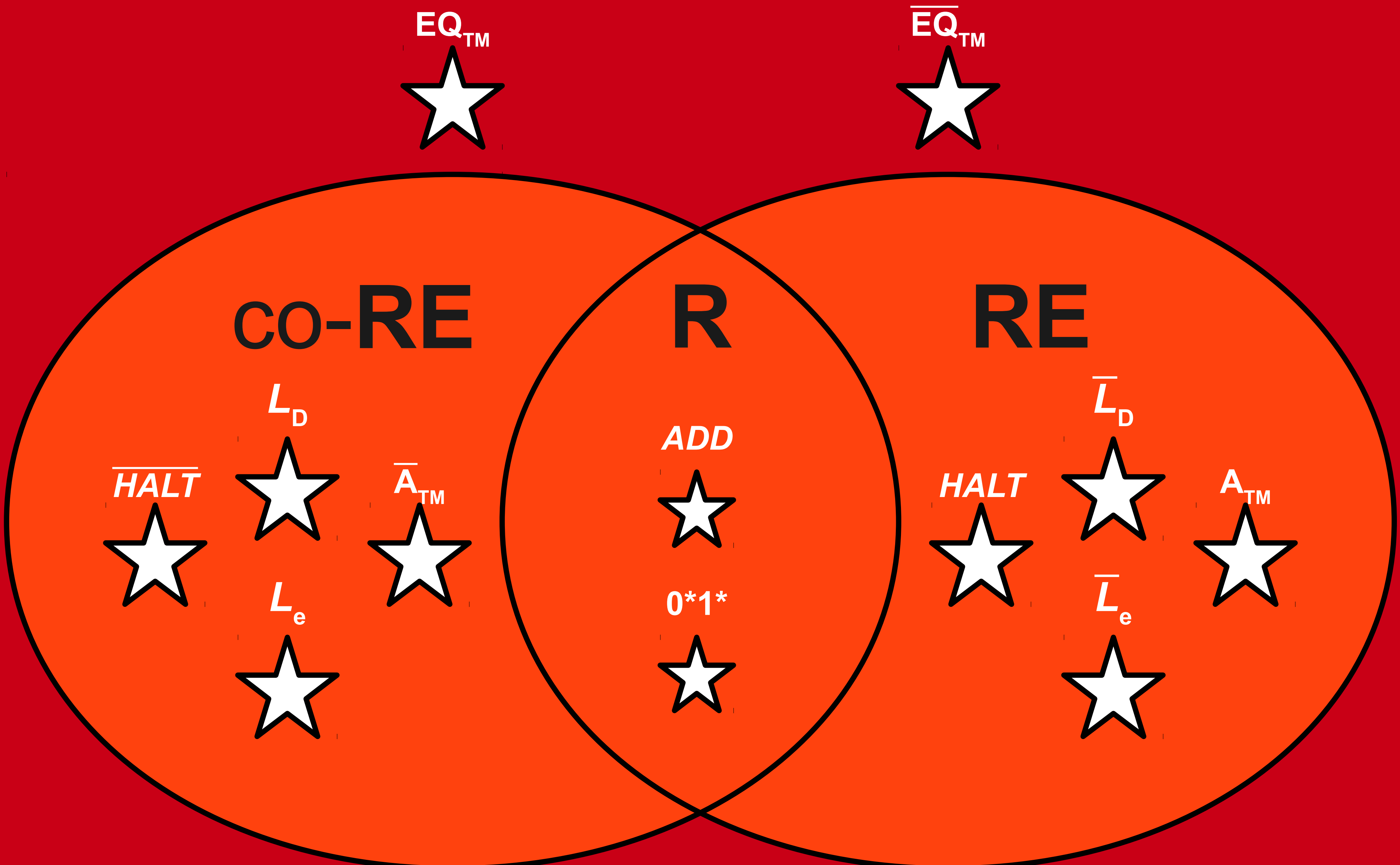
Complexity Theory

Part I

Problem set 7 due right now using a late period

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”

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.
  - $\forall x.\ x + 1 \neq 0$
  - $\forall x.\ \forall y.\ (x + 1 = y + 1 \rightarrow x = y)$
  - $\forall x.\ x + 0 = x$
  - $\forall x.\ \forall y.\ (x + y) + 1 = x + (y + 1)$
  - $\forall x.\ ((P(0) \land \forall y.\ (P(y) \rightarrow P(y + 1))) \rightarrow \forall x.\ P(x)$
- 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 $c = 1$.

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

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

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

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

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

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

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

The Limits of Decidability

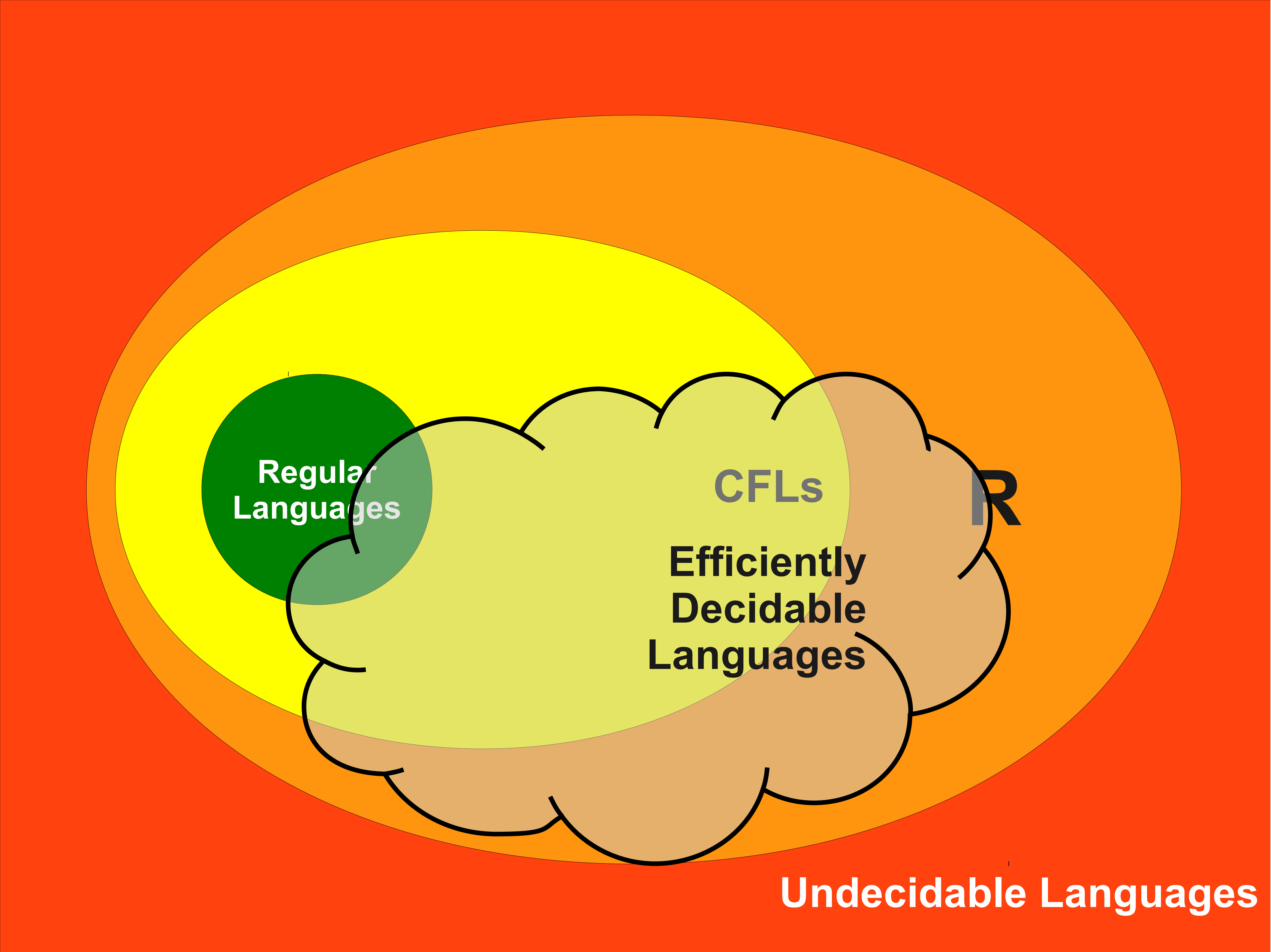
- The fact that a problem is decidable does not mean that it is *feasibly* decidable.
- In **computability theory**, we ask the question

