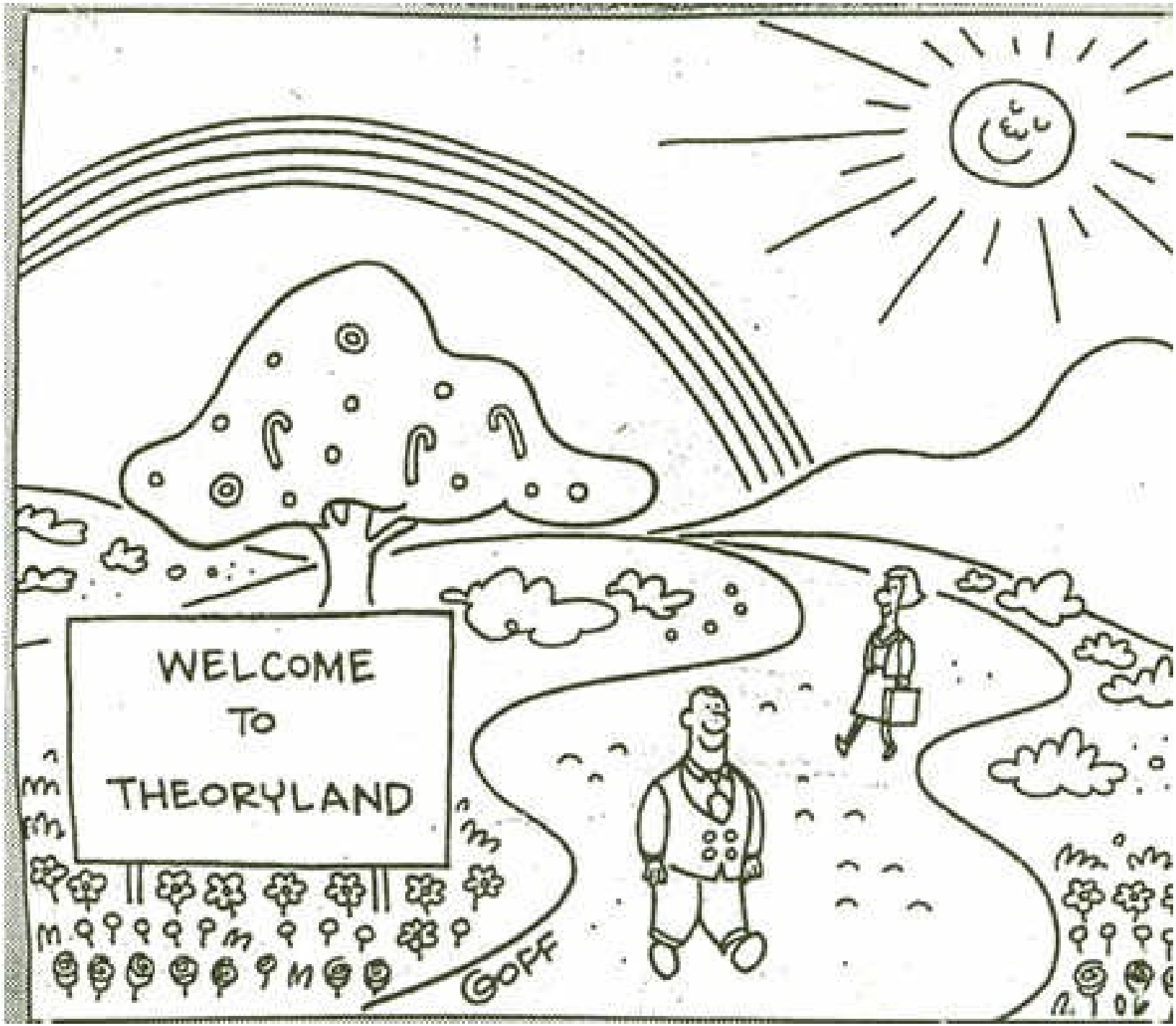
Friday Four Square!
4:15PM, Outside Gates

Announcements

- Problem Set 7 due right now.
- Problem Set 8 due next Friday at 2:15 PM.
  - Stop by OH with questions!
  - Email the staff list with questions!

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

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 is true or false under the laws of Presburger arithmetic.

A Decidable Problem

- **Presburger arithmetic** is a logical system for reasoning about arithmetic.
  - $\forall x.\ x + 1 \neq 0$
  - $\forall x.\ \forall y.\ (x + 1 = y + 1 \rightarrow x = y)$
  - $\forall x.\ x + 0 = x$
  - $\forall x.\ \forall y.\ (x + y) + 1 = x + (y + 1)$
  - $\forall x.\ ((P(0) \wedge \forall y.\ (P(y) \rightarrow P(y + 1))) \rightarrow \forall x.\ P(x)$
- Given a statement, it is decidable whether that statement is true or false under the laws of Presburger arithmetic.
- Any Turing machine that decides a Presburger arithmetic statement with n symbols may have to move the tape head $2^{2^{cn}}$ times.

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

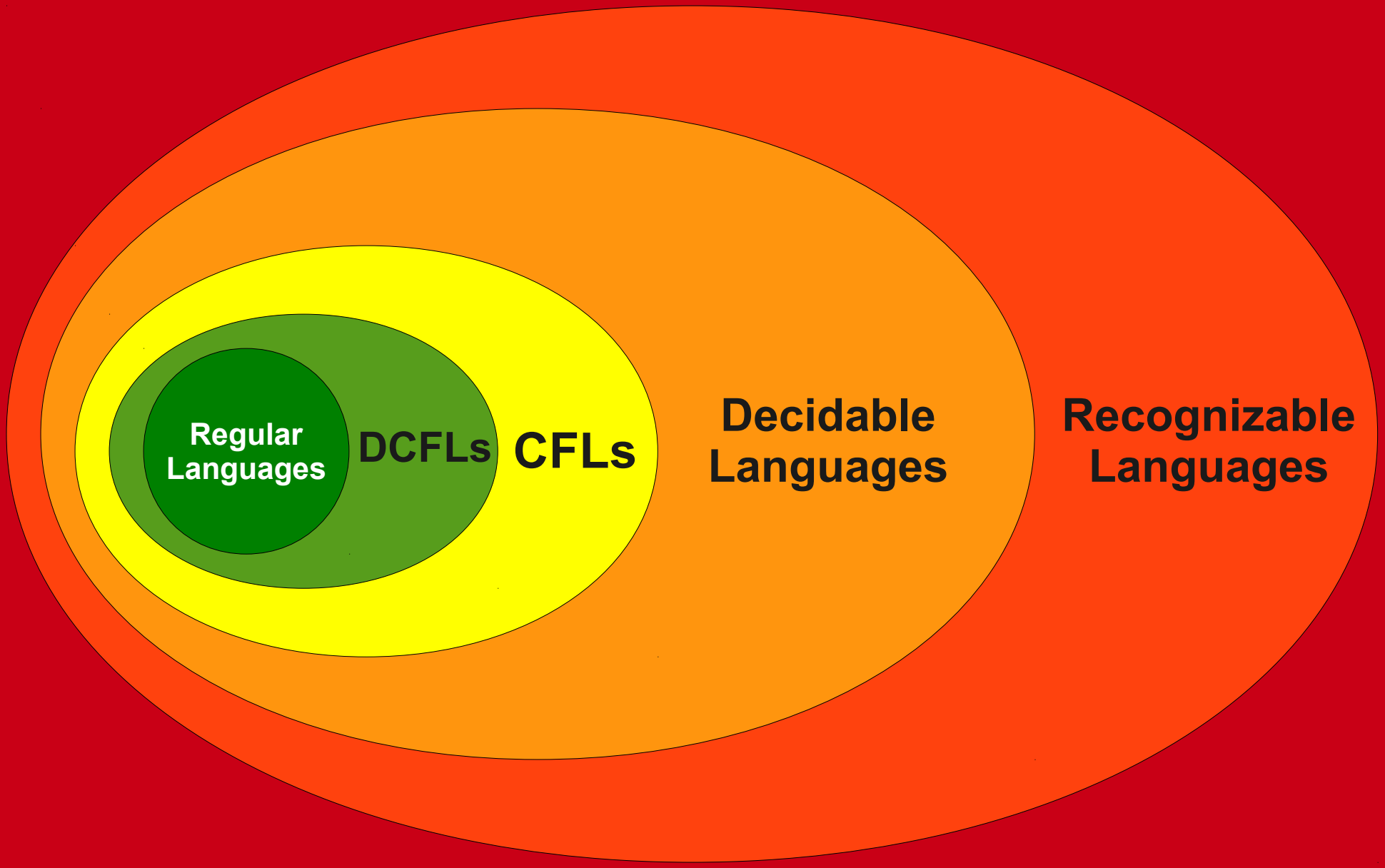
- The fact that a problem is decidable does not mean that it is *feasibly* decidable.
- In **computability theory**, we ask the question

