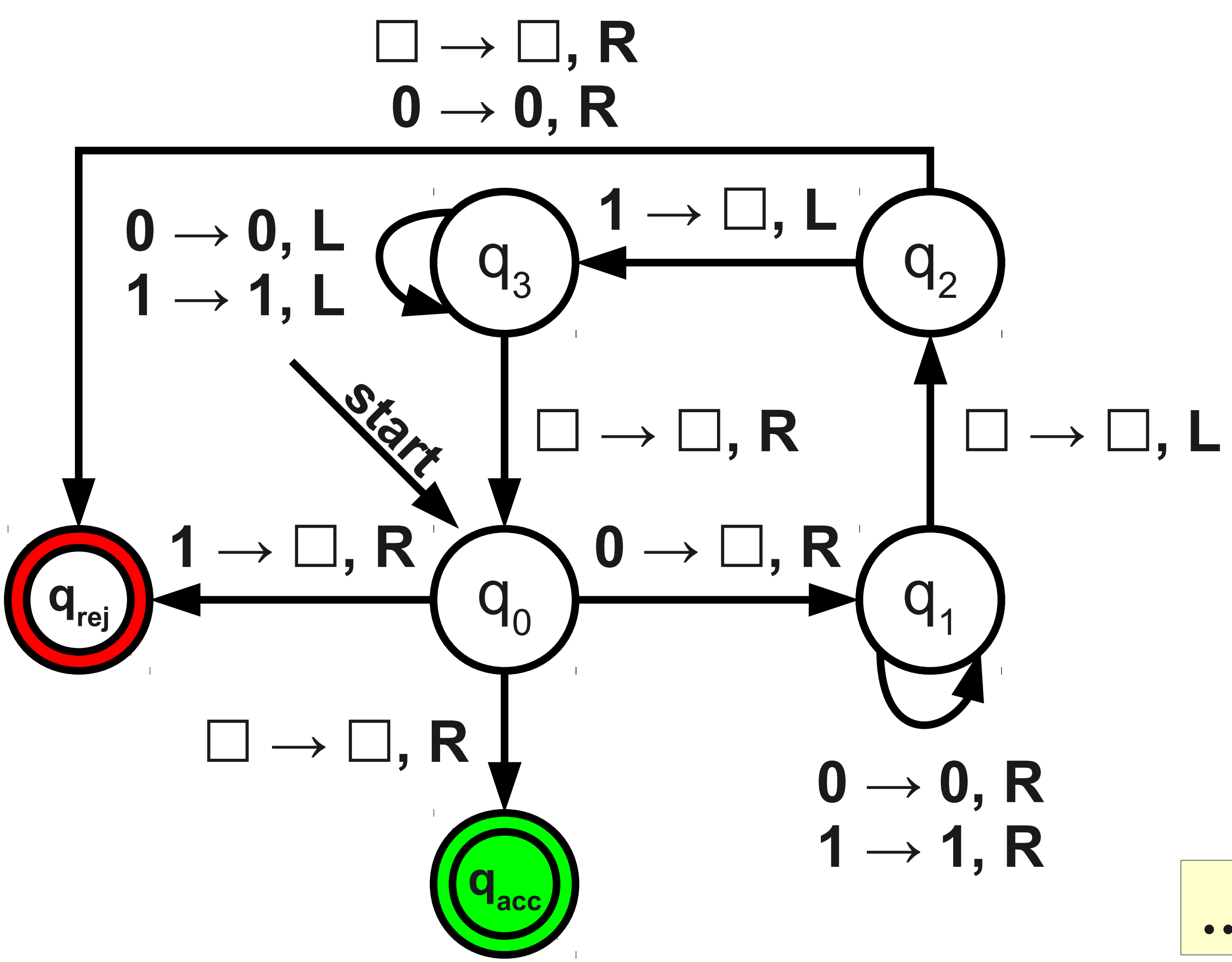
- A **step** of a Turing machine is one event where the TM takes a transition.

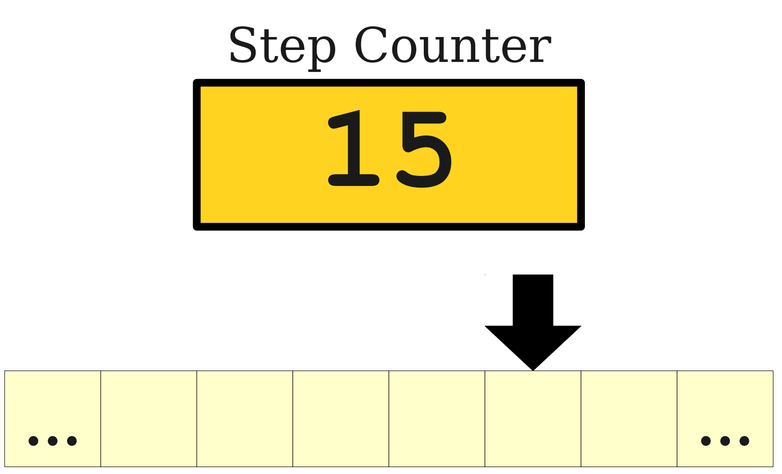




Time Complexity

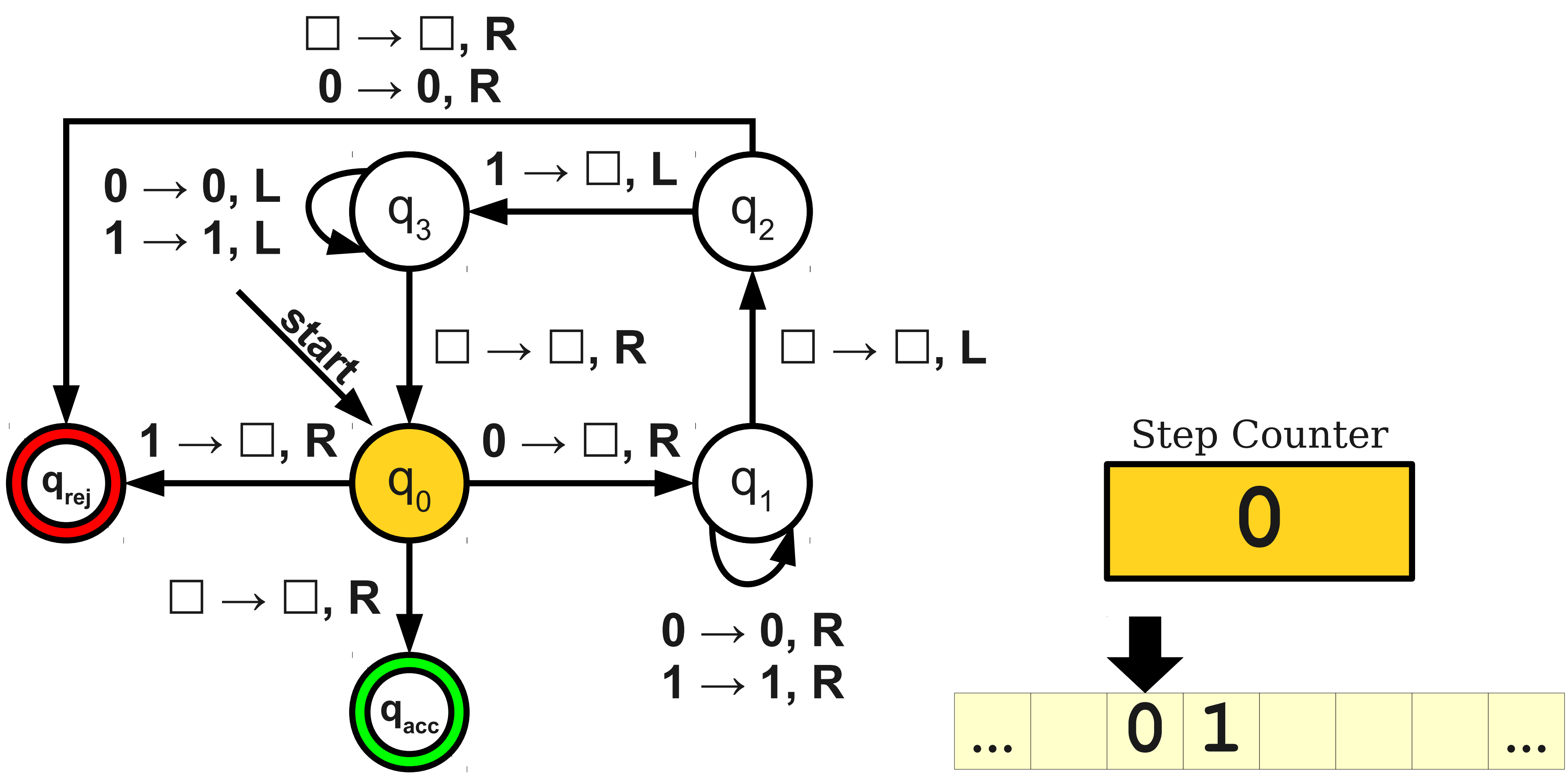
- A **step** of a Turing machine is one event where the TM takes a transition.





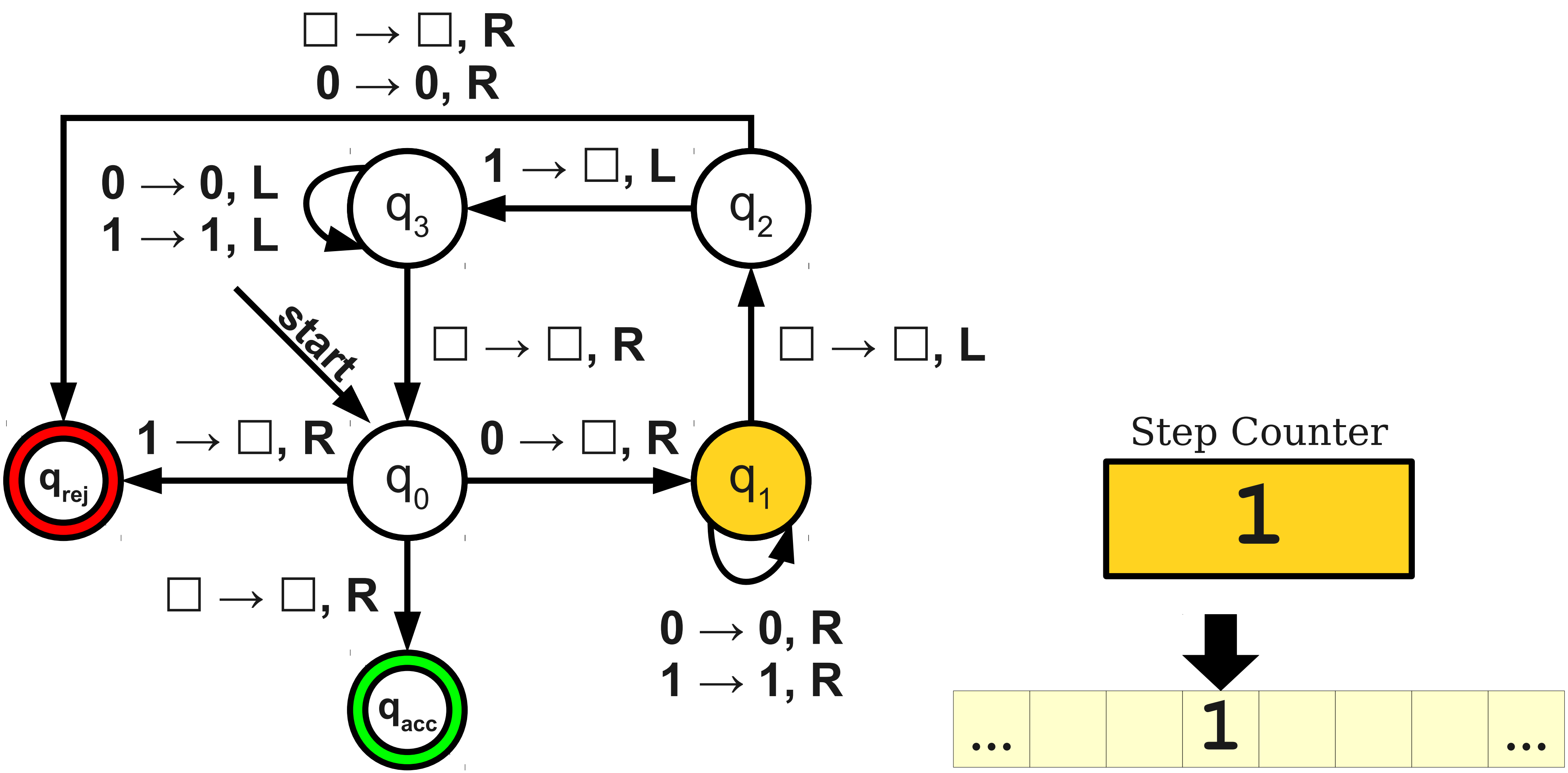
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