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
CFLs
R
Efficiently Decidable Languages
Undecidable Languages

The Setup

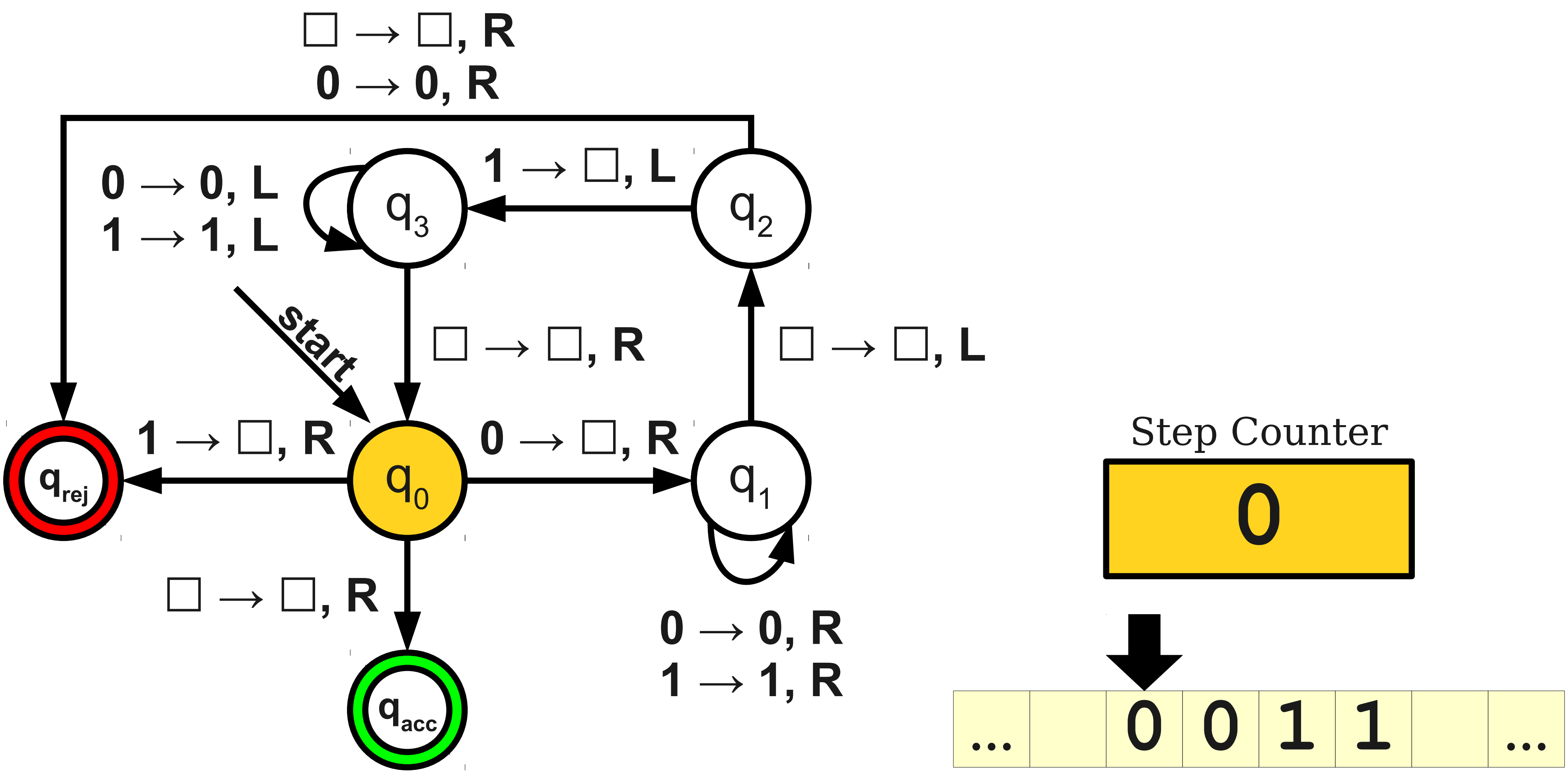
- In order to study computability, we needed to answer these questions:
  - What is “computation?”
  - What is a “problem?”
  - What does it mean to “solve” a problem?
- To study complexity, we need to answer these questions:
  - What does “complexity” even mean?
  - What is an “efficient” solution to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