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

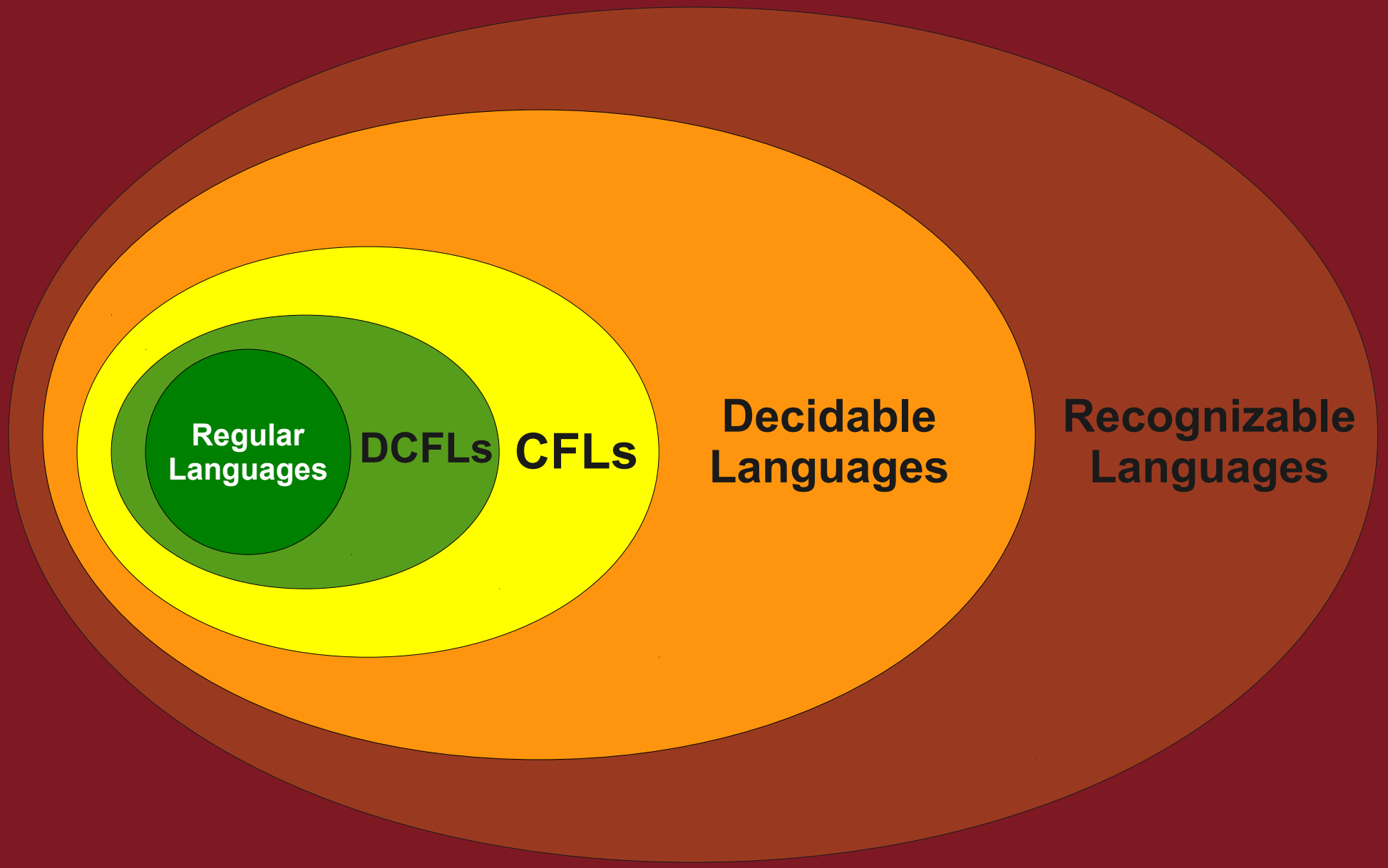
- In **complexity theory**, we ask the question

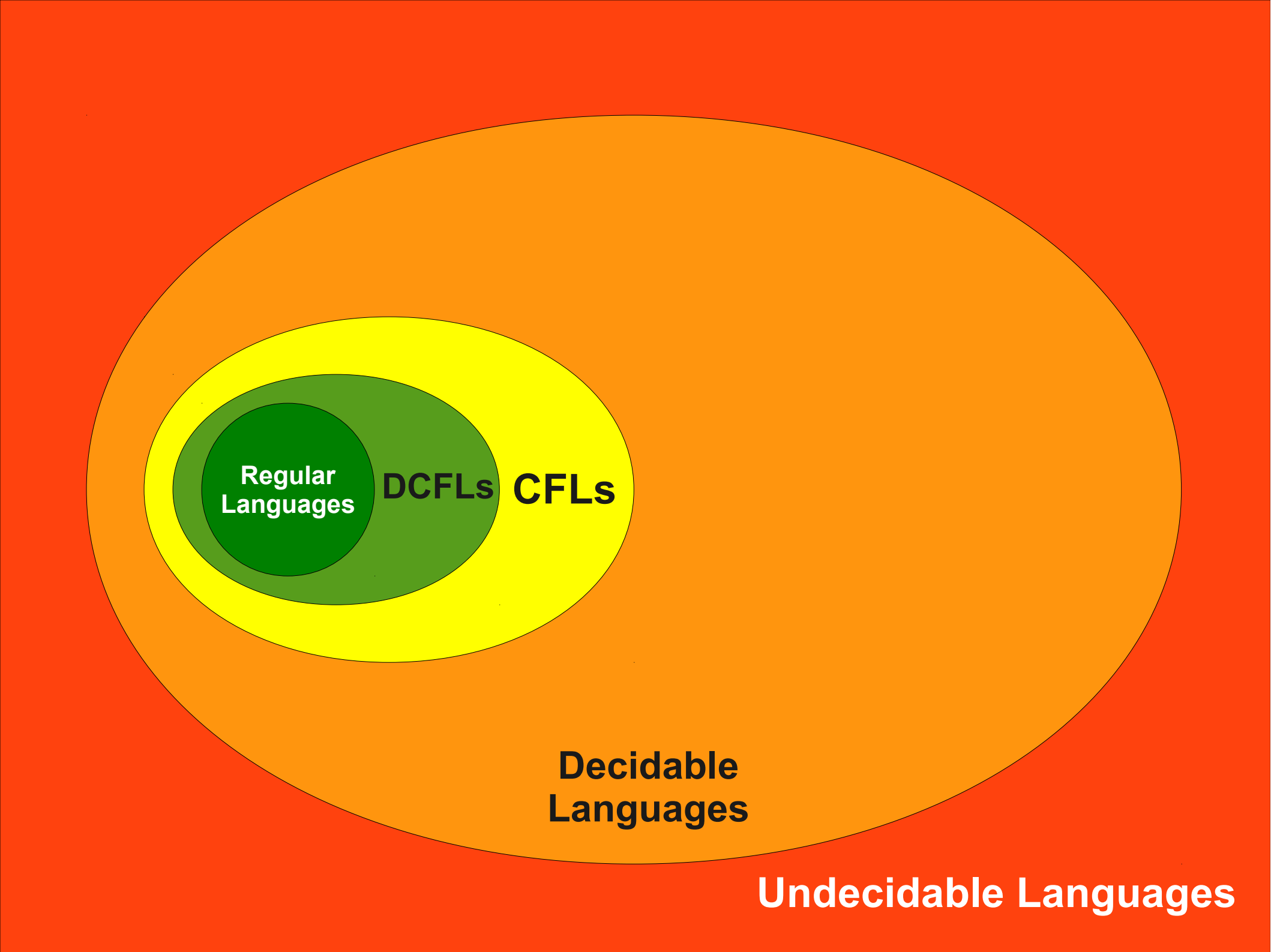
  Is it possible to solve problem P **efficiently**?