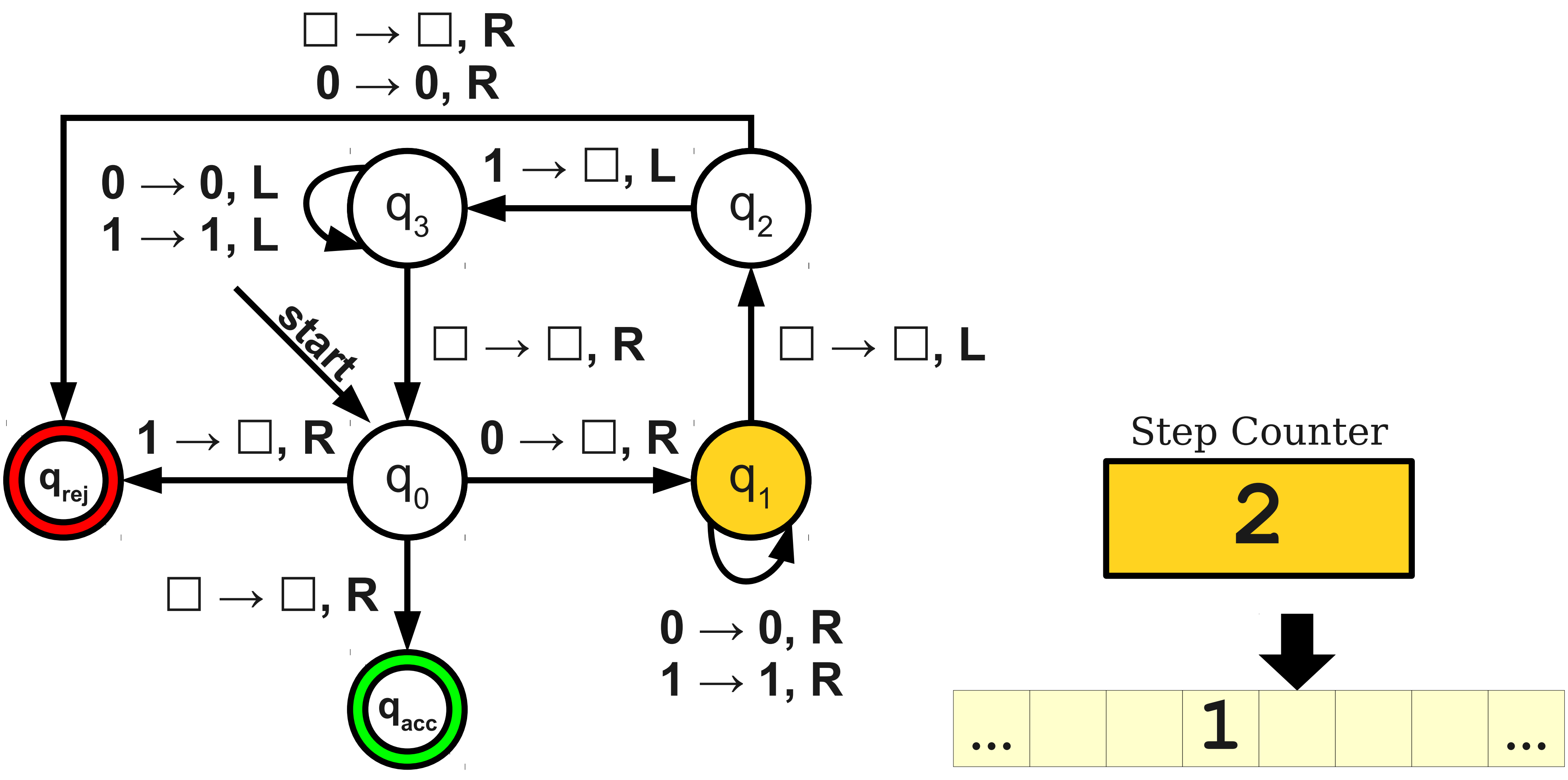
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



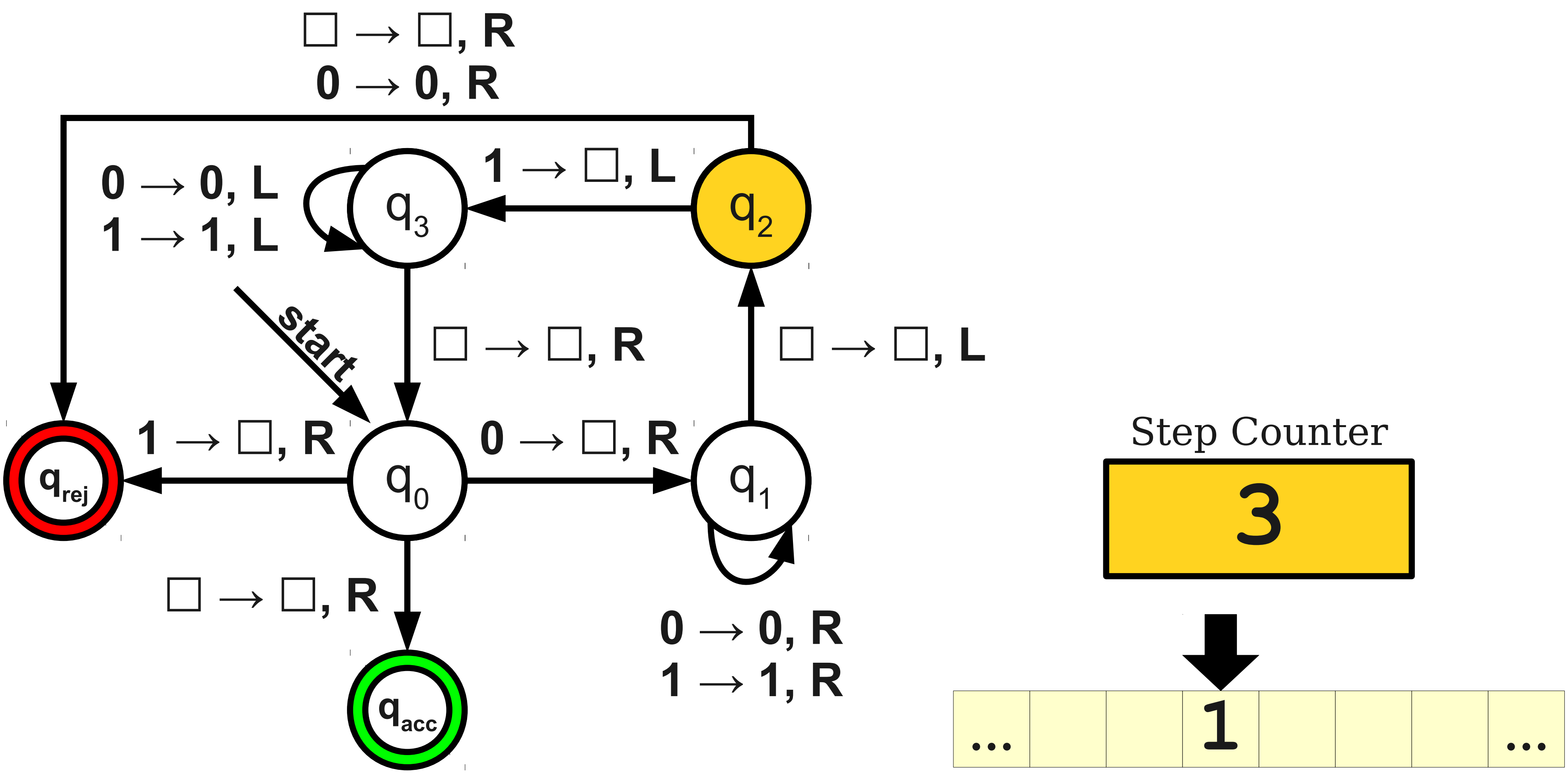
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



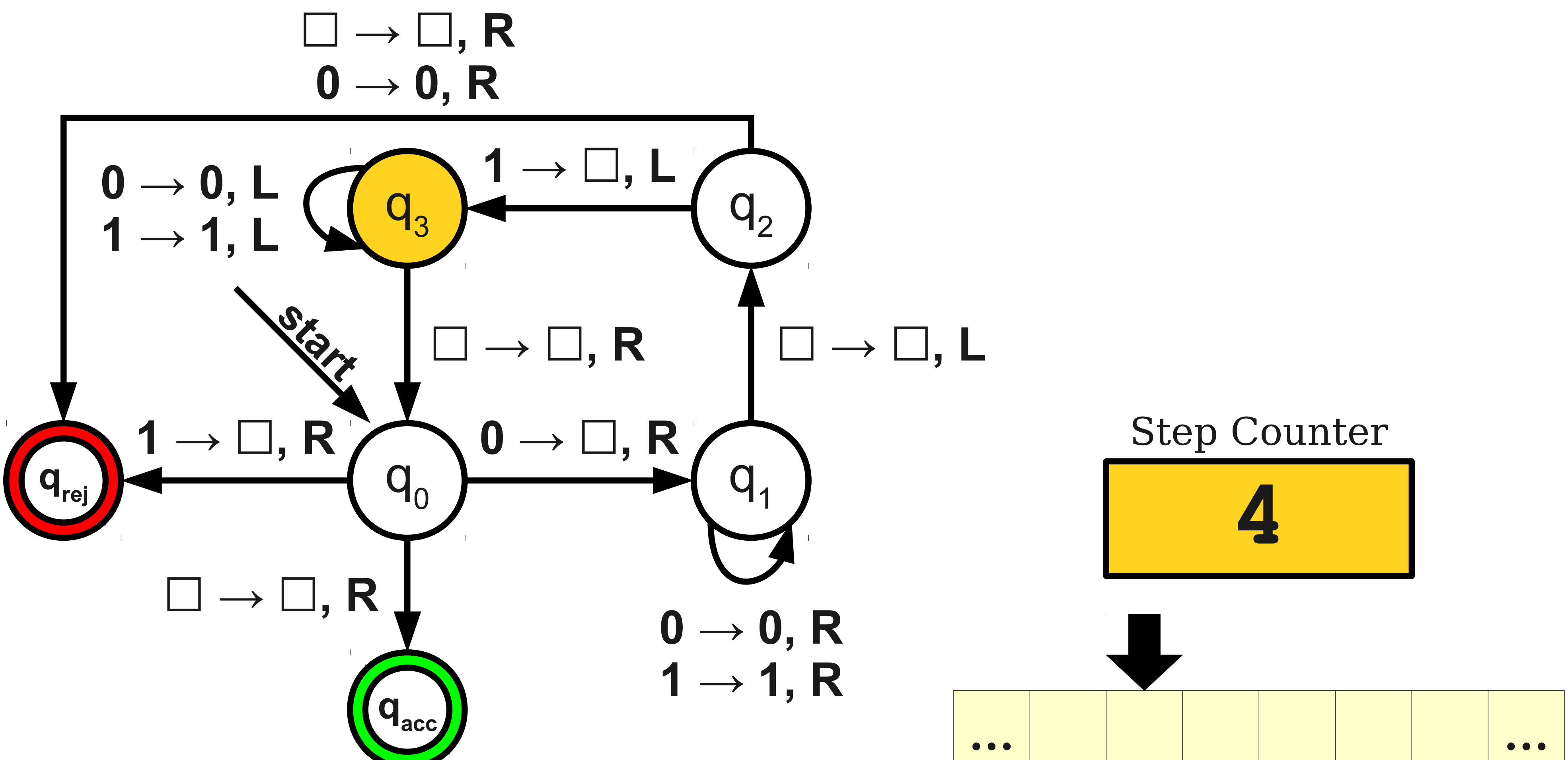
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