Measuring Complexity

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

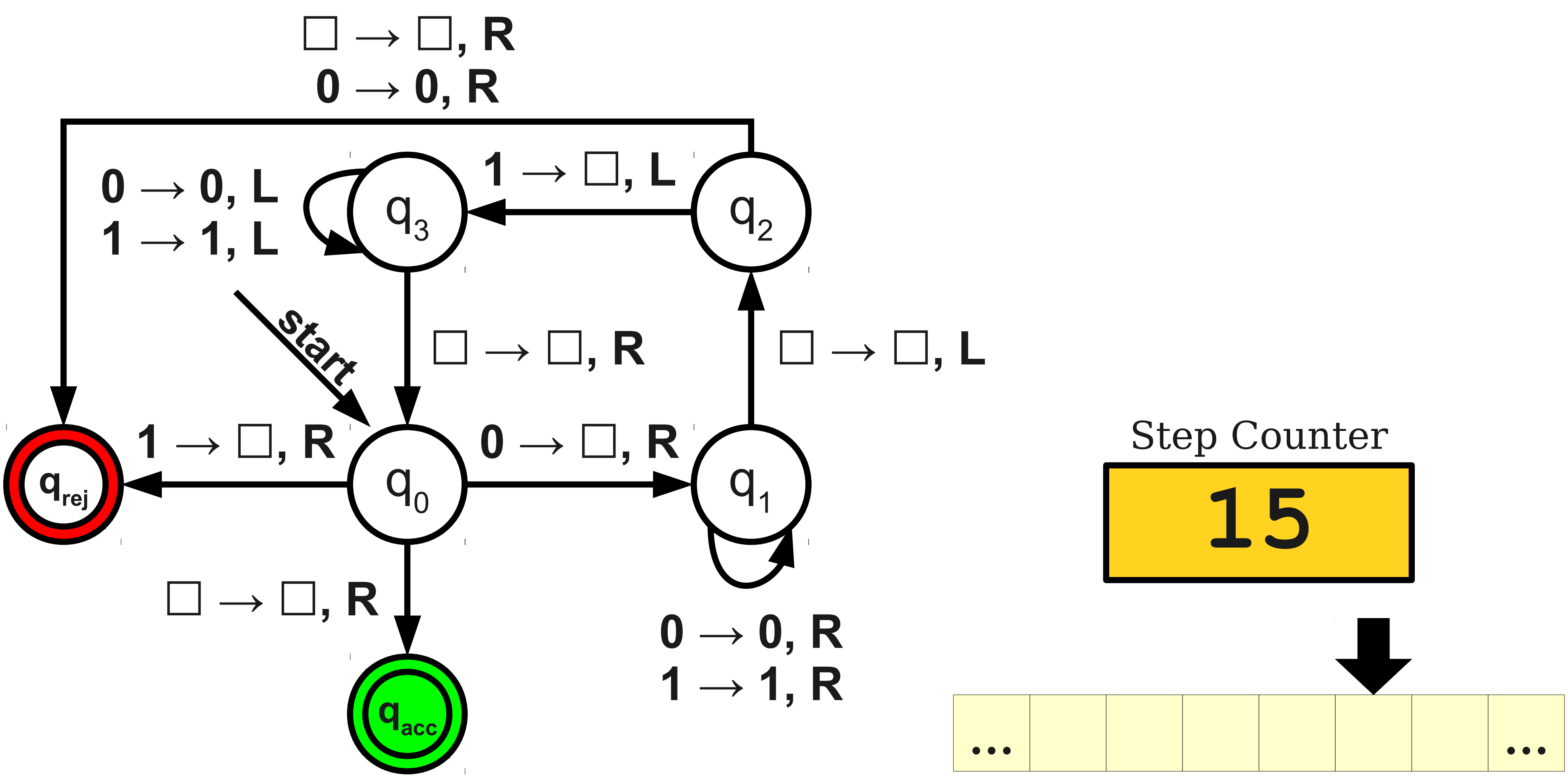
Time Complexity

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



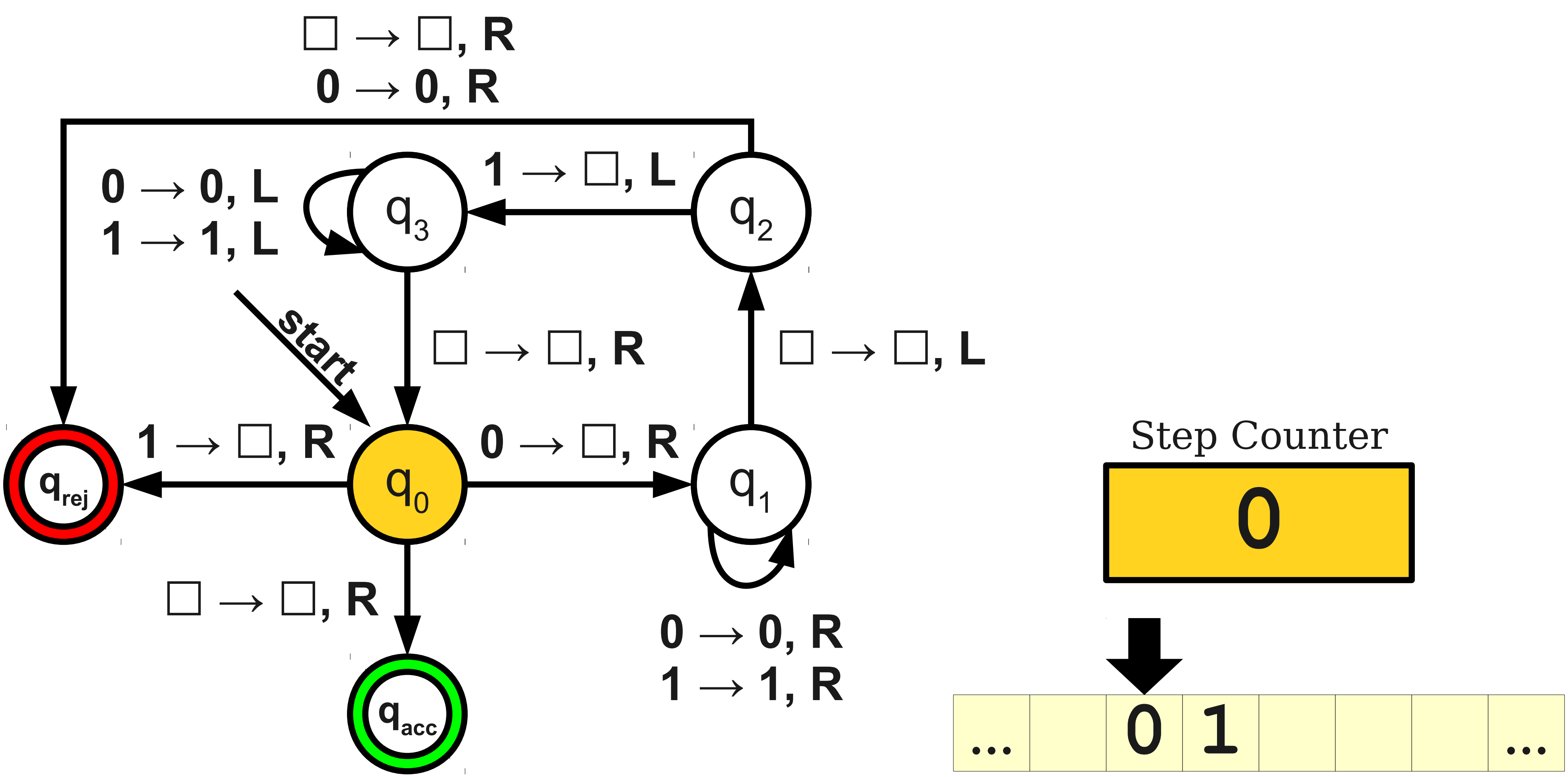
Time Complexity

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