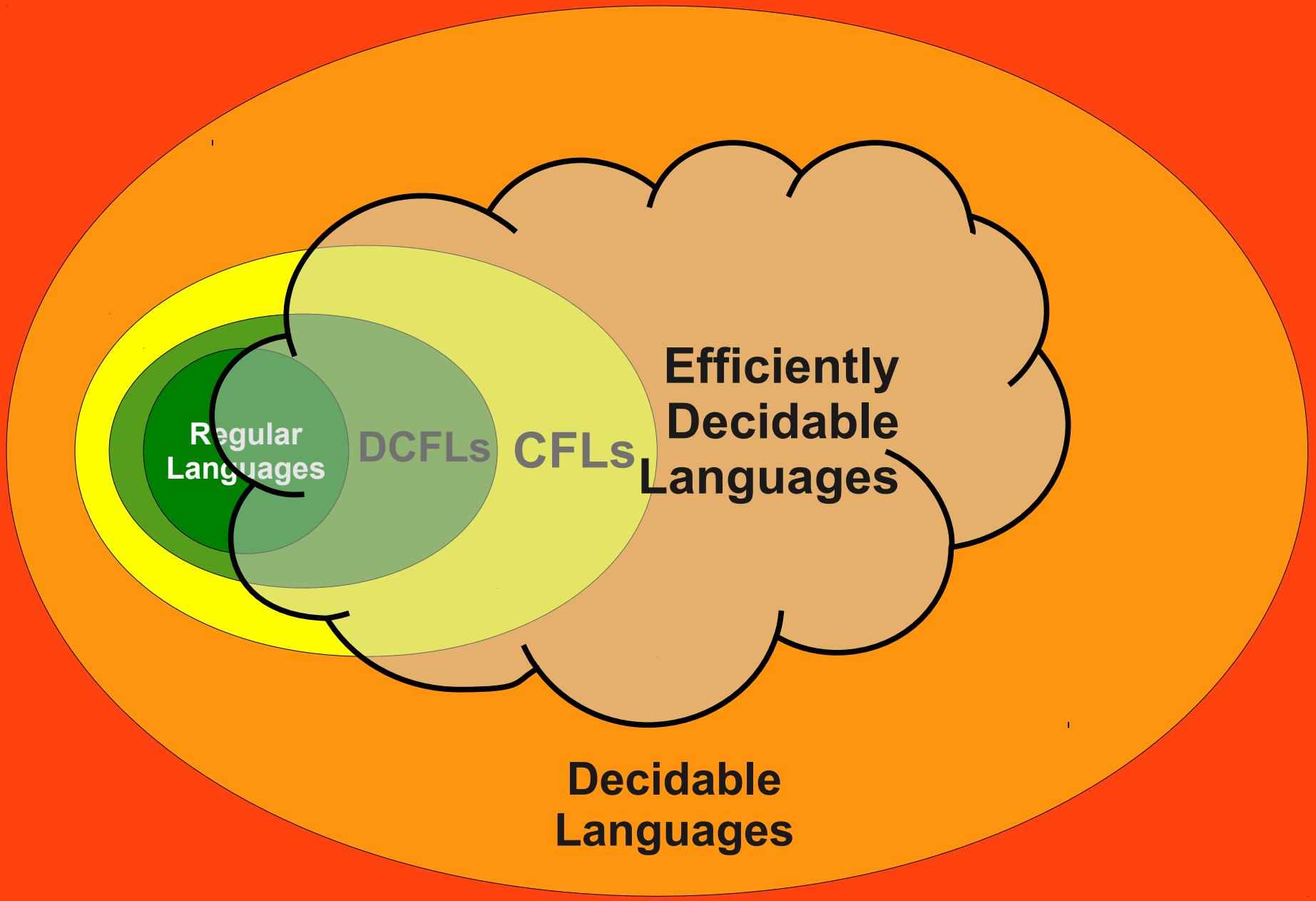
- In the remainder of this course, we will explore this question in more detail.



All Languages


Regular Languages
DCFLs
CFLs
Decidable Languages
Recognizable Languages
All Languages


Regular Languages
DCFLs
CFLs
Decidable Languages
Undecidable 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.
- Running a TM on different inputs might take a different number of steps.

	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	0	1	0				

	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Time Complexity

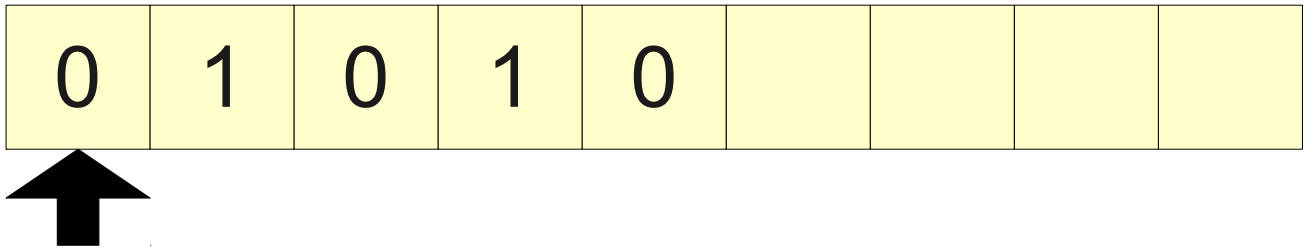
- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	0	1	0				

	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



	0	1	B
q_0	0 R q_1	reject	accept
q_1	reject	1 R q_0	accept

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	0	1	0				

	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

0

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	0	1	0				

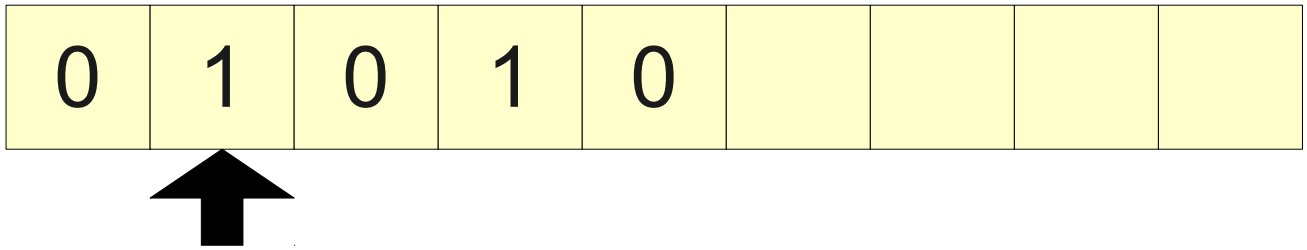
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

0

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



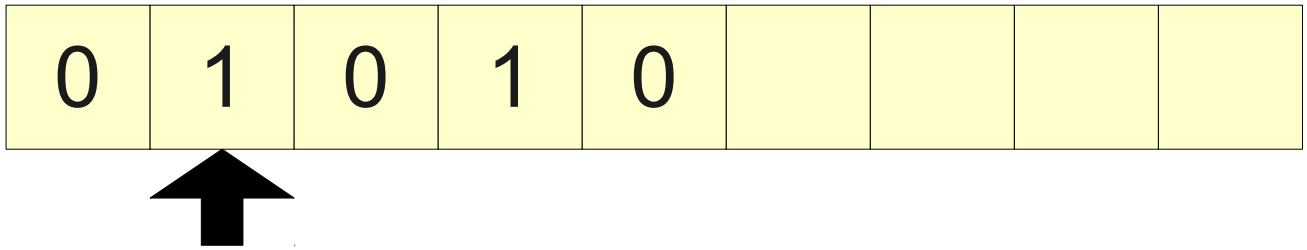
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

1

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

1

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	0	1	0				

	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

2

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	0	1	0				

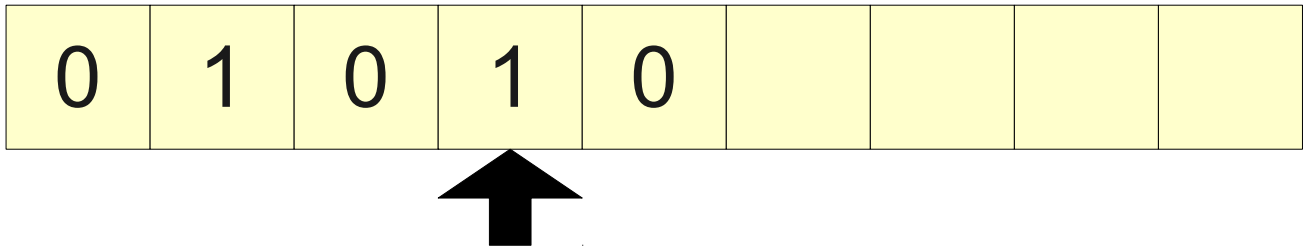
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

2

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



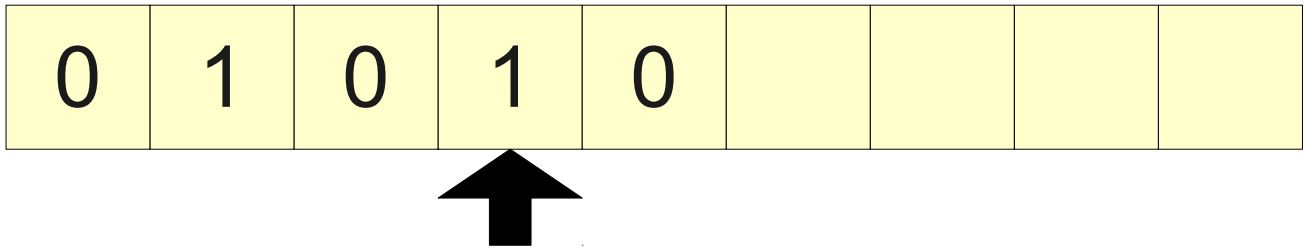
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