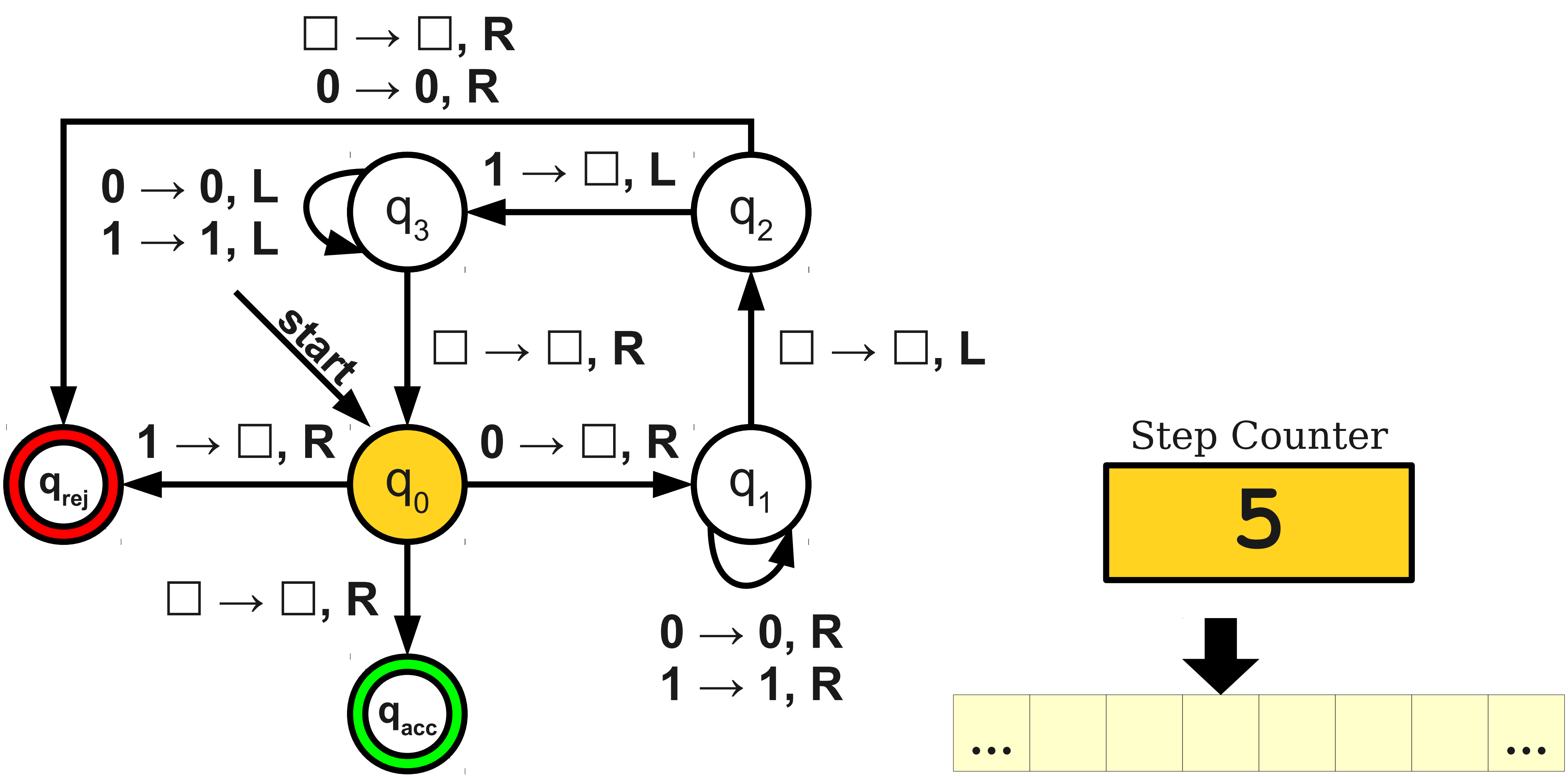
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



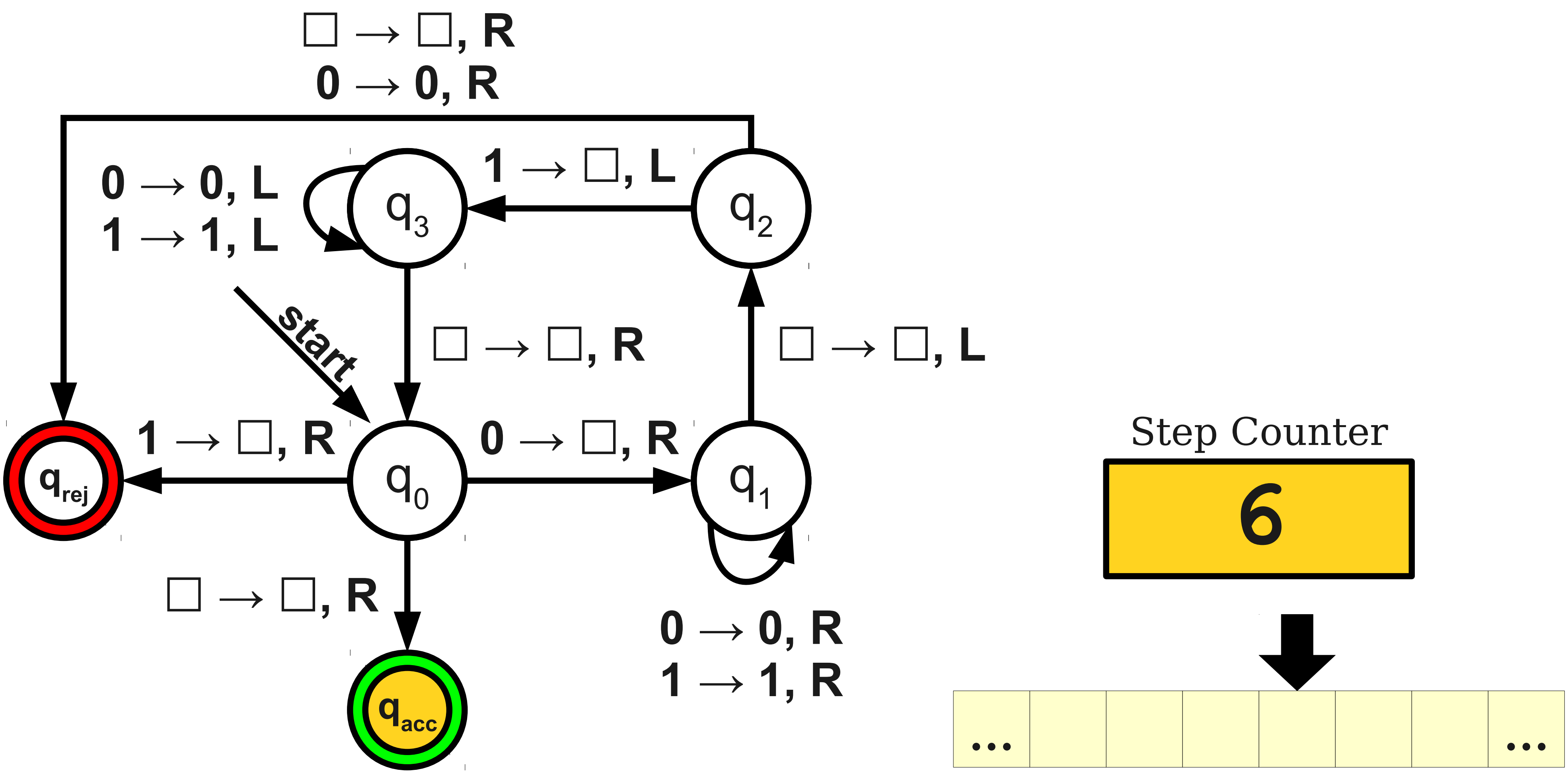
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