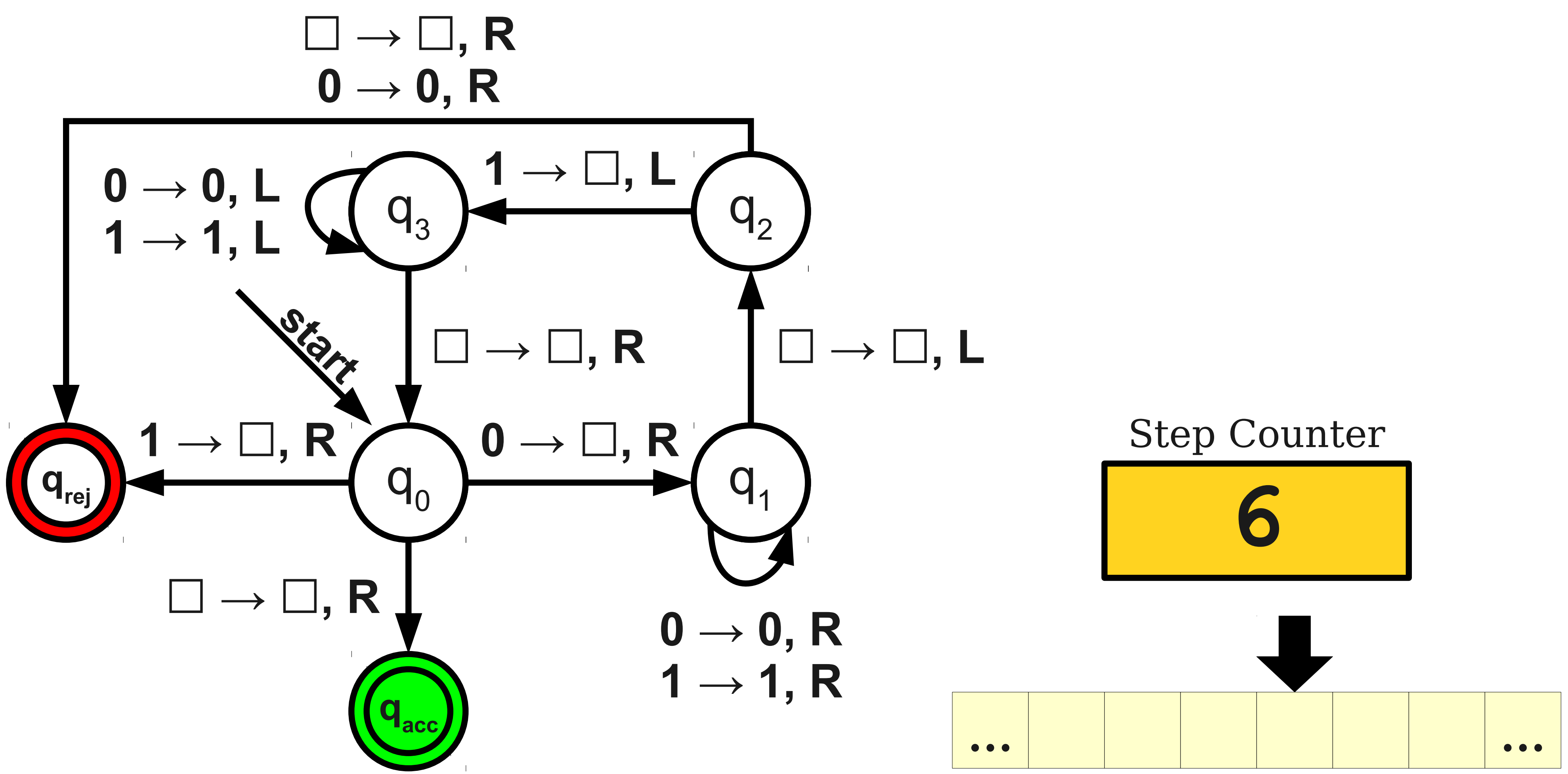
Time Complexity

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



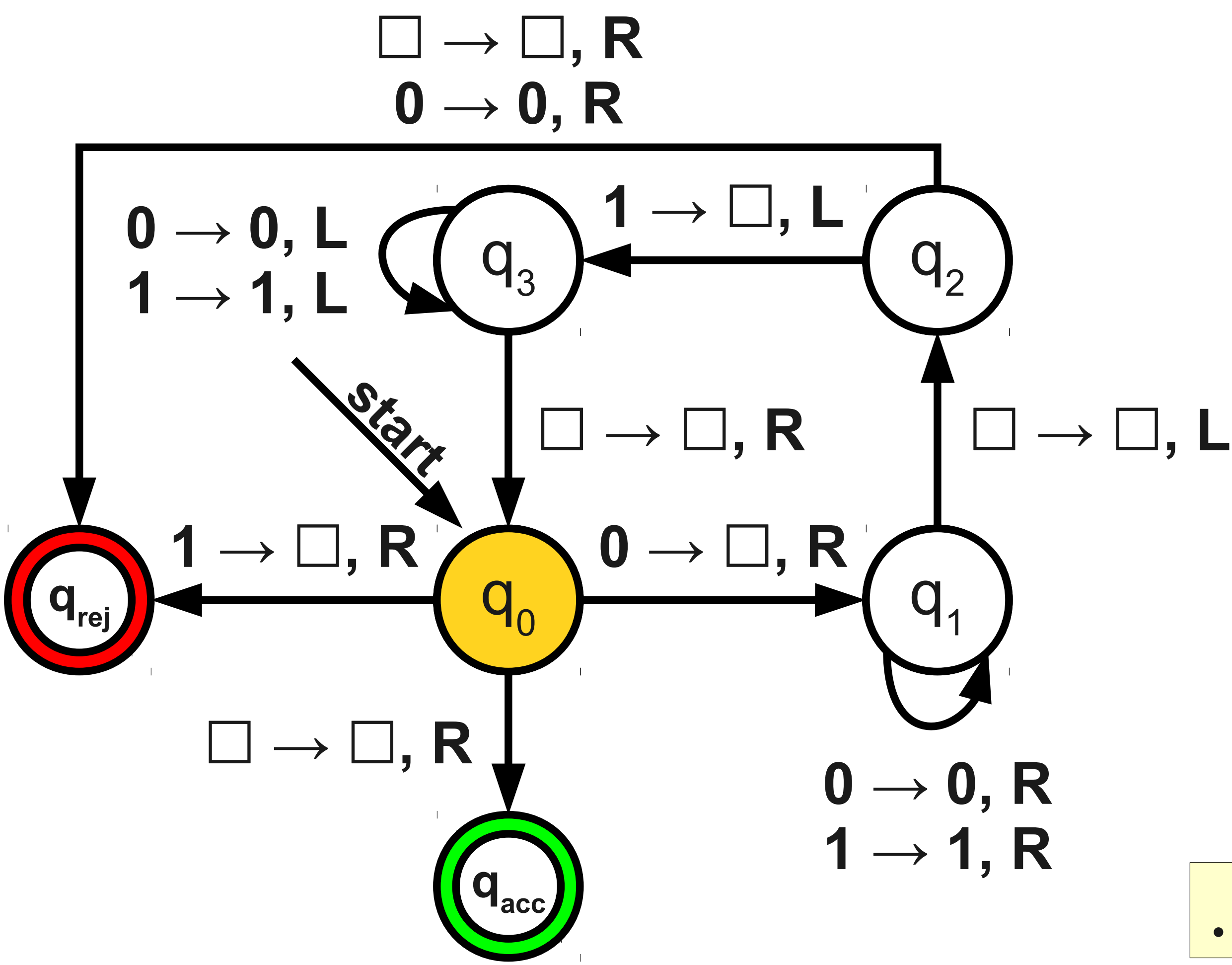
Time Complexity

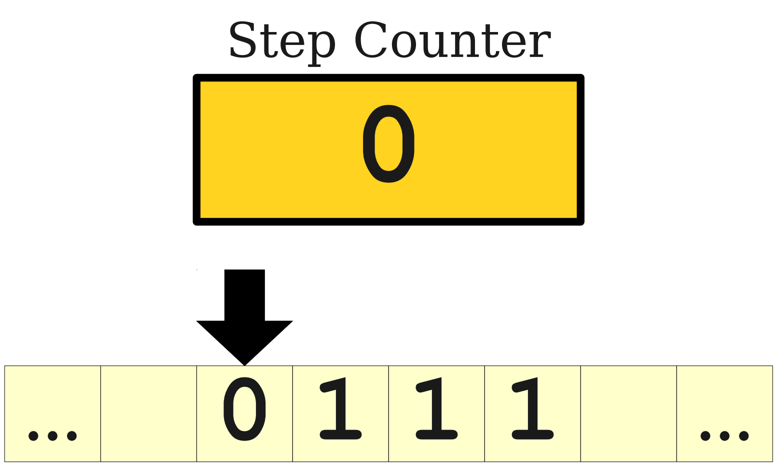
- A **step** of a Turing machine is one event where the TM takes a transition.