3

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

3

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	0	1	0				

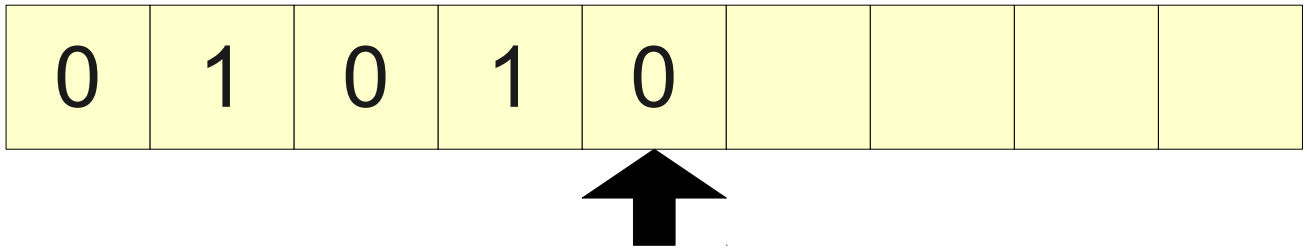
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

4

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



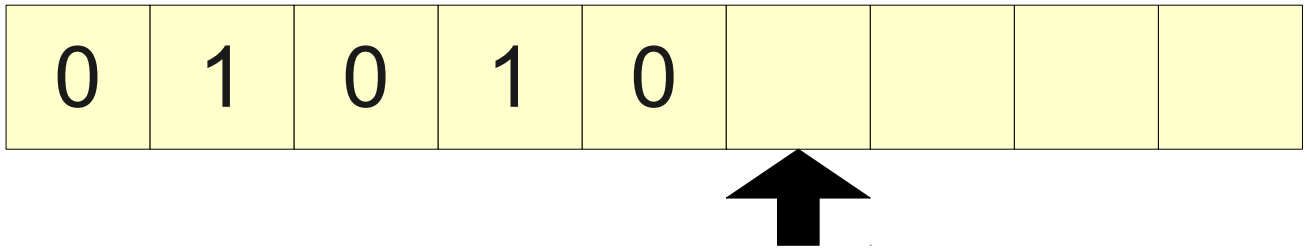
	0	1	B
q_0	0 R q_1	reject	accept
q_1	reject	1 R q_0	accept

Step Counter

4

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



	0	1	B
q_0	0 R q_1	reject	accept
q_1	reject	1 R q_0	accept

Step Counter

5

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	0	1	0				

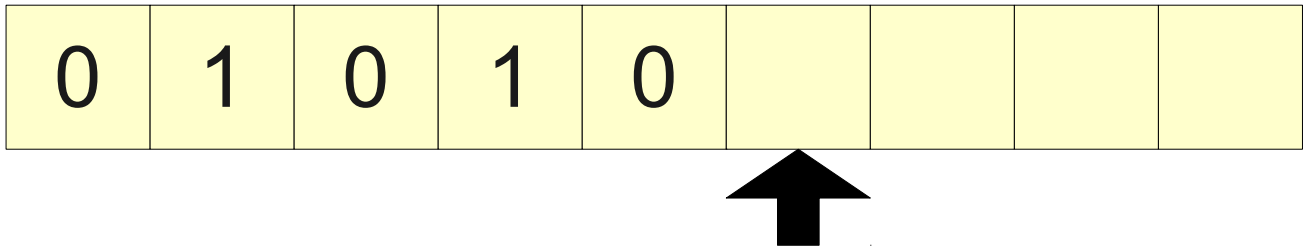
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

5

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



	0	1	B
q_0	0 R q_1	reject	accept
q_1	reject	1 R q_0	accept

Step Counter

6

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	0	1	0				

	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

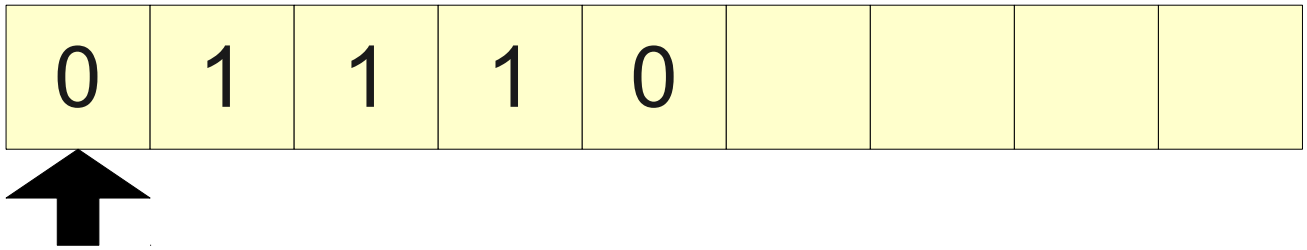
Remember that accepting or rejecting works by transitioning into a special state!

Step Counter

6

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

0

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	1	1	0				

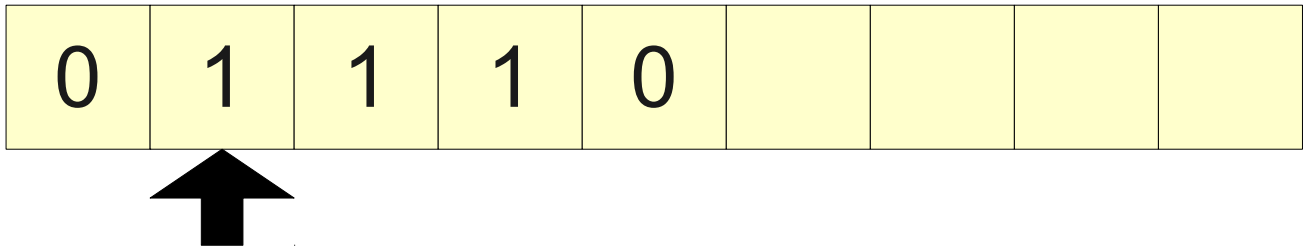
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

0

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



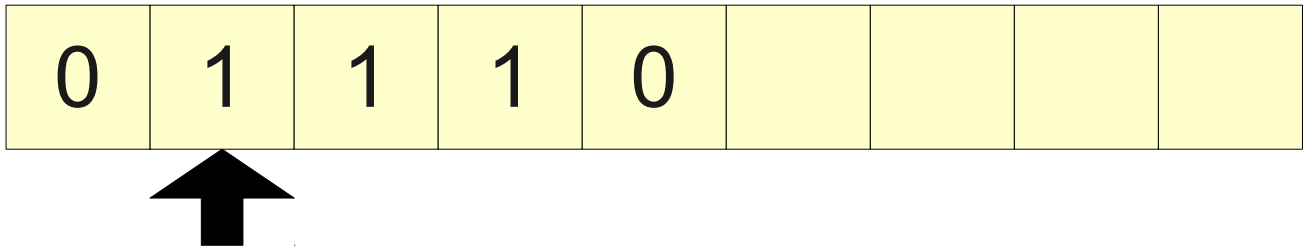
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

1

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

1

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	1	1	0				

	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

2

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

0	1	1	1	0				

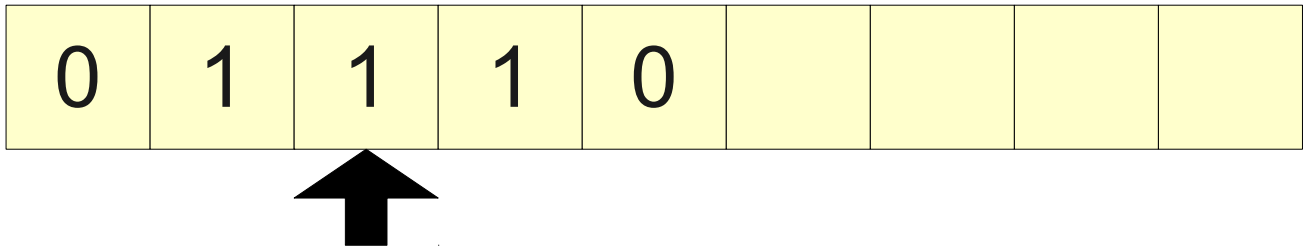
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

2

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



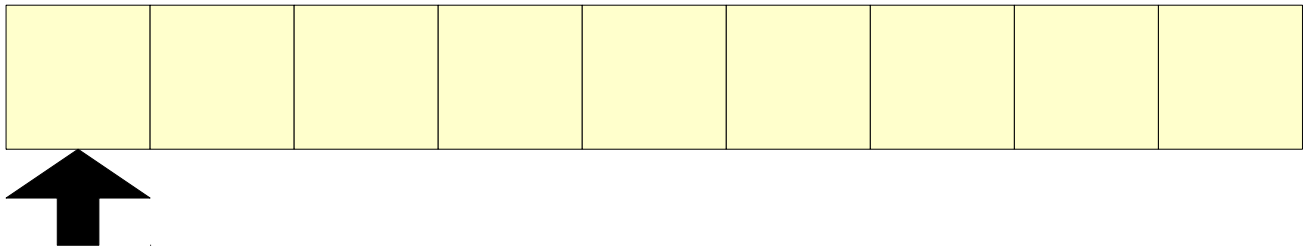
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