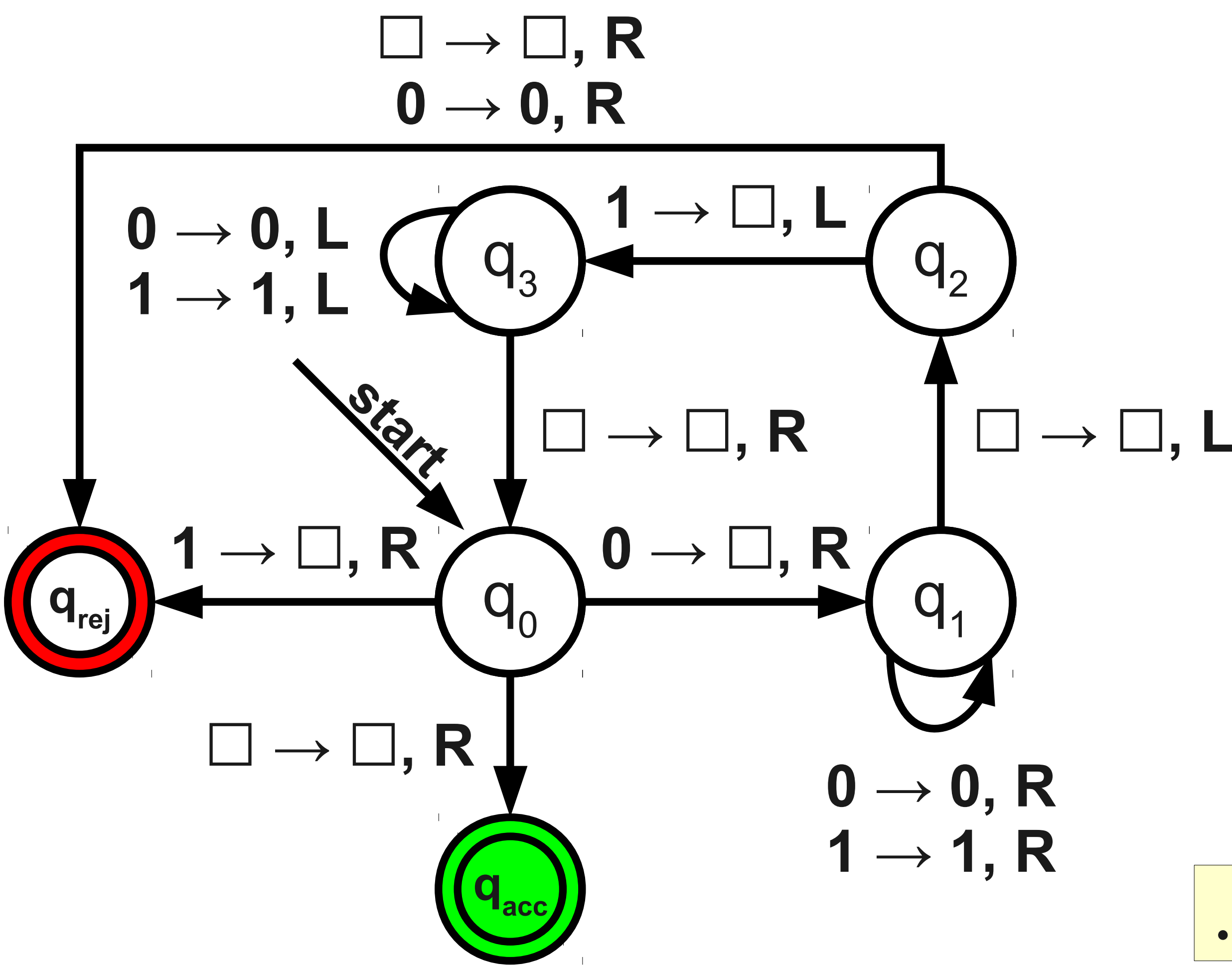
Time Complexity

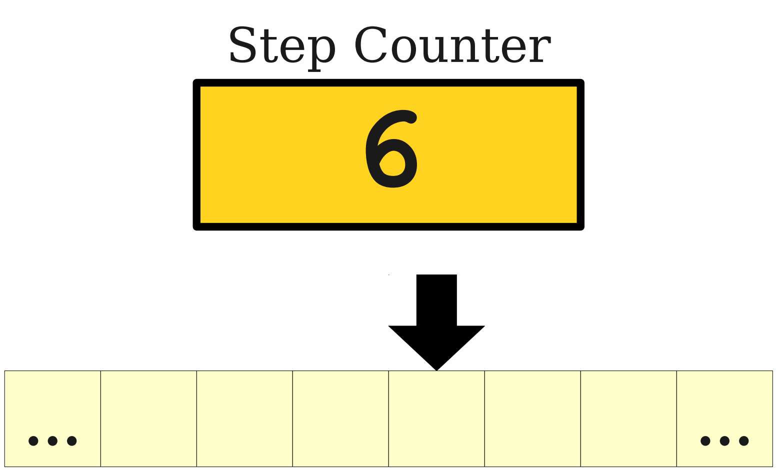
- A **step** of a Turing machine is one event where the TM takes a transition.



Time Complexity

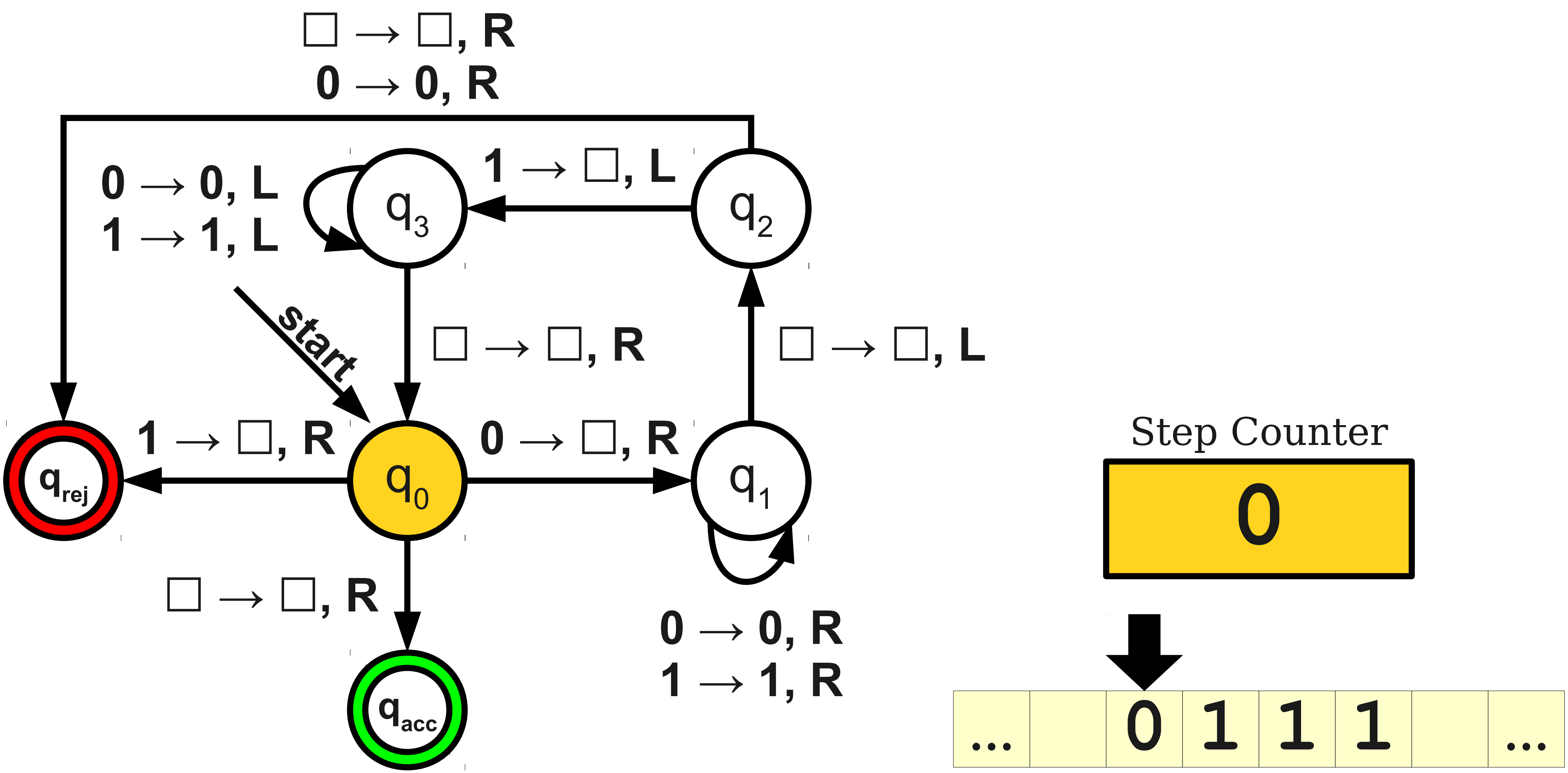
- A **step** of a Turing machine is one event where the TM takes a transition.





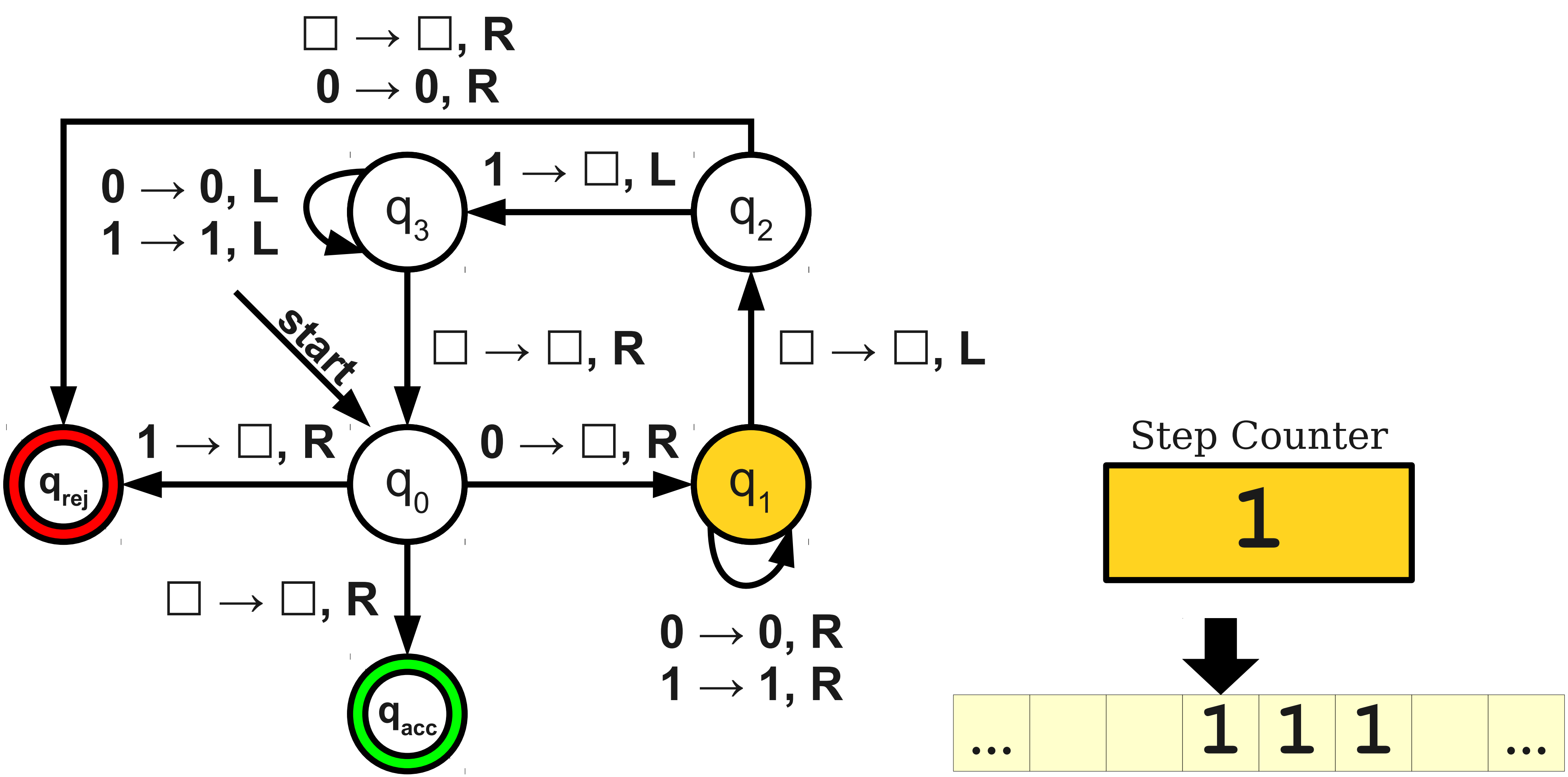
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