Time Complexity

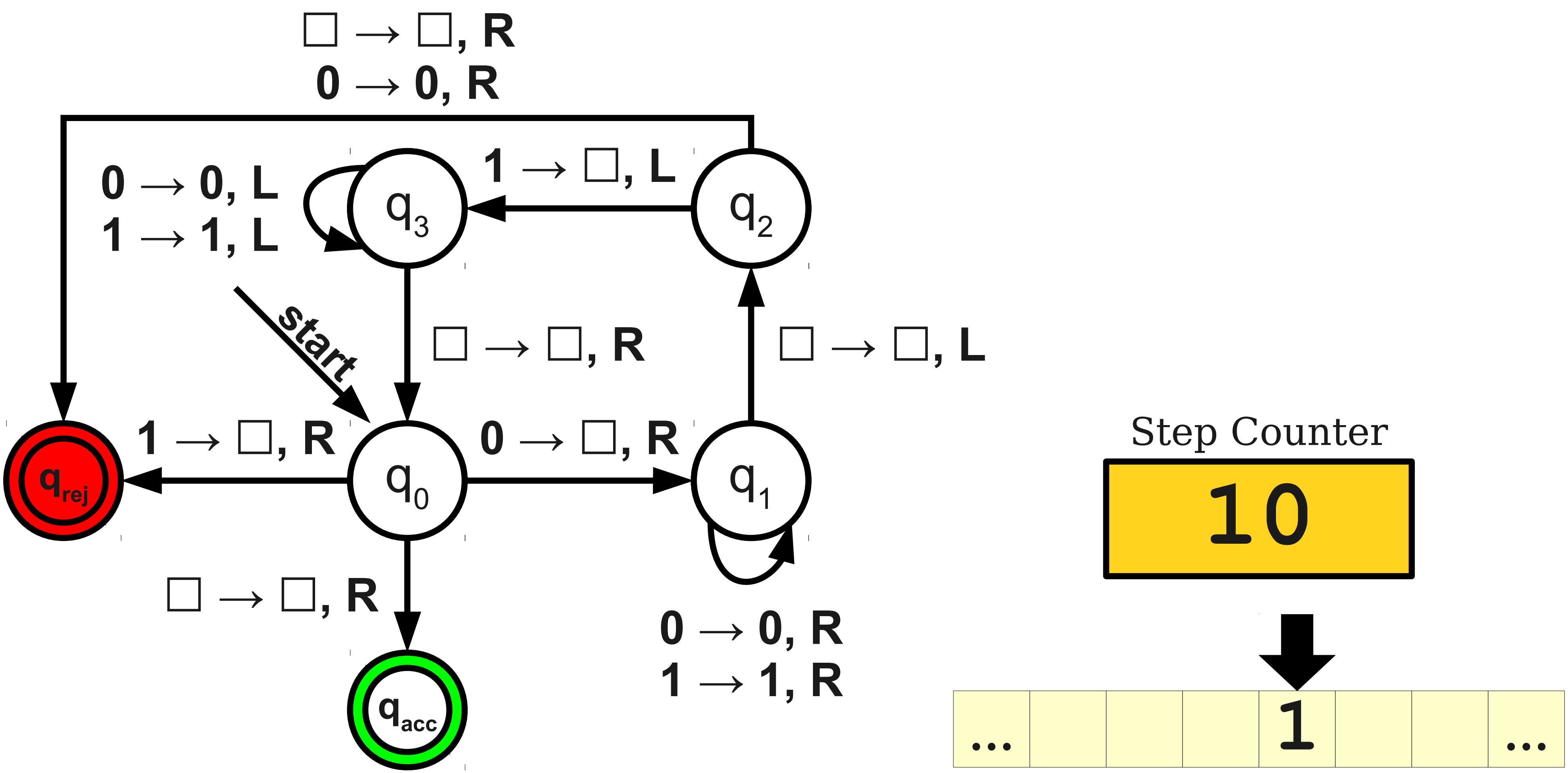
- A **step** of a Turing machine is one event where the TM takes a transition.





Time Complexity

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



Time Complexity

- 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 denoting the *worst-case* number of steps M takes on any input of length n.
  - By convention, n denotes the length of the input.
  - Assume we're only dealing with deciders, so there's no need to handle looping TMs.
- 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. **At most *n* steps.**
- If none are found, accept. **At most 1 step.**
- If one is found, continue scanning until a matching `1` or `0` is found. **At most *n* more steps.**
- If none are found, reject. **At most 1 step**
- Otherwise, cross off that symbol and repeat." **At most *n* steps to get back to the start of the tape.**

At most *n*/2 loops

\+

At most 3*n* + 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. That tells us how *scalable* our algorithm will be.
- 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

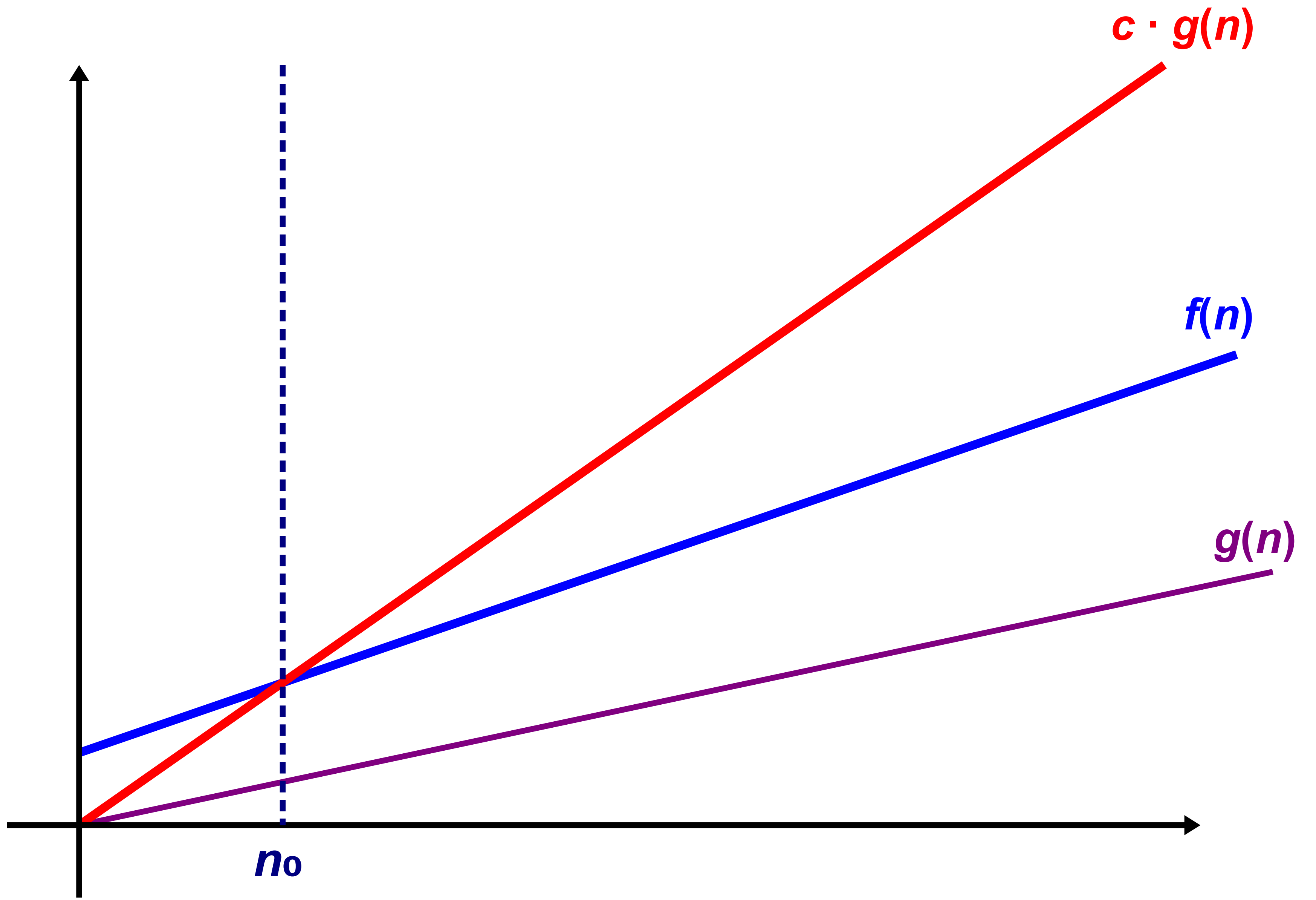
- Formally speaking, let f, $g : \mathbb{N} \to \mathbb{N}$.
- We say **$f(n) = O(g(n))$** iff

There are constants n_0, c such that

$$\forall n \in \mathbb{N}.\ (n \geq n_0 \to f(n) \leq c \cdot g(n))$$

- Intuitively, when n gets "sufficiently large" (i.e. greater than n_0), $f(n)$ is bounded from above by some constant multiple (specifically, c) of $g(n)$.

$$f(n) = O(g(n))$$



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

A Quick Note

- Time complexity depends on the model of computation.
  - A computer can binary search over a sorted array in time O(log n).
  - A TM has to spend at least n time doing this, since it has no random access.
- For now, assume that the slowdown going from a computer to a TM or vice-versa is not “too bad.”

The Story So Far

- We now have a definition of the runtime of a TM.
- We can use big-O notation to measure the relative growth rates of different runtimes.
- **Big question:** How do we define efficiency?

Time-Out For Announcements!

Problem Set 6 Graded

- All Problem Set 6's have been graded. Late submissions will be returned at the end of lecture today.

A Question from Last Time

“Aren't there some cases where we can know a TM is infinite looping? Couldn't we modify the U_{TM} so it keeps a record of IDs and then if it sees the same one twice know it was in a loop? This doesn't guarantee to find all loops, but would it be useful?”