3

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



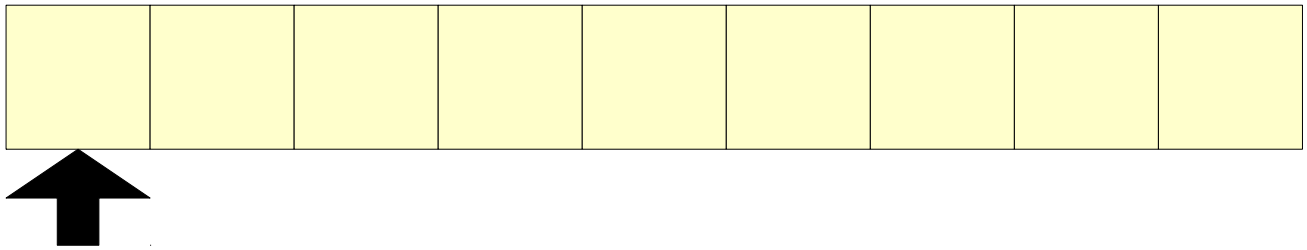
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

0

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



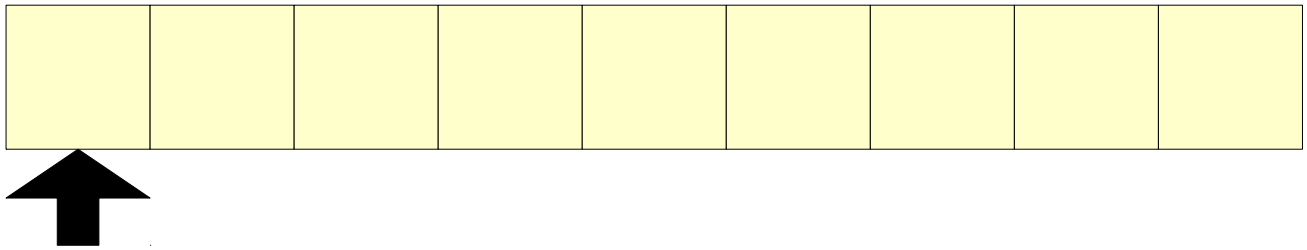
	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

0

Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.



	0			1			B
q_0	0	R	q_1	reject			accept
q_1	reject			1	R	q_0	accept

Step Counter

1

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 f(n) that measures the *worst-case* number of steps M takes on any input of length n.
  - By convention, n is the length of the input.
  - If the TM loops on some input of length k, then f(k) = ∞.
- The previous TM has a time complexity of f(n) = n + 1.
  - Any input of length n of the form `01010`... halts after n + 1 steps.
  - Some inputs may take less time to halt, but time complexity considers the worst-case complexity.

A Slight Problem

- Consider the following TM over Σ = {`0`, `1`} for the language *BALANCE* = { *w* | *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 is found, reject.
  - Otherwise, cross off that symbol and repeat.”

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 is found, reject.
  - Otherwise, cross off that symbol and repeat.”

At most n steps.

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.
  - If none is found, reject.
  - Otherwise, cross off that symbol and repeat.”

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 is found, reject.
  - Otherwise, cross off that symbol and repeat.”

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 is found, reject. **At most 1 step**
  - Otherwise, cross off that symbol and repeat.”

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 is 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.**

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 is 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.**

+

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 is 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 3n + 2 steps.

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 is 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 3n + 2 steps.

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 is 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 3n + 2 steps.

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 is 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 3n + 2 steps.

× **At most n/2 loops.**

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 is 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 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:
  - $4n + 4 = \mathbf{O(n)}$
  - $137n + 271 = \mathbf{O(n)}$
  - $n^2 + 3n + 4 = \mathbf{O(n^2)}$
  - $2^n + n^3 = \mathbf{O(2^n)}$

Big-O Notation, Formally

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

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

- Intuitively, as n gets "sufficiently 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.”

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

Life is Easier with Big-O

- M = “On input w:
  - Scan across the tape until a 0 or 1 is found. **O(n) steps**
  - If none are found, accept. **O(1) steps**
  - 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.”

Life is Easier with Big-O

- M = “On input w:
  - Scan across the tape until a 0 or 1 is found. **O(n) steps**
  - If none are found, accept. **O(1) steps**
  - If one is found, continue scanning until a matching 1 or 0 is found. **O(n) steps**
  - If none is found, reject.
  - Otherwise, cross off that symbol and repeat.”

Life is Easier with Big-O

- M = “On input w:
  - Scan across the tape until a 0 or 1 is found. **O(n) steps**
  - If none are found, accept. **O(1) steps**
  - If one is found, continue scanning until a matching 1 or 0 is found. **O(n) steps**
  - If none is found, reject. **O(1) steps**
  - Otherwise, cross off that symbol and repeat.”

Life is Easier with Big-O

- M = “On input w:
  - Scan across the tape until a 0 or 1 is found. **O(n) steps**
  - If none are found, accept. **O(1) steps**
  - If one is found, continue scanning until a matching 1 or 0 is found. **O(n) steps**
  - If none is found, reject. **O(1) steps**
  - Otherwise, cross off that symbol and repeat.” **O(n) steps**

Life is Easier with Big-O

- M = “On input w:
  - Scan across the tape until a 0 or 1 is found. **O(n) steps**
  - If none are found, accept. **O(1) steps**
  - If one is found, continue scanning until a matching 1 or 0 is found. **O(n) steps**
  - If none is found, reject. **O(1) steps**
  - Otherwise, cross off that symbol and repeat.” **+ O(n) steps**

Life is Easier with Big-O

- M = “On input w:
  - Scan across the tape until a 0 or 1 is found. **O(n) steps**
  - If none are found, accept. **O(1) steps**
  - If one is found, continue scanning until a matching 1 or 0 is found. **O(n) steps**
  - If none is found, reject. **O(1) steps**
  - Otherwise, cross off that symbol and repeat.” **+ O(n) steps**

O(n) steps.

Life is Easier with Big-O

- M = “On input w:
  - Scan across the tape until a 0 or 1 is found. — **O(n) steps**
  - If none are found, accept. — **O(1) steps**
  - If one is found, continue scanning until a matching 1 or 0 is found. — **O(n) steps**
  - If none is found, reject. — **O(1) steps**
  - Otherwise, cross off that symbol and repeat.” — **+ O(n) steps**

O(n) loops

O(n) steps.

Life is Easier with Big-O

- M = “On input w:
  - Scan across the tape until a 0 or 1 is found. — **O(n) steps**
  - If none are found, accept. — **O(1) steps**
  - If one is found, continue scanning until a matching 1 or 0 is found. — **O(n) steps**
  - If none is found, reject. — **O(1) steps**
  - Otherwise, cross off that symbol and repeat.” — **O(n) steps**

O(n) loops

+	**O(n) steps.**
×	**O(n) loops.**

Life is Easier with Big-O

- M = “On input w:
  - Scan across the tape until a 0 or 1 is found. — **O(n) steps**
  - If none are found, accept. — **O(1) steps**
  - If one is found, continue scanning until a matching 1 or 0 is found. — **O(n) steps**
  - If none is found, reject. — **O(1) steps**
  - Otherwise, cross off that symbol and repeat.” — **O(n) steps**

O(n) loops

+	**O(n) steps**
	O(n) steps.
×	**O(n) loops.**
	$O(n^2)$ steps.

An Important Observation

- Recall: A **multitape Turing machine** (MTTM) is a Turing machine with multiple tapes.
- The input tape holds the original input.
- Each tape head can move independently of the rest.
- Each tape head can base its transition on the symbols under all tape heads.

An MTTM for *BALANCE*

- M' = “On input w:
  - Scan across the tape and copy all 1s to a secondary tape.
  - Move both tape heads back to the start of their tapes.
  - Until the end of the input is reached:
    - Scan on the input tape until a 0 is found.
    - Match the 0 with a 1 on the second tape.
  - If an imbalance is found, reject.
  - If all 0s and 1s are matched, accept.

An MTTM for *BALANCE*

- M' = “On input w:
  - Scan across the tape and copy all 1s to a secondary tape.
  - Move both tape heads back to the start of their tapes.
  - Until the end of the input is reached:
    - Scan on the input tape until a 0 is found.
    - Match the 0 with a 1 on the second tape.
  - If an imbalance is found, reject.
  - If all 0s and 1s are matched, accept.