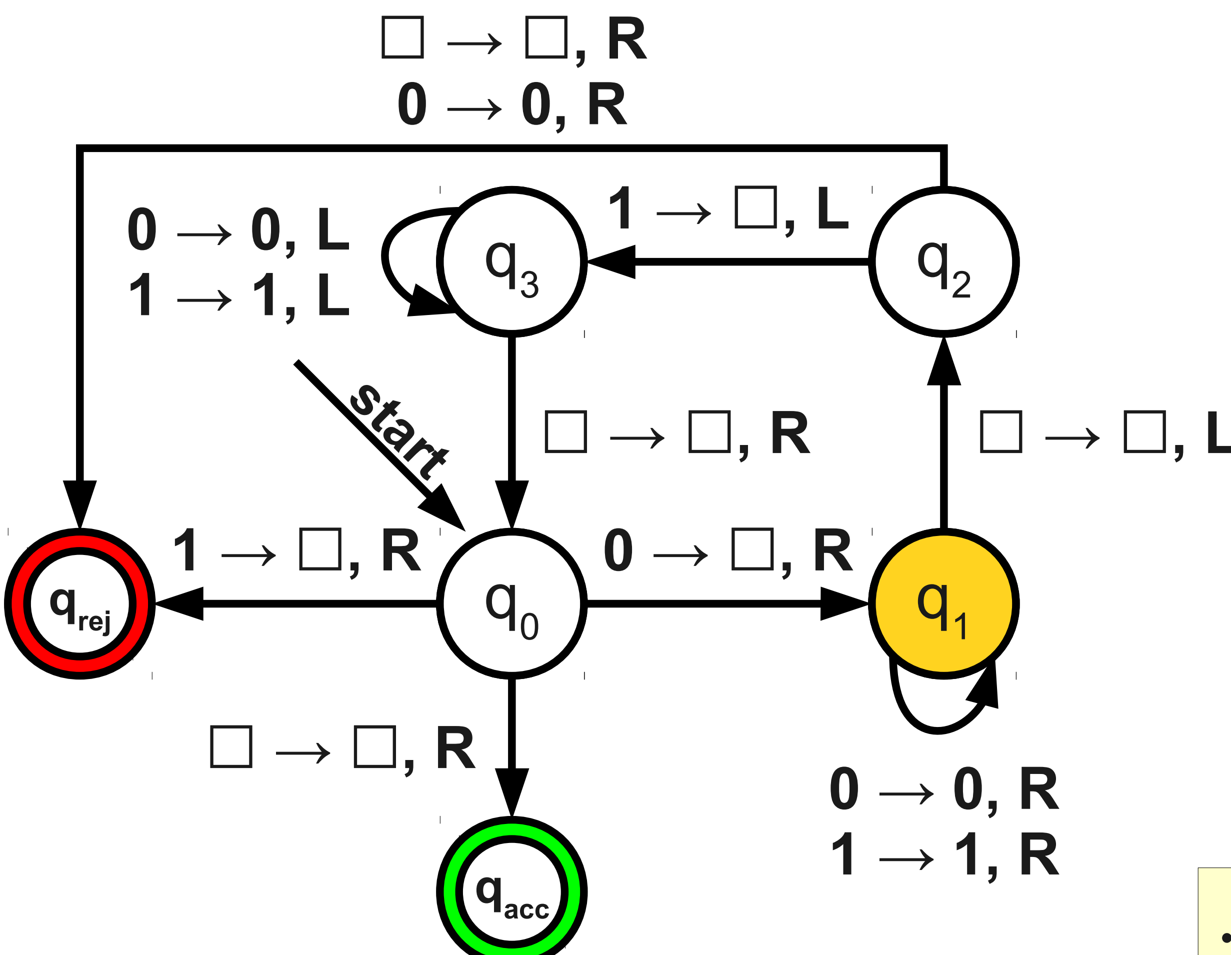
Time Complexity

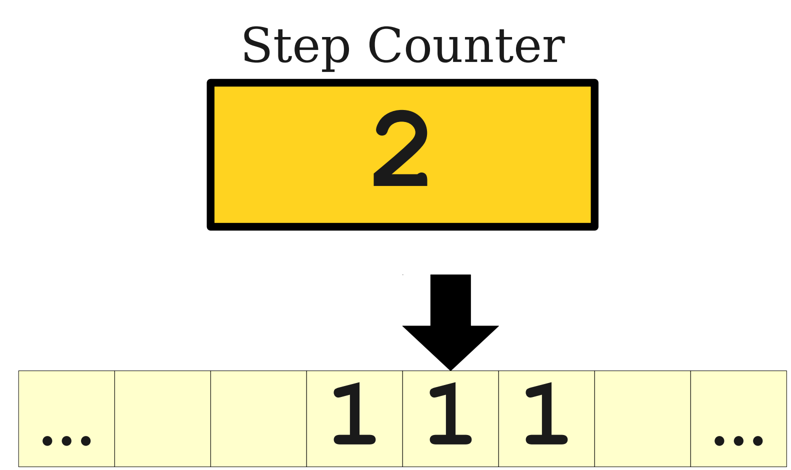
- A **step** of a Turing machine is one event where the TM takes a transition.



Time Complexity

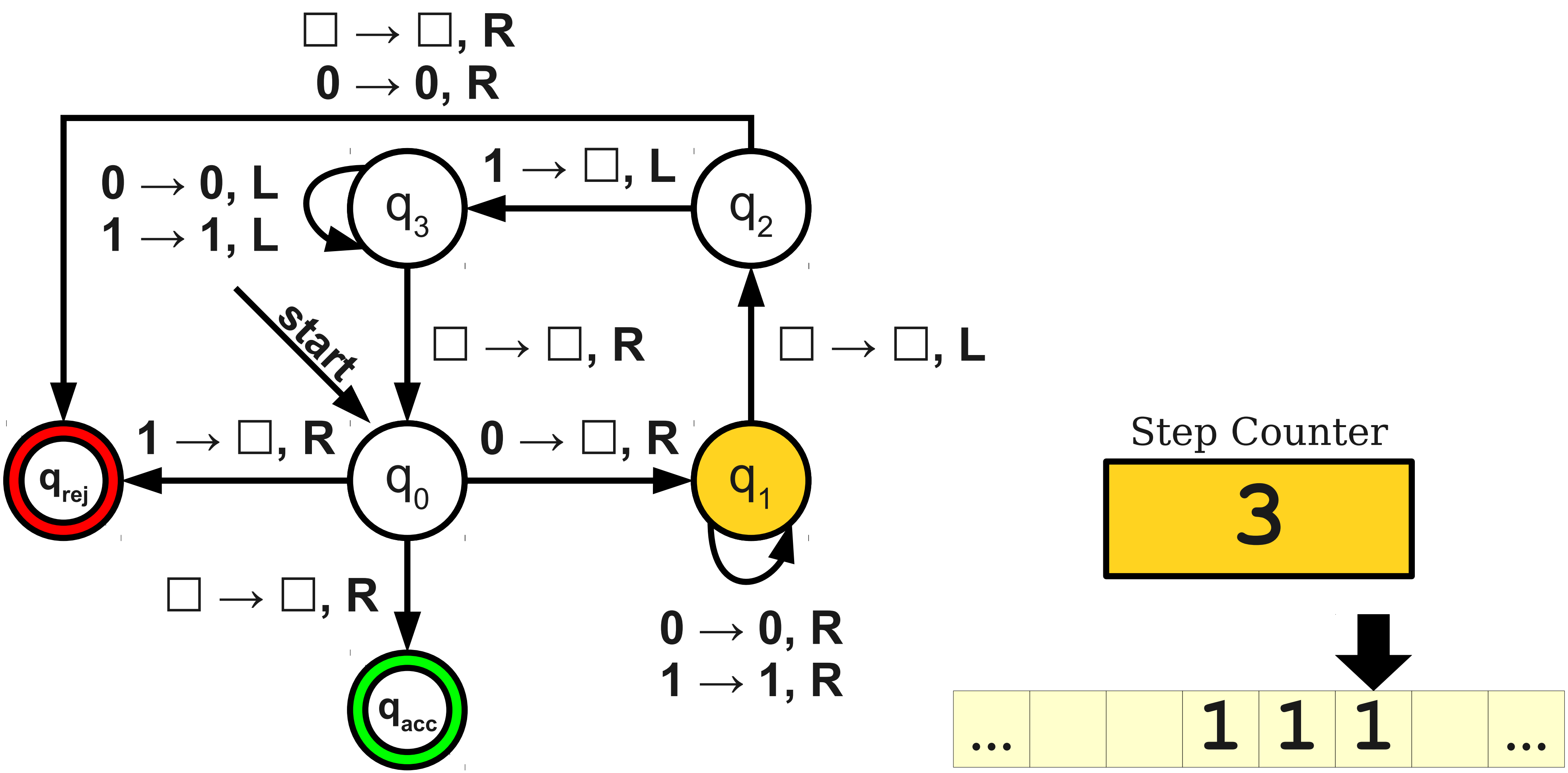
- A **step** of a Turing machine is one event where the TM takes a transition.





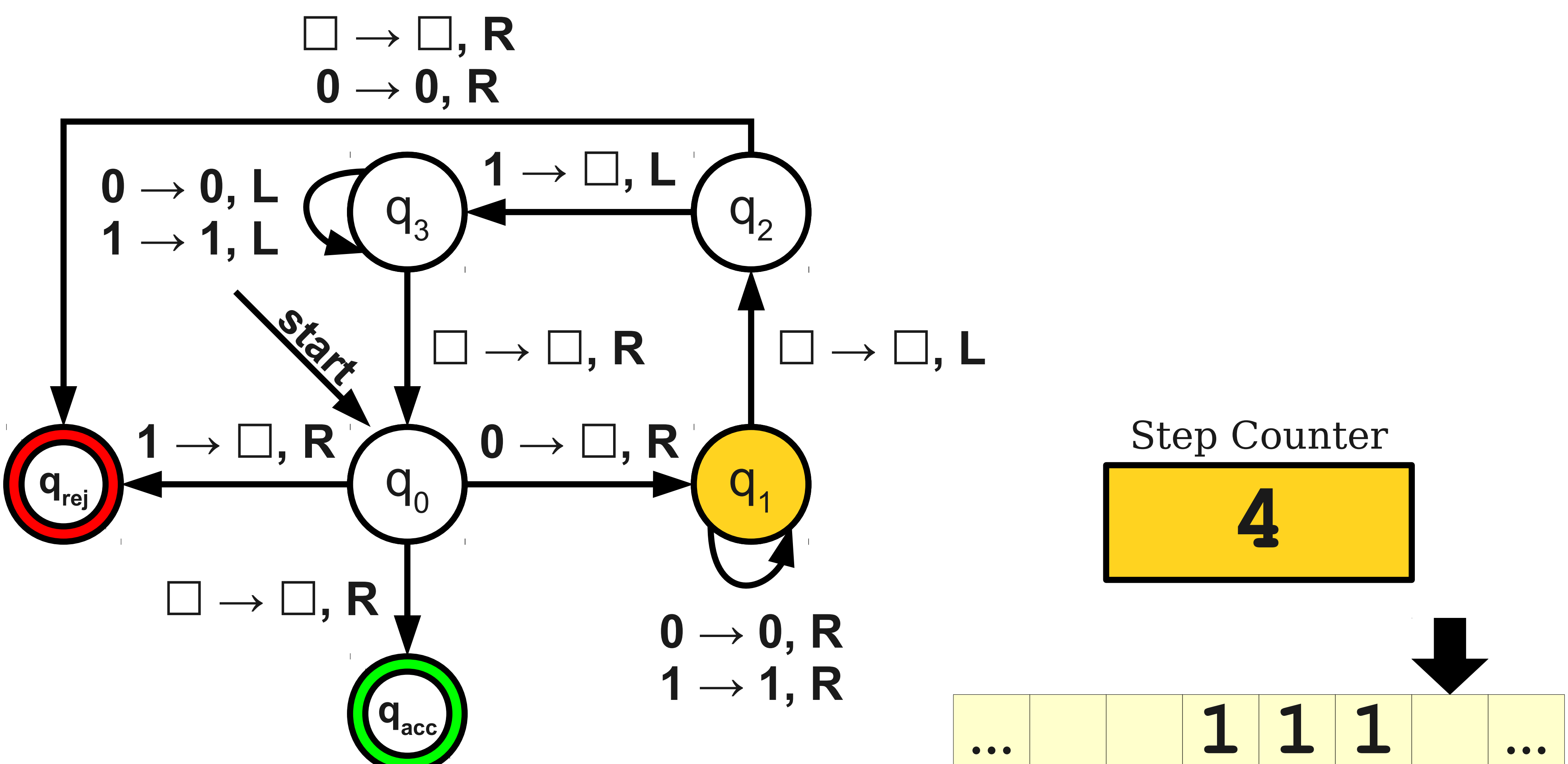
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