Back to CS103!

What is an efficient algorithm?

Searching Finite Spaces

- Many decidable problems can be solved by searching over a large but finite space of possible options.
- Searching this space might take a staggeringly long time, but only finite time.
- From a decidability perspective, this is totally fine.
- From a complexity perspective, this is totally unacceptable.

A Sample Problem

4	3	11	9	7	13	5	6	1	12	2	8	0	10

Goal: Find the length of the longest increasing subsequence of this sequence.

A Sample Problem

4	3	11	9	7	13	5	6	1	12	2	8	0	10

Longest so far:

4	11

How many different subsequences are there in a sequence of *n* elements? **2^n**

How long does it take to check each subsequence? **$O(n)$** time.

Runtime is around **$O(n \cdot 2^n)$**.

A Sample Problem

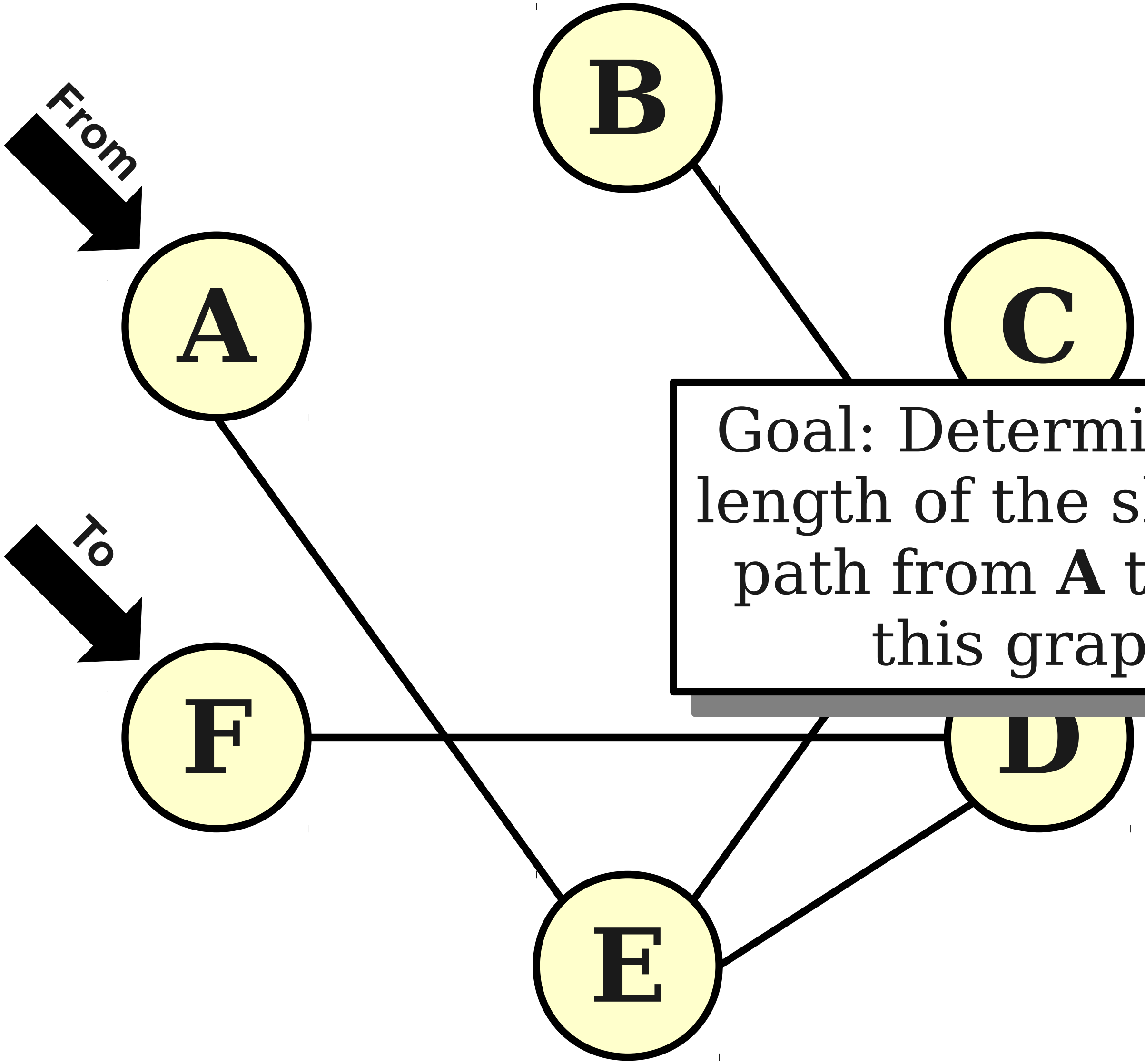
4	3	11	9	7	13	5	6	1	12	2	8	0	10

1	1	2	2	2	3	2	3	1	4	2	4	1	5

How many elements of the sequence do we have to look at when considering the kth element of the sequence? $\boldsymbol{k - 1}$

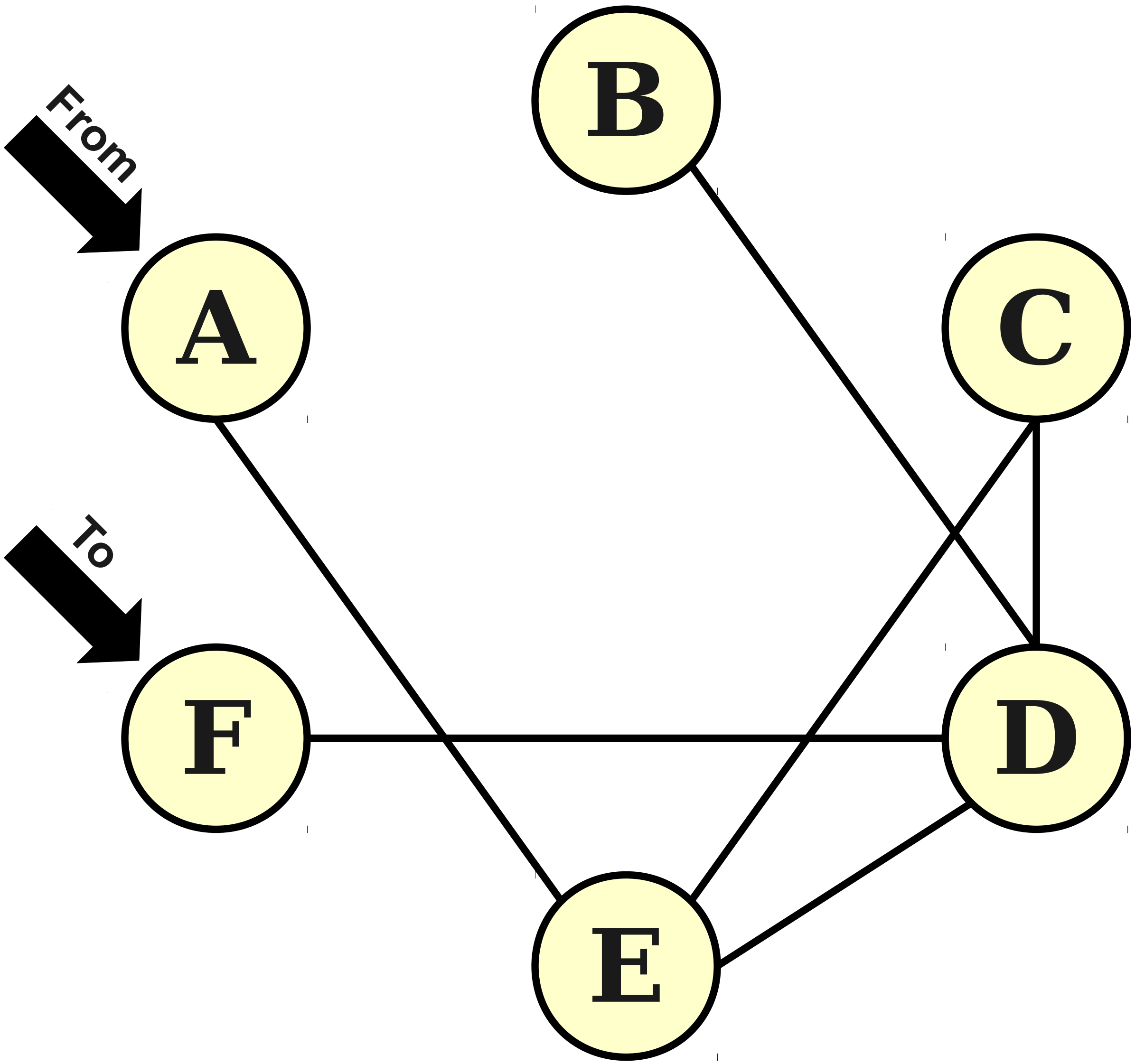
Total runtime is
$1 + 2 + \ldots + (n - 1) = \mathbf{O}(\boldsymbol{n^2})$