O(n) steps

An MTTM for *BALANCE*

- M' = “On input w:
  - Scan across the tape and copy all 1s to a secondary tape. **O(n) steps**
  - Move both tape heads back to the start of their tapes. **O(n) steps**
  - Until the end of the input is reached:
    - Scan on the input tape until a 0 is found.
    - Match the 0 with a 1 on the second tape.
  - If an imbalance is found, reject.
  - If all 0s and 1s are matched, accept.

An MTTM for *BALANCE*

- M' = “On input w:
  - Scan across the tape and copy all 1s to a secondary tape. **O(n) steps**
  - Move both tape heads back to the start of their tapes. **O(n) steps**
  - Until the end of the input is reached:
    - Scan on the input tape until a 0 is found.
    - Match the 0 with a 1 on the second tape.

    O(n) steps
  - If an imbalance is found, reject.
  - If all 0s and 1s are matched, accept.

An MTTM for *BALANCE*

- M' = “On input w:
  - Scan across the tape and copy all 1s to a secondary tape. **O(n) steps**
  - Move both tape heads back to the start of their tapes. **O(n) steps**
  - Until the end of the input is reached:
    - Scan on the input tape until a 0 is found.
    - Match the 0 with a 1 on the second tape.

    O(n) steps
  - If an imbalance is found, reject. **O(1) steps**
  - If all 0s and 1s are matched, accept.

An MTTM for *BALANCE*

- M' = “On input w:
  - Scan across the tape and copy all 1s to a secondary tape. **O(n) steps**
  - Move both tape heads back to the start of their tapes. **O(n) steps**
  - Until the end of the input is reached:
    - Scan on the input tape until a 0 is found.
    - Match the 0 with a 1 on the second tape.

    O(n) steps
  - If an imbalance is found, reject. **O(1) steps**
  - If all 0s and 1s are matched, accept. **O(1) steps**

An MTTM for *BALANCE*

- M' = “On input w:
  - Scan across the tape and copy all 1s to a secondary tape. **O(n) steps**
  - Move both tape heads back to the start of their tapes. **O(n) steps**
  - Until the end of the input is reached: **O(n) steps**
    - Scan on the input tape until a 0 is found.
    - Match the 0 with a 1 on the second tape.
  - If an imbalance is found, reject. **O(1) steps**
  - If all 0s and 1s are matched, accept. + **O(1) steps**

An MTTM for *BALANCE*

- M' = “On input w:
  - Scan across the tape and copy all 1s to a secondary tape. **O(n) steps**
  - Move both tape heads back to the start of their tapes. **O(n) steps**
  - Until the end of the input is reached: **O(n) steps**
    - Scan on the input tape until a 0 is found.
    - Match the 0 with a 1 on the second tape.
  - If an imbalance is found, reject. **O(1) steps**
  - If all 0s and 1s are matched, accept. + **O(1) steps**

O(n) steps.

A Performance Comparison

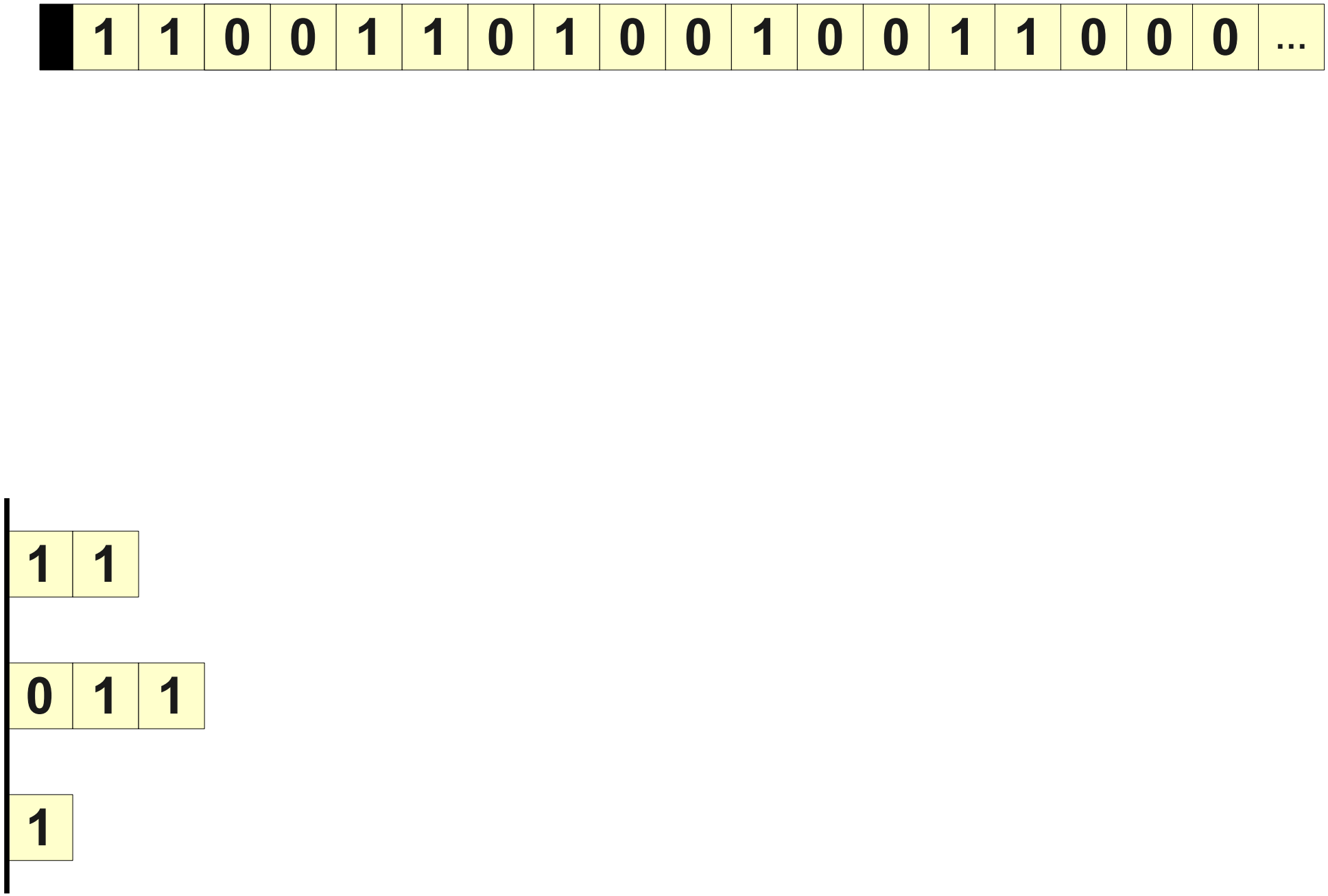
- Our original 1-tape TM for *BALANCE* runs in $O(n^2)$ time.
  - Using some tricks as suggested by Sipser, this can be whittled down to O(n log n).
- Our MTTM can decide *BALANCE* in O(n) time.
- Nontrivial result: There is no single-tape TM that can decide *BALANCE* in O(n) time.
- The MTTM is *inherently faster* than the single-tape TM!

Complexity is Tricky

- The Church-Turing thesis states that any feasible model of computation is no more powerful than a TM.
- However, some models of computation might be more *efficient* than the TM.
- When analyzing complexity, the actual model of computation matters!

Connecting Models of Computation

- **Theorem**: If there is a **WB** program for L whose time complexity is $f(n)$, there is a TM whose time complexity is at most $2f(n)$.
- **Proof sketch**: Every line in a WB program gets converted into a set of TM states. Executing each line makes at most two transitions. Thus if the **WB** program takes time $f(n)$, then TM takes time at most $2f(n)$.


1 1 0 0 1 1 0 1 0 0 1 0 0 1 1 0 0 0 ...
1 1
0 1 1
1

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	...
																		...
																		...
																		...
																		...

1	1	
0	1	1
1		

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	...
																		...
																		...
																		...
																		...

1	1	
0	1	1
1		

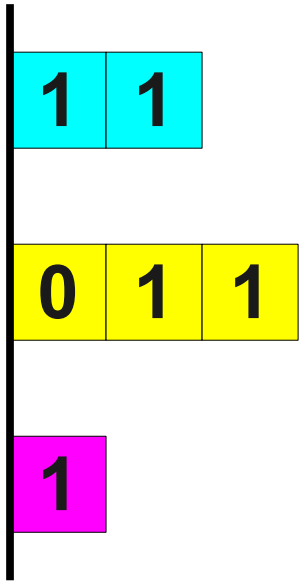
1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	...
																		...
																		...
																		...
																		...

1	1	
0	1	1
1		

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	...
>	1	1	<															...
>	0	1	1	<														...
>	1	<																...
																		...