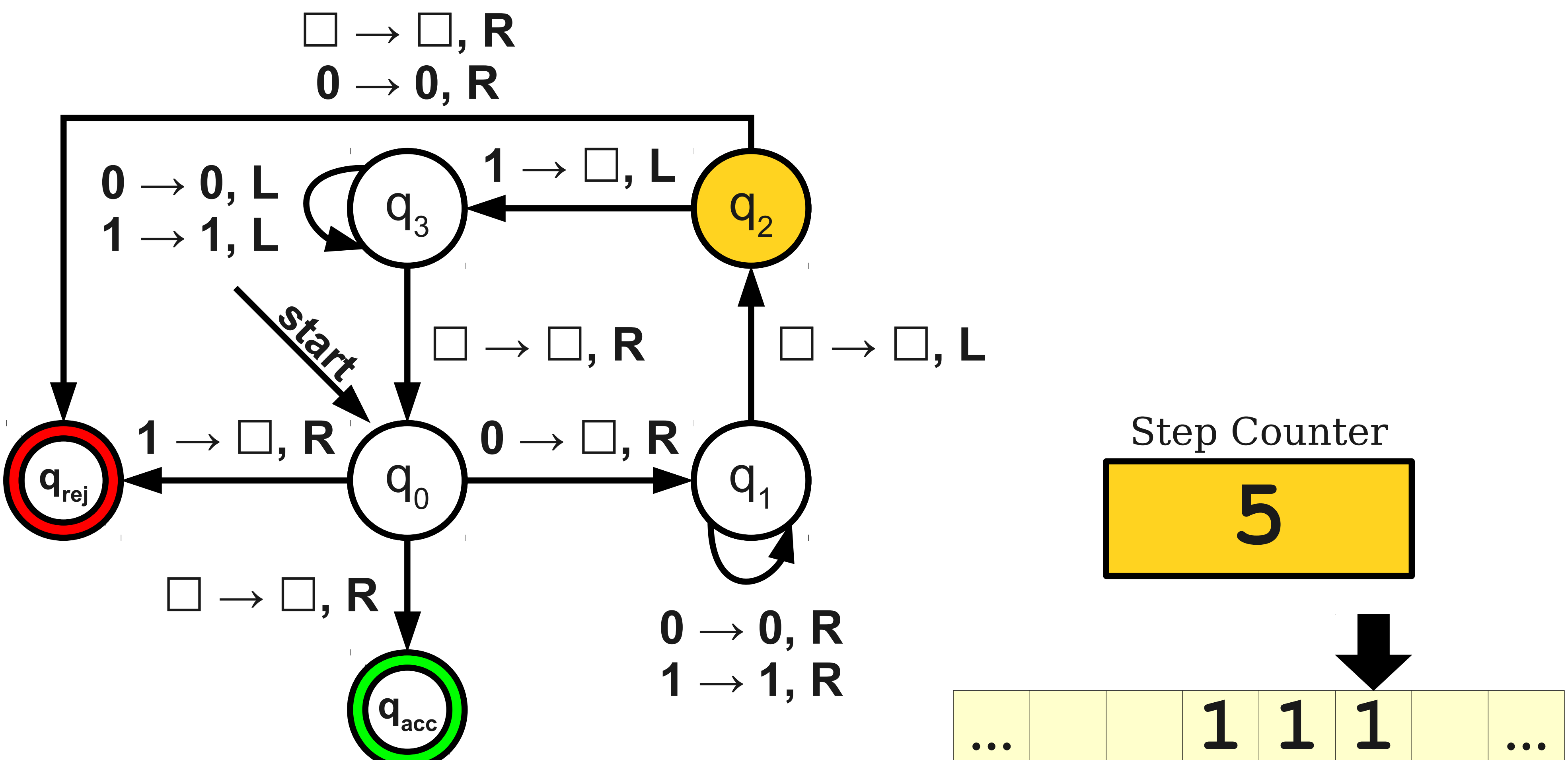
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



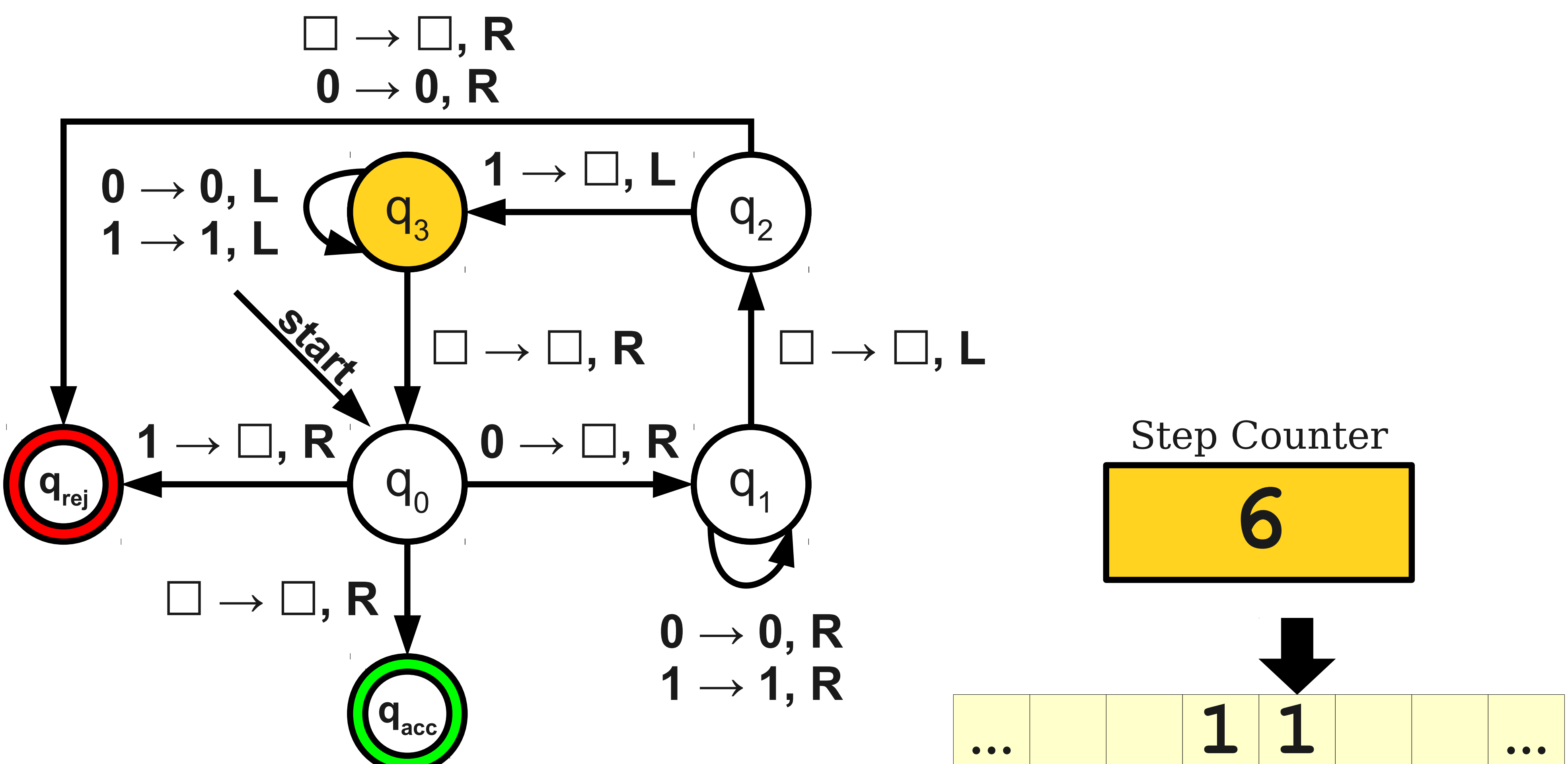
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



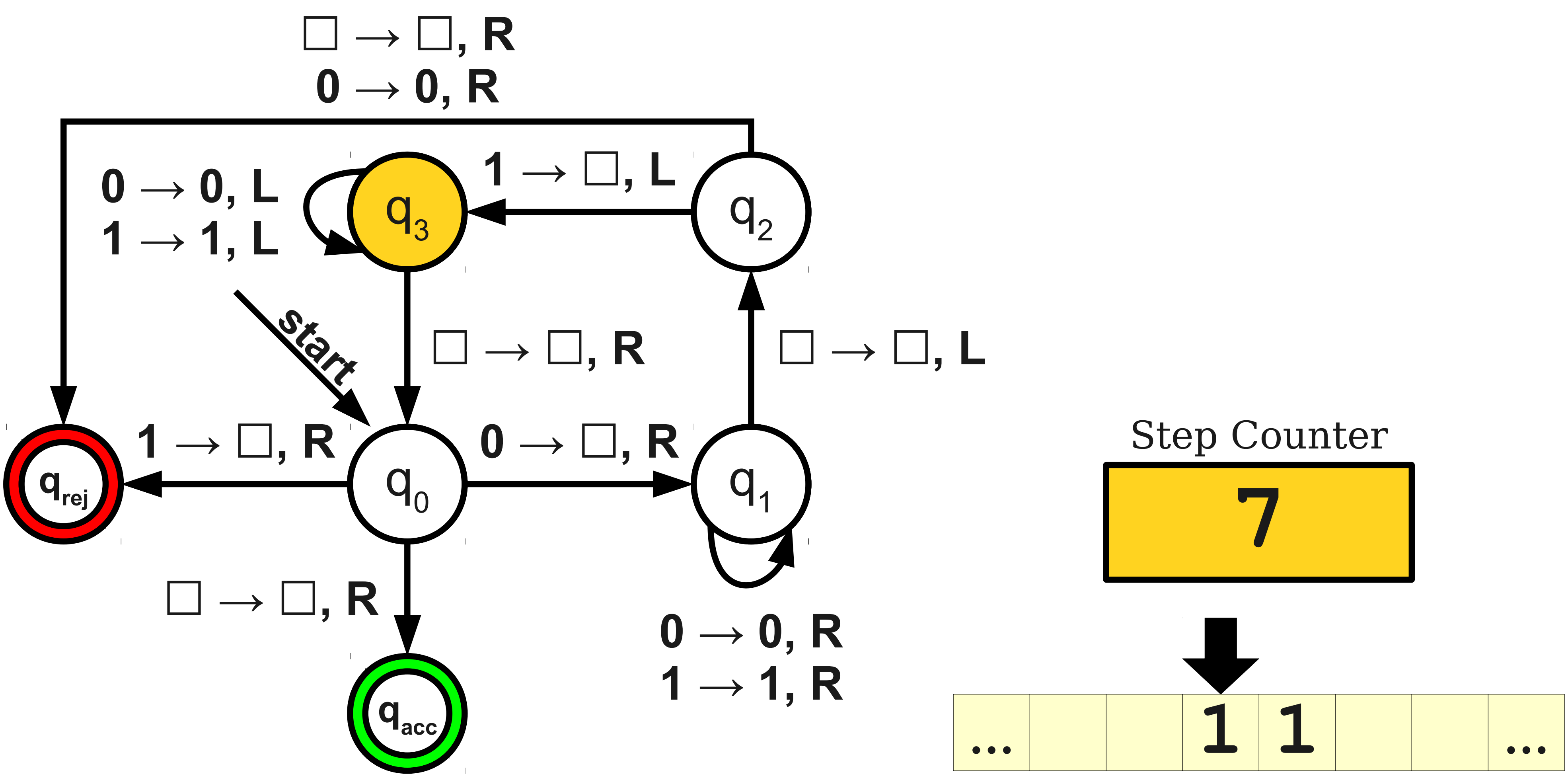
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