Another Problem



Goal: Determine the length of the shortest path from **A** to **F** in this graph.

Another Problem

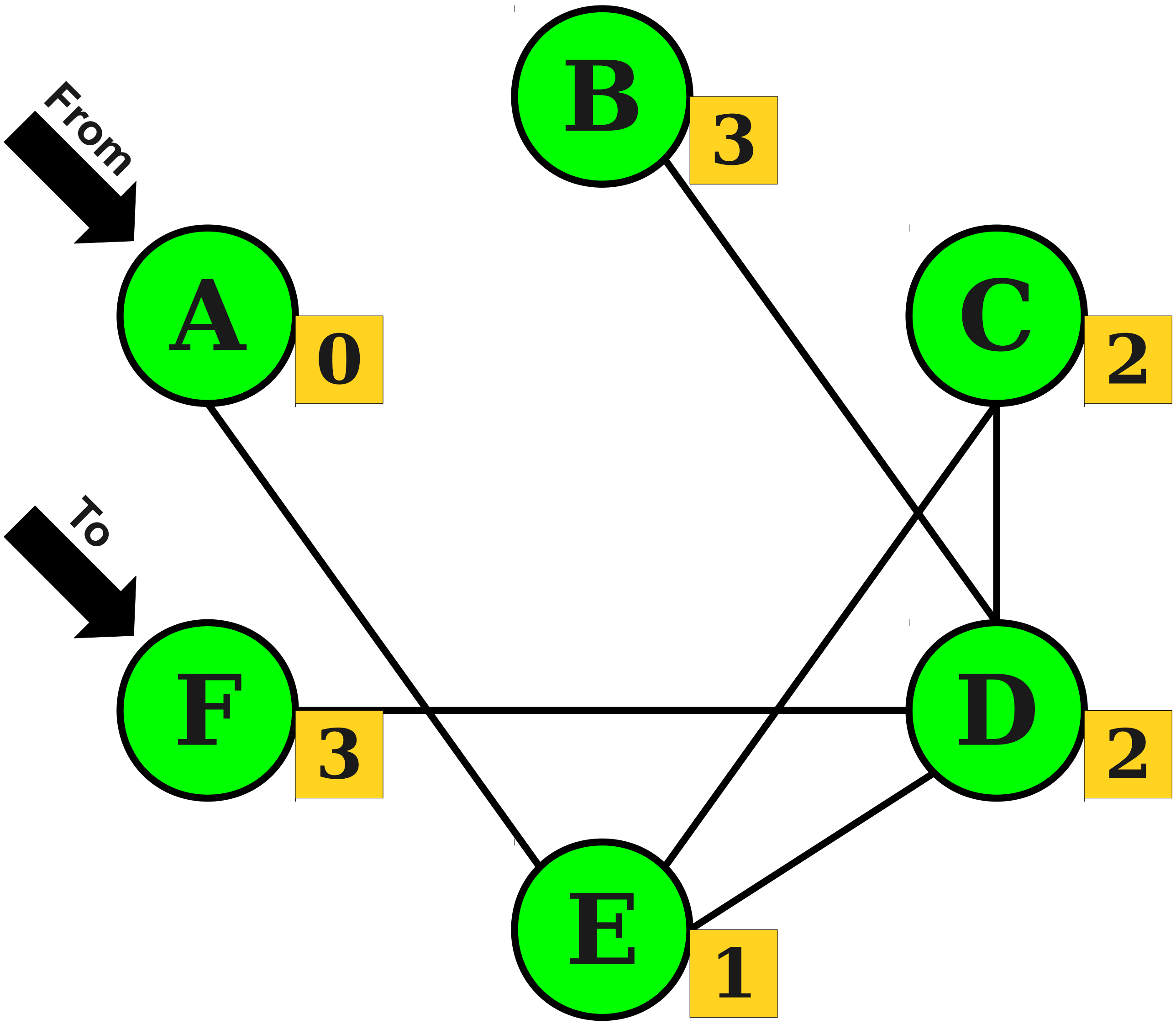


Number of possible ways to order a subset of n nodes is **$O(n \times n!)$**

Time to check a path is **$O(n)$**.

Runtime: **$O(n^2 \cdot n!)$**

Another Problem



With a precise analysis, runtime is **O(n + m)**, where n is the number of nodes and m is the number of edges.

For Comparison

- **Longest increasing subsequence:**
  - Naive: $O(n \cdot 2^n)$
  - Fast: $O(n^2)$
- **Shortest path problem:**
  - Naive: $O(n^2 \cdot n!)$
  - Fast: $O(n + m)$, where n is the number of nodes and m the number of edges. (Take CS161 for details!)

Defining Efficiency

- When dealing with problems that search for the “best” object of some sort, there are often at least exponentially many possible options.
- Brute-force solutions tend to take at least exponential time to complete.
- Clever algorithms often run in time $O(n)$, or $O(n^2)$, or $O(n^3)$, etc.

Polynomials and Exponentials

- A TM runs in **polynomial time** iff its runtime is some polynomial in n.
  - That is, time $O(n^k)$ for some constant k.
- Polynomial functions “scale well.”
  - Small changes to the size of the input do not typically induce enormous changes to the overall runtime.
- Exponential functions scale terribly.
  - Small changes to the size of the input induce huge changes in the overall runtime.

The Cobham-Edmonds Thesis

A language L can be **decided efficiently** iff there is a TM that decides it in polynomial time.

Equivalently, L can be decided efficiently iff it can be decided in time $O(n^k)$ for some $k \in \mathbb{N}$.

Like the Church-Turing thesis, this is ***not*** a theorem!

It's an assumption about the nature of efficient computation, and it is somewhat controversial.