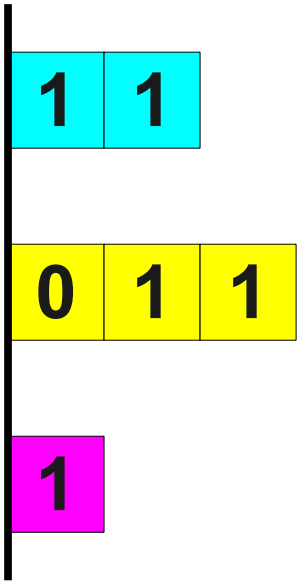
1	1	
0	1	1
1		

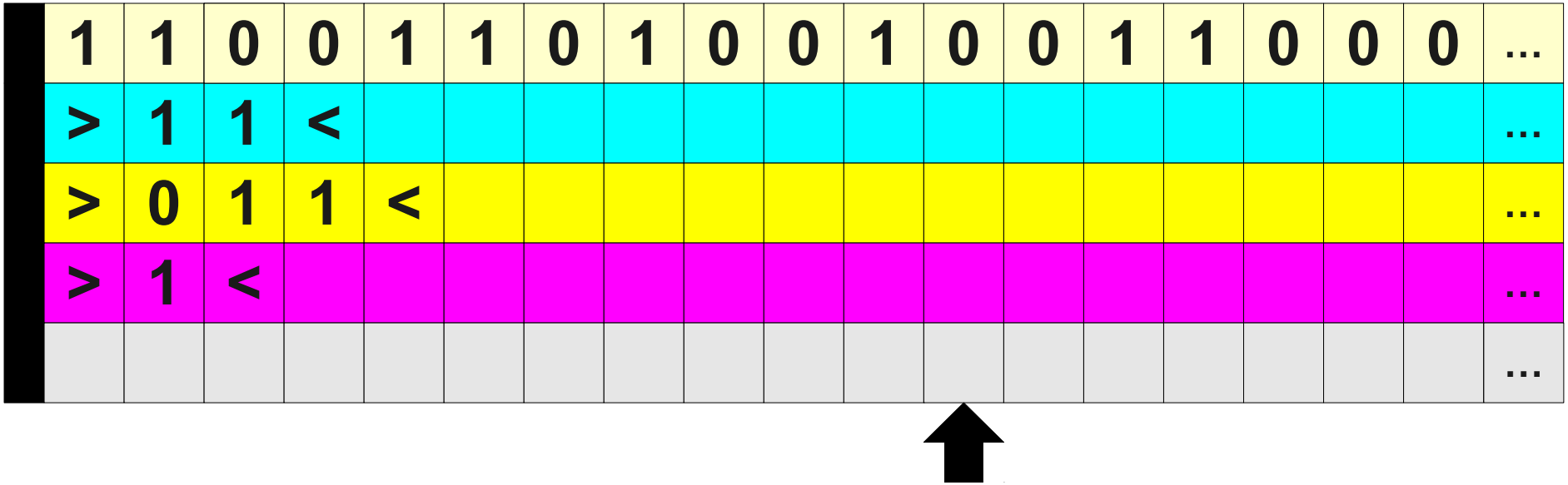
1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	...
>	1	1	<															...
>	0	1	1	<														...
>	1	<																...
																		...



1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	...
>	1	1	<															...
>	0	1	1	<														...
>	1	<																...
																		...

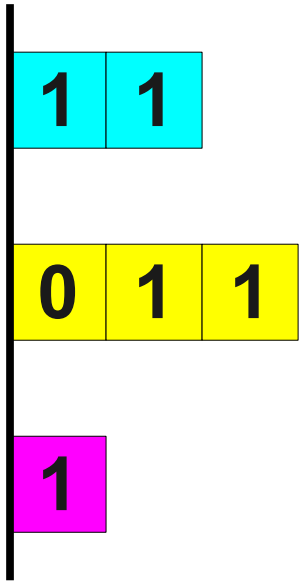
0: Push 1 onto Stack 3.





0: Push 1 onto Stack 3.

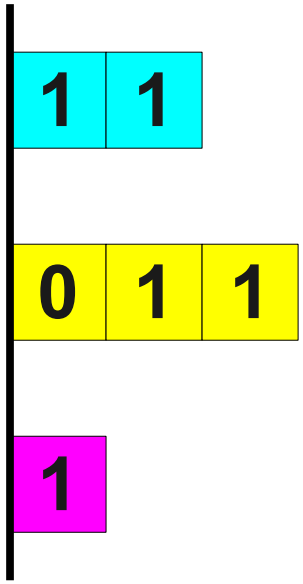
0: Write × on track 5.



1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	<																…
											×							…

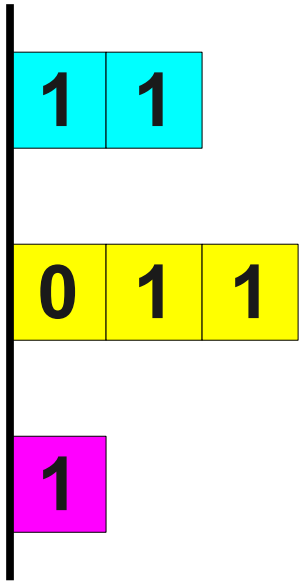
0: Push 1 onto Stack 3.

0: Write × on track 5.



1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	<																…
											×							…

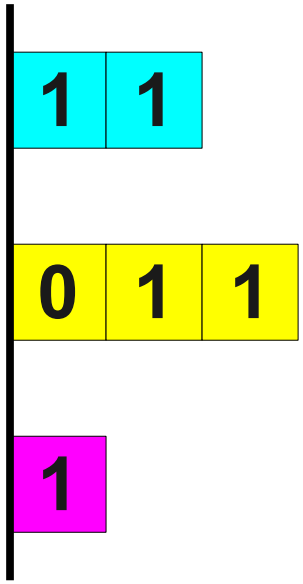
0: Push 1 onto Stack 3.



0: Write × on track 5.

1: Move left until {>} on track 4.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	<																…
											×							…

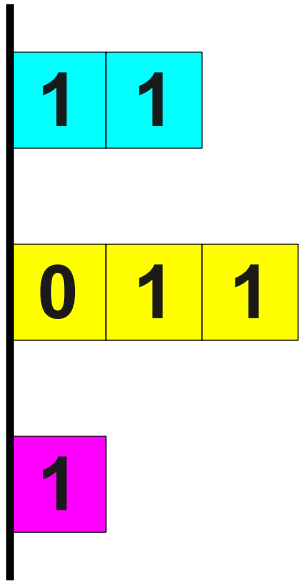


0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	<																…
											×							…



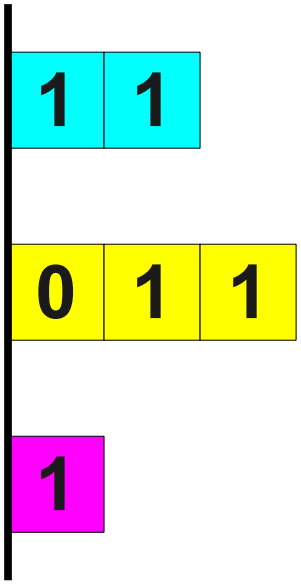
0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	<																…
											×							…



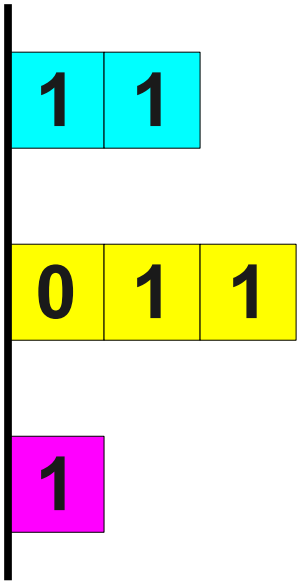
0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	<																…
											×							…



0: Push 1 onto Stack 3.

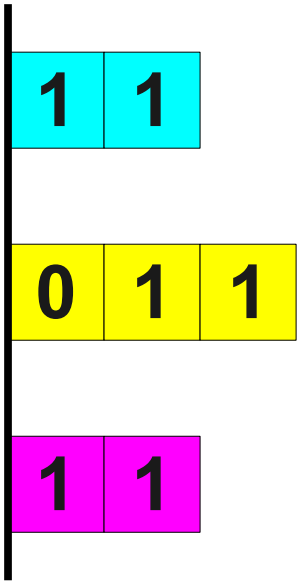
0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	1																…
											×							…



0: Push 1 onto Stack 3.