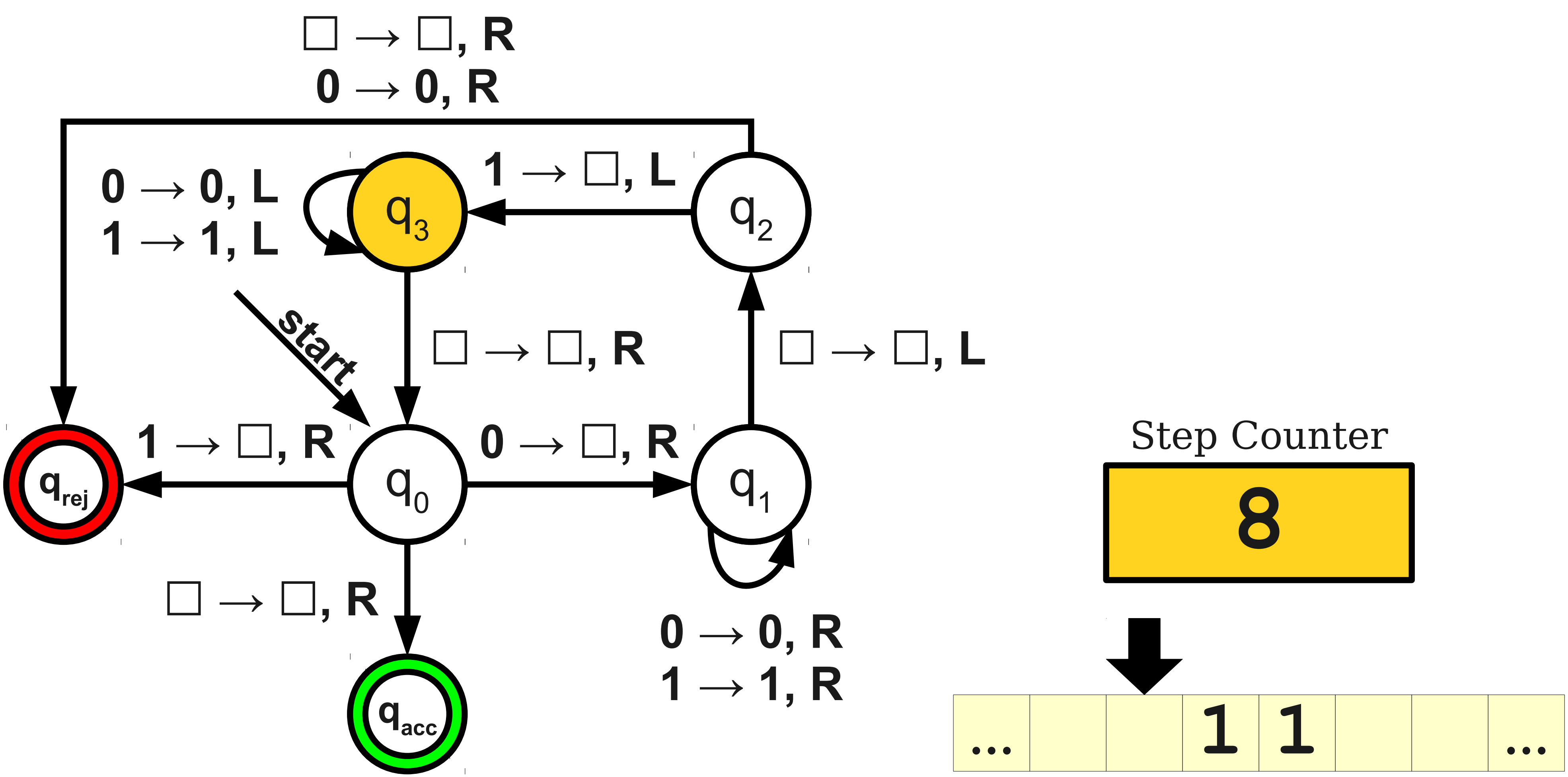
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.



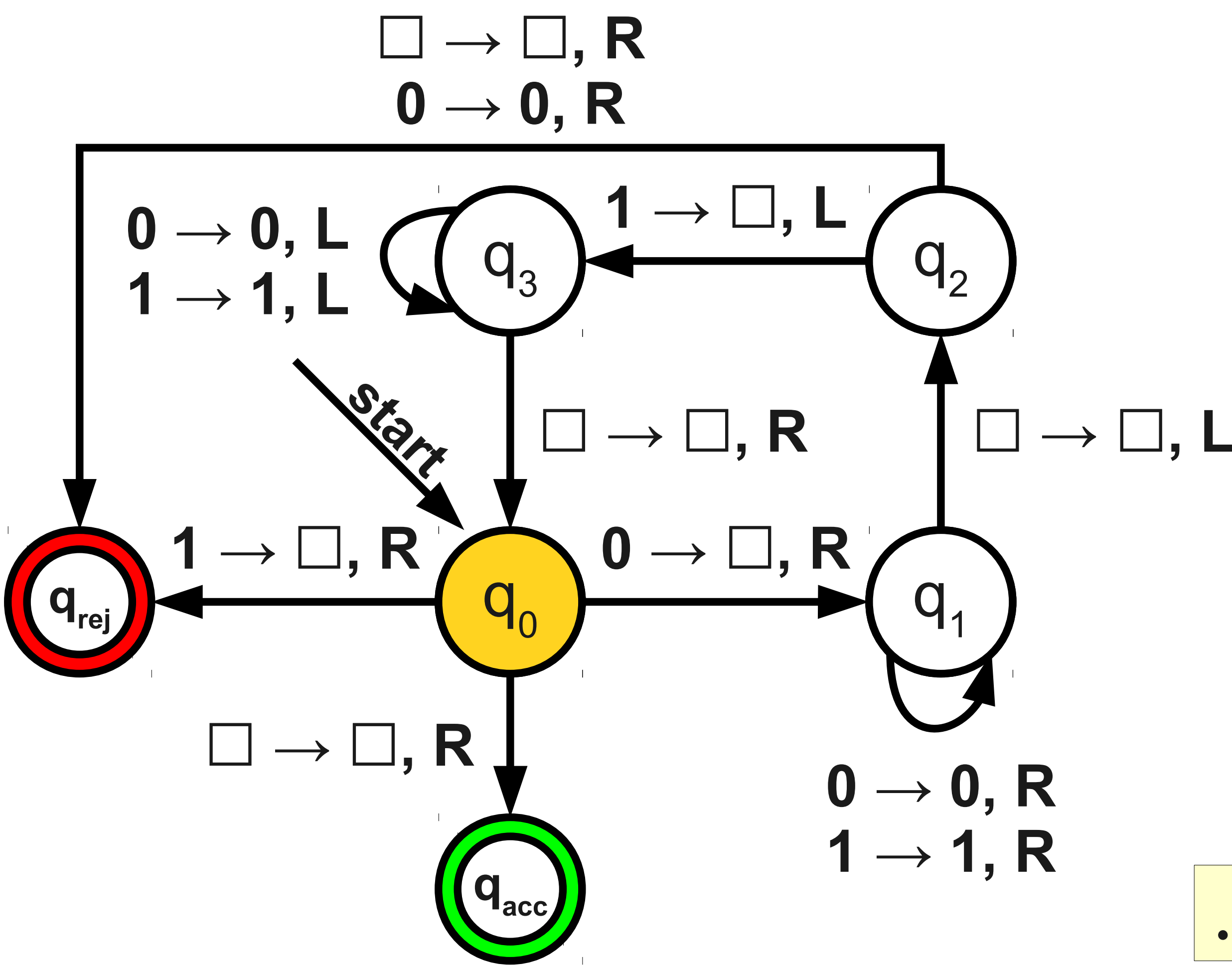
Time Complexity

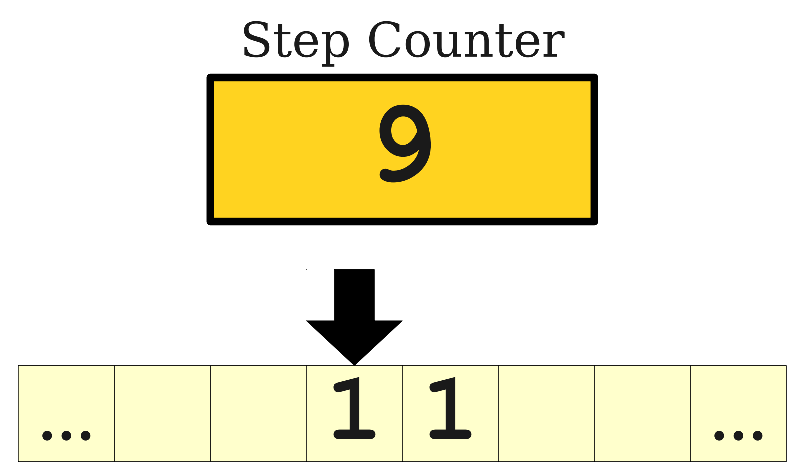
- A **step** of a Turing machine is one event where the TM takes a transition.



Time Complexity

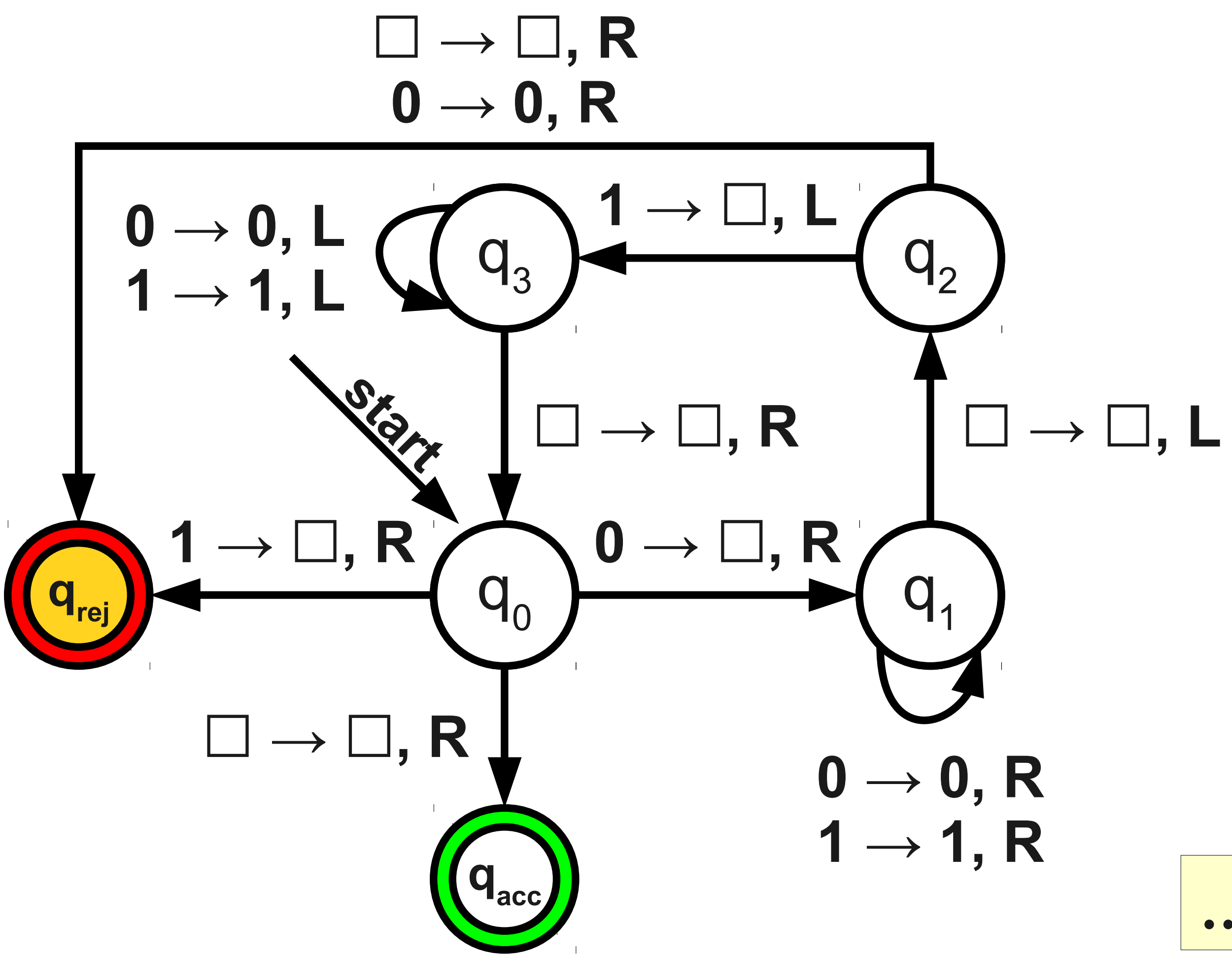
- A **step** of a Turing machine is one event where the TM takes a transition.