The Cobham-Edmonds Thesis

- Efficient runtimes:
  - $4n + 13$
  - $n^3 - 2n^2 + 4n$
  - $n \log \log n$
- "Efficient" runtimes:
  - $n^{1,000,000,000,000}$
  - 10^{500}
- Inefficient runtimes:
  - 2^n
  - $n!$
  - n^n
- "Inefficient" runtimes:
  - $n^{0.0001 \log n}$
  - 1.000000001^n

The Complexity Class **P**

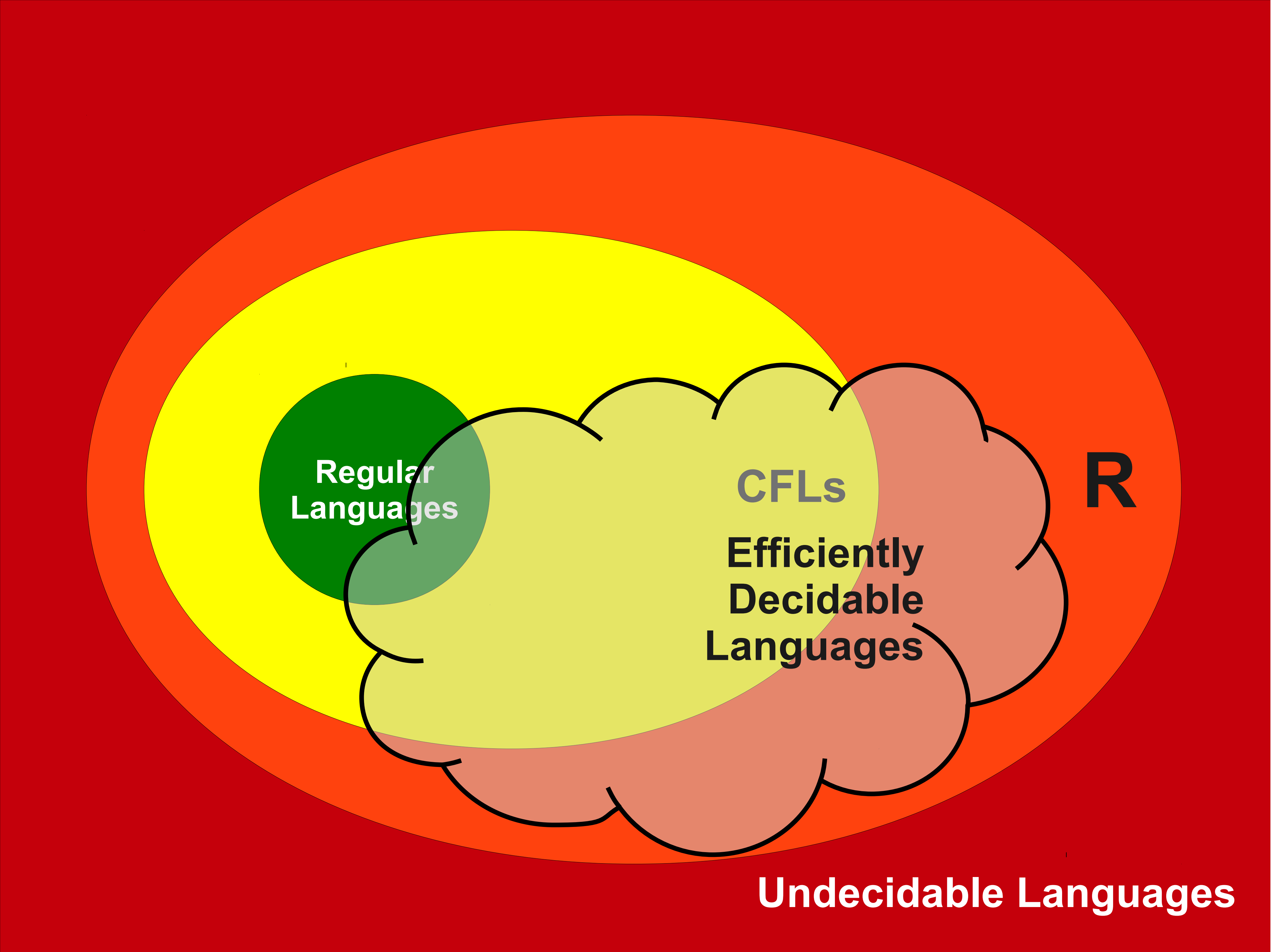
- The **complexity class P** (for **p**olynomial time) contains all problems that can be solved in polynomial time.
- Formally:

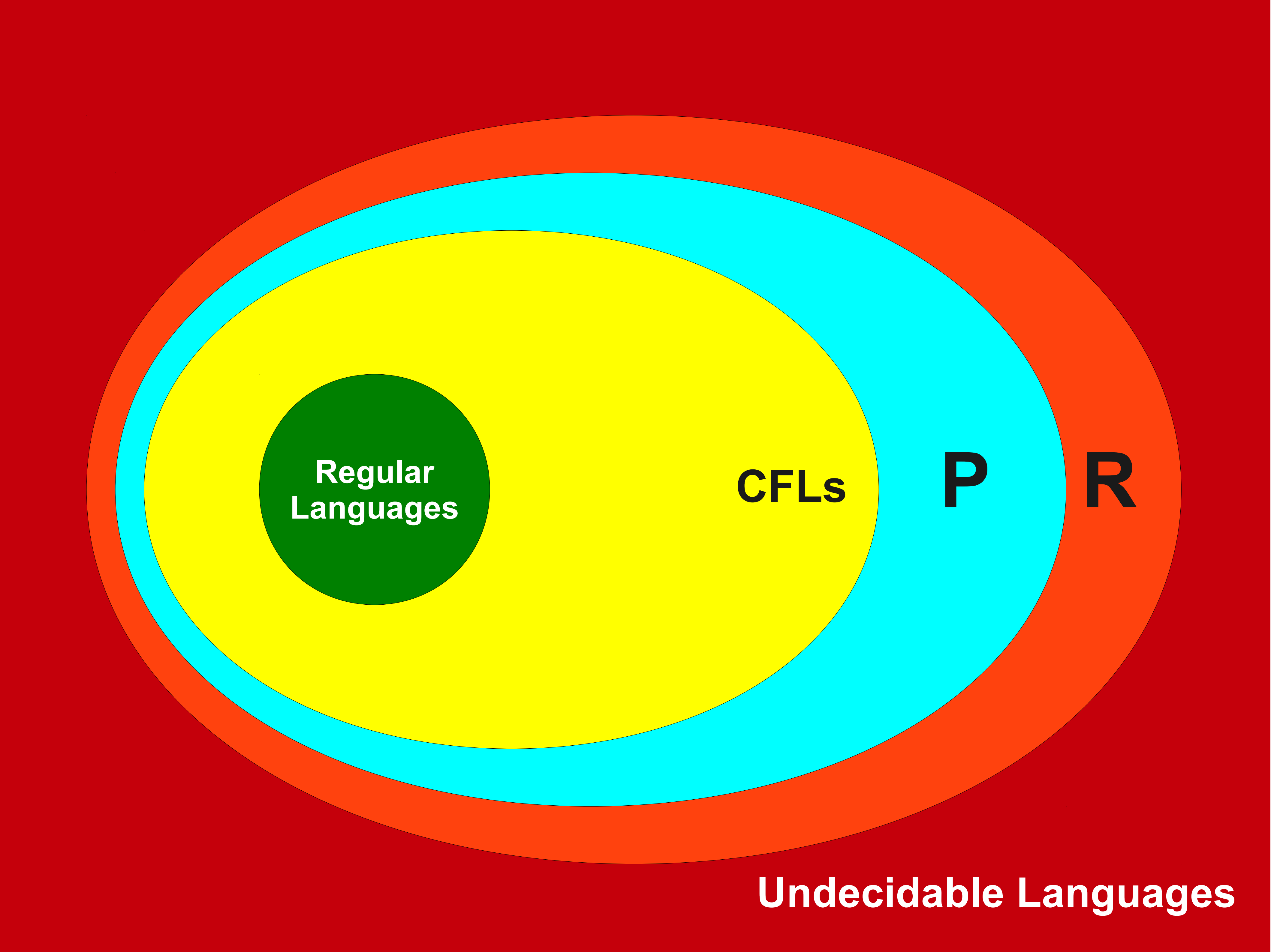
$$\mathbf{P} = \{\ L \mid \text{There is a polynomial-time decider for } L\ \}$$

- Assuming the Cobham-Edmonds thesis, a language is in **P** iff it can be decided efficiently.

Examples of Problems in **P**

- All regular languages are in **P**.
  - All have linear-time TMs.
- All CFLs are in **P**.
  - Requires a more nuanced argument (the *CYK algorithm* or *Earley's algorithm.*)
- Many other problems are in **P**.
  - More on that in a second.


Regular Languages
CFLs
Efficiently Decidable Languages
R
Undecidable Languages


Regular Languages
CFLs
P
R
Undecidable Languages