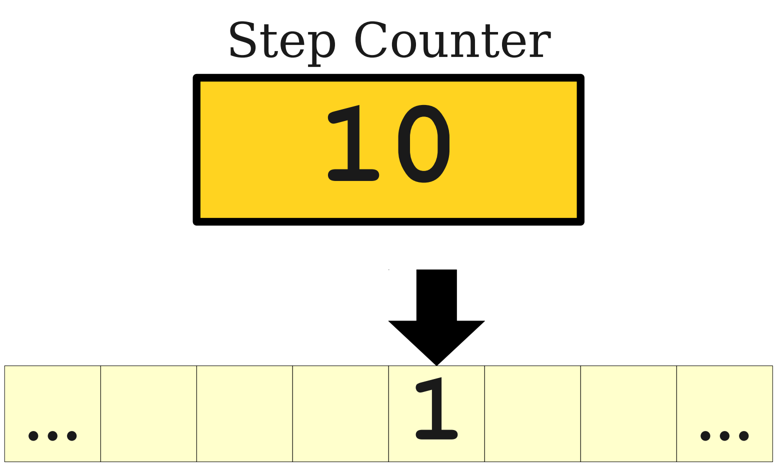




Time Complexity

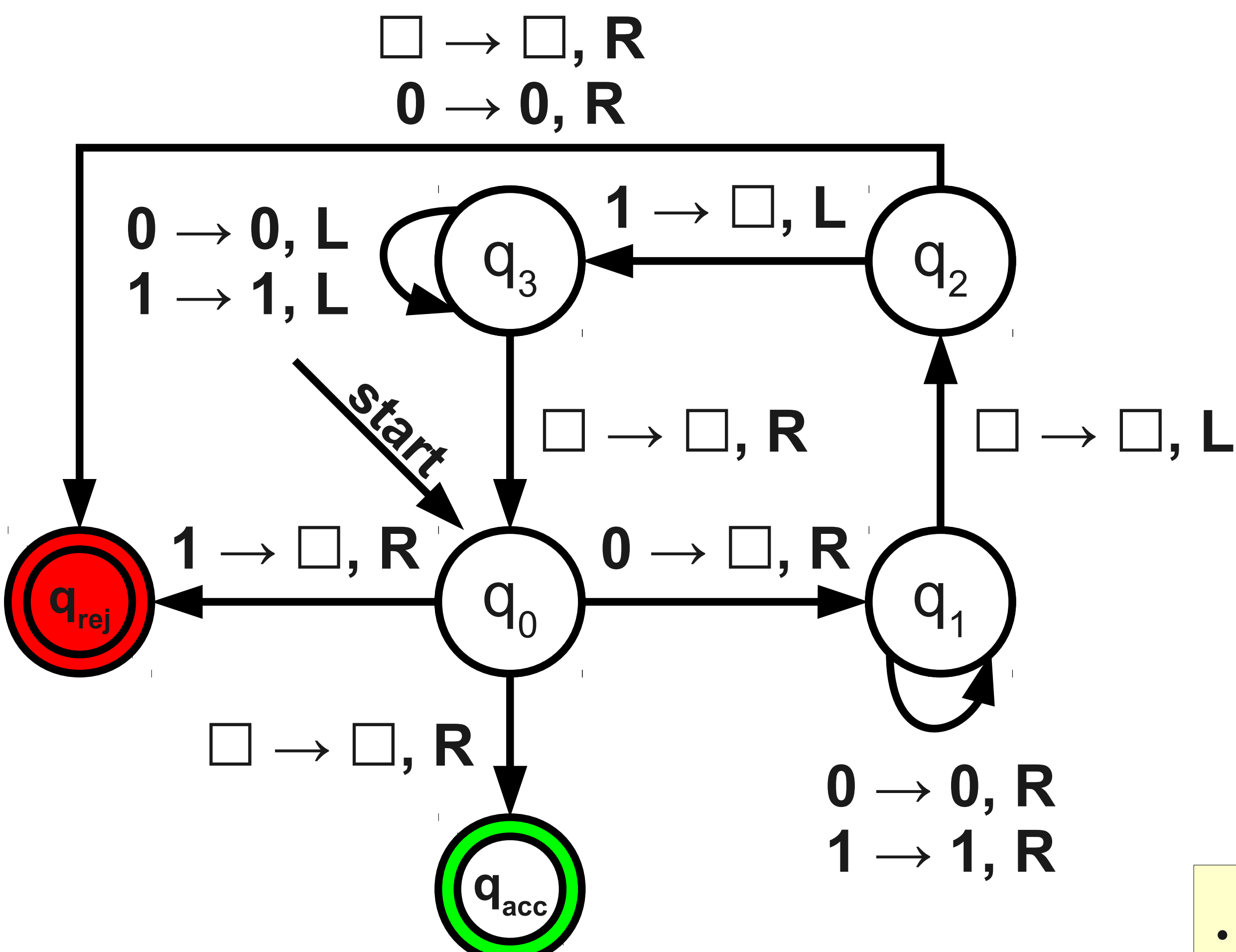
- A **step** of a Turing machine is one event where the TM takes a transition.

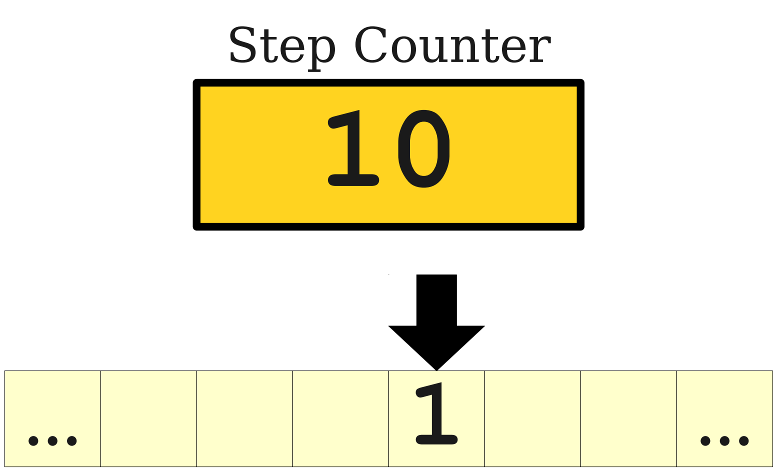




Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.





Time Complexity

- The number of steps a TM takes on some input is sensitive to
  - The structure of that input.
  - The length of the input.
- How can we come up with a consistent measure of a machine's runtime?

Time Complexity

- The **time complexity** of a TM M is a function (typically denoted $f(n)$) that measures the *worst-case* number of steps M takes on any input of length n.
  - By convention, n denotes the length of the input.
  - If M loops on some input of length k, then $f(k) = \infty$.
- The previous TM has a time complexity that is (roughly) proportional to $n^2 / 2$.
  - Difficult and utterly unrewarding exercise: compute the *exact* time complexity of the previous TM.

A Slight Problem

- Consider the following TM over $\Sigma = \{$`0`, `1`$\}$ for the language *BALANCE* $= \{ w \in \Sigma^* \mid w$ has the same number of `0`s and `1`s $\}$:
  - M = "On input w:
    - Scan across the tape until a `0` or `1` is found.
    - If none are found, accept.
    - If one is found, continue scanning until a matching `1` or `0` is found.
    - If none is found, reject.
    - Otherwise, cross off that symbol and repeat."
- What is the time complexity of M?

A Loss of Precision

- When considering *computability*, using high-level TM descriptions is perfectly fine.
- When considering *complexity*, high-level TM descriptions make it nearly impossible to precisely reason about the actual time complexity.
- What are we to do about this?

The Best We Can

M = "On input w:

- Scan across the tape until a `0` or `1` is found.
- If none are found, accept.
- If one is found, continue scanning until a matching `1` or `0` is found.
- If none are found, reject.
- Otherwise, cross off that symbol and repeat."

At most n steps.

At most 1 step.

At most n more steps.

At most 1 step

At most n steps to get back to the start of the tape.

At most $n/2$ loops

+

At most $3n + 2$ steps.

× At most $n/2$ loops.

At most $3n^2 / 2 + n$ steps.

An Easier Approach

- In complexity theory, we rarely need an exact value for a TM's time complexity.
- Usually, we are curious with the long-term growth rate of the time complexity.
- For example, if the time complexity is $3n + 5$, then doubling the length of the string roughly doubles the worst-case runtime.
- If the time complexity is $2^n - n^2$, since 2^n grows much more quickly than n^2, for large values of n, increasing the size of the input by 1 doubles the worst-case running time.

Big-O Notation

- Ignore *everything* except the dominant growth term, including constant factors.
- Examples:
  - $4n + 4 = \mathbf{O}(\boldsymbol{n})$
  - $137n + 271 = \mathbf{O}(\boldsymbol{n})$
  - $n^2 + 3n + 4 = \mathbf{O}(\boldsymbol{n^2})$
  - $2^n + n^3 = \mathbf{O}(\mathbf{2}^{\boldsymbol{n}})$
  - $137 = \mathbf{O(1)}$
  - $n^2 \log n + \log^5 n = \mathbf{O}(\boldsymbol{n^2} \mathbf{\log} \boldsymbol{n})$

Big-O Notation, Formally

- Let $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$.
- Then $f(n) = \mathrm{O}(g(n))$ iff there exist constants $c \in \mathbb{R}$ and $n_0 \in \mathbb{N}$ such that

For any $n \geq n_0$, $f(n) \leq cg(n)$

- Intuitively, as n gets "large" (greater than n_0), $f(n)$ is bounded from above by some multiple (determined by c) of $g(n)$.

Properties of Big-O Notation

- **Theorem**: If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.
  - Intuitively: If you run two programs one after another, the big-O of the result is the big-O of the sum of the two runtimes.
- **Theorem**: If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n)f_2(n) = O(g_1(n)g_2(n))$.
  - Intuitively: If you run one program some number of times, the big-O of the result is the big-O of the program times the big-O of the number of iterations.
- This makes it substantially easier to analyze time complexity, though we do lose some precision.

Life is Easier with Big-O

M = "On input w:

- Scan across the tape until a `0` or `1` is found.
- If none are found, accept.
- If one is found, continue scanning until a matching `1` or `0` is found.
- If none is found, reject.
- Otherwise, cross off that symbol and repeat."

O(n) steps

O(1) steps

O(n) steps

O(1) steps

\+ O(n) steps

O(n) loops

O(n) steps

× O(n) loops

O(n^2) steps

Next Time

- **P**
  - What problems can be decided efficiently?
- **Polynomial-Time Reductions**
  - Constructing efficient algorithms.
- **NP**
  - What can we *verify* quickly?