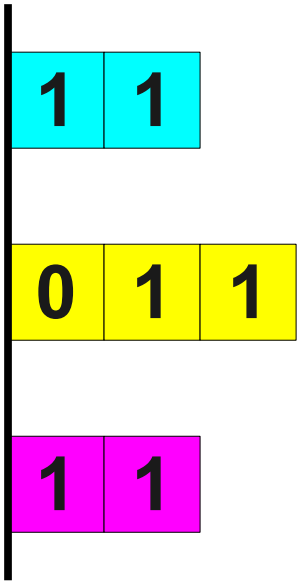
0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	1																…
											×							…



0: Push 1 onto Stack 3.

0: Write × on track 5.

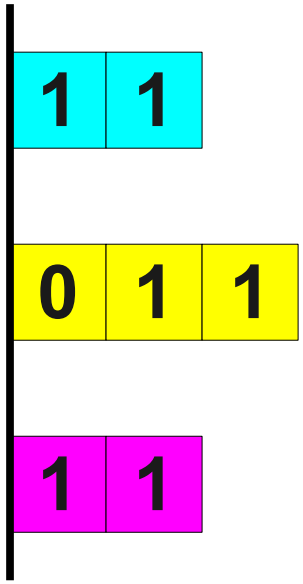
1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	1																…
											×							…



0: Push 1 onto Stack 3.

0: Write × on track 5.

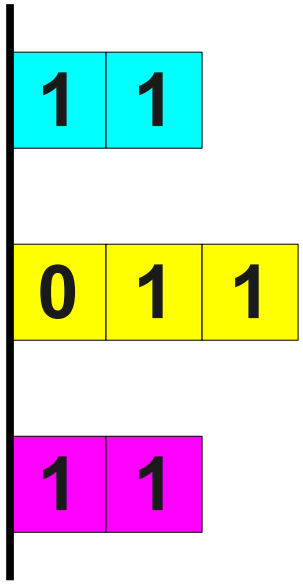
1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	1																…
											×							…



0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

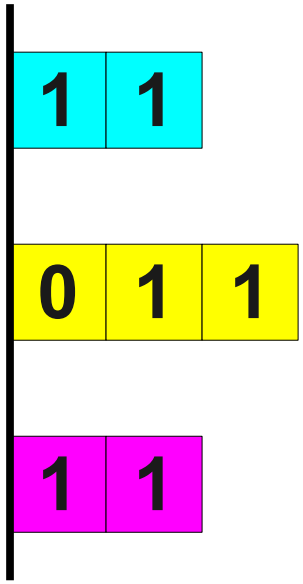
2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

5: Write < on track 4

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	1	<															…
											×							…



0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

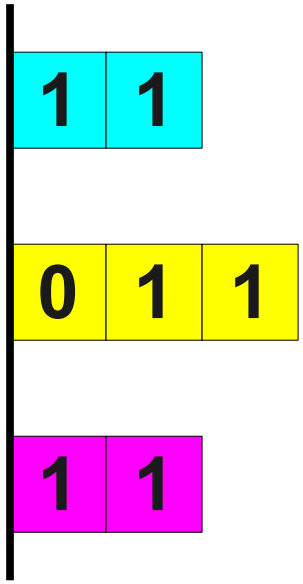
2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

5: Write < on track 4

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	1	<															…
											×							…



0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

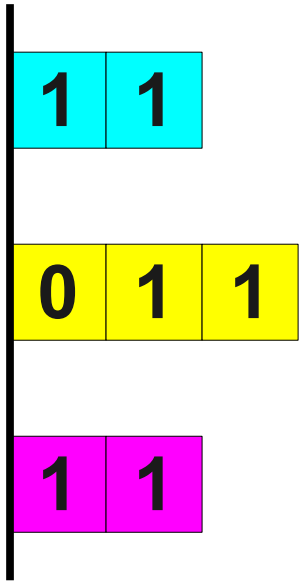
3: Write 1 on track 4.

4: Move right.

5: Write < on track 4

6: Move left until {>} on track 4.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	1	<															…
											×							…



0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

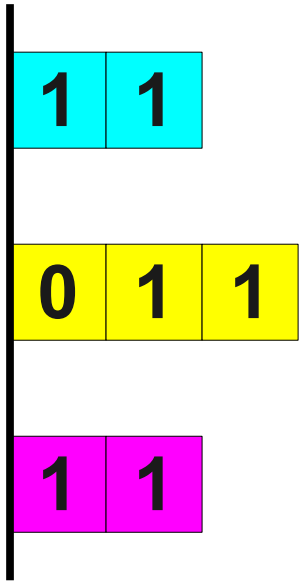
3: Write 1 on track 4.

4: Move right.

5: Write < on track 4

6: Move left until {>} on track 4.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	1	<															…
											×							…



0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

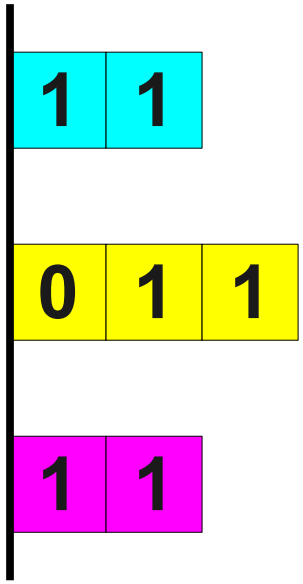
5: Write < on track 4

6: Move left until {>} on track 4.

7: Move right until {×} on track 5.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	1	<															…
											×							…

0: Push 1 onto Stack 3.



0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

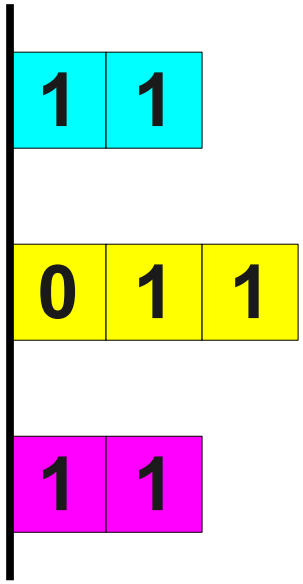
5: Write < on track 4

6: Move left until {>} on track 4.

7: Move right until {×} on track 5.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	1	<															…
											×							…

0: Push 1 onto Stack 3.



0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

5: Write < on track 4

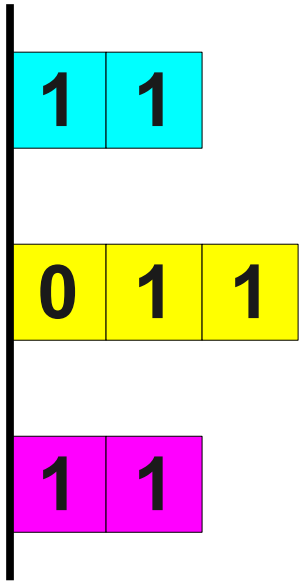
6: Move left until {>} on track 4.

7: Move right until {×} on track 5.

8: Write B on track 5.

1	1	0	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	…
>	1	1	<															…
>	0	1	1	<														…
>	1	1	<															…
																		…

0: Push 1 onto Stack 3.



0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

5: Write < on track 4

6: Move left until {>} on track 4.

7: Move right until {×} on track 5.

8: Write B on track 5.

Multitape TM Efficiency

- We can simulate a multitape TM using multiple stacks (we saw this in our discussion of **WB** programs).
- We can simulate a TM with multiple stacks using a TM with multiple tracks.
- Time to push or pop a stack is determined by
  - How many elements are on that stack, and
  - Where the read head is on the tape.

Multitape TM Efficiency

- **Lemma**: After k steps, a multitrack TM with any number of stacks has the tape head at most k steps to the right of the start position and has at most k elements pushed on the stack.
- **Corollary**: The time required to simulate one move of a multitape TM at time k is $O(k)$.
- **Theorem**: If there is a multitape TM for L with time complexity $f(n)$, there is a single-tape TM for L with time complexity $O(f(n)^2)$.
- **Proof Sketch**: At most $O(f(n))$ work is required to simulate any move of the multitape TM, because there are at most $f(n)$ moves made. Doing $O(f(n))$ work $f(n)$ times requires time at most $O(f(n)^2)$. ■

More Impressive Results

- What is the connection between the big-O notation we're used to for real computers and the time complexity of Turing machines?
- **Theorem**: Any algorithm written on a standard computer that does not use multiplications or bit shifts and runs in time $f(n)$ can be simulated by a single-tape TM in time $O(f(n)^6)$.
- Proof involves building up a simulator for standard computers using TMs; talk to me if you'd like a reference.

Time Complexity

- Armed with big-O notation, we can start to define different complexity classes.
- The **time complexity class** TIME(f(n)) is the set of languages decidable by a TM with runtime O(f(n)).

TIME(n)

- All regular languages are in TIME(n)
  - Build a DFA for a regular language.
  - Convert the DFA into a TM.
  - Accepts in time at most n + 1.
- Very difficult result: Any language in TIME(n) is regular.
  - (This is why we can't build a single-tape TM for *BALANCE* that runs in O(n) time.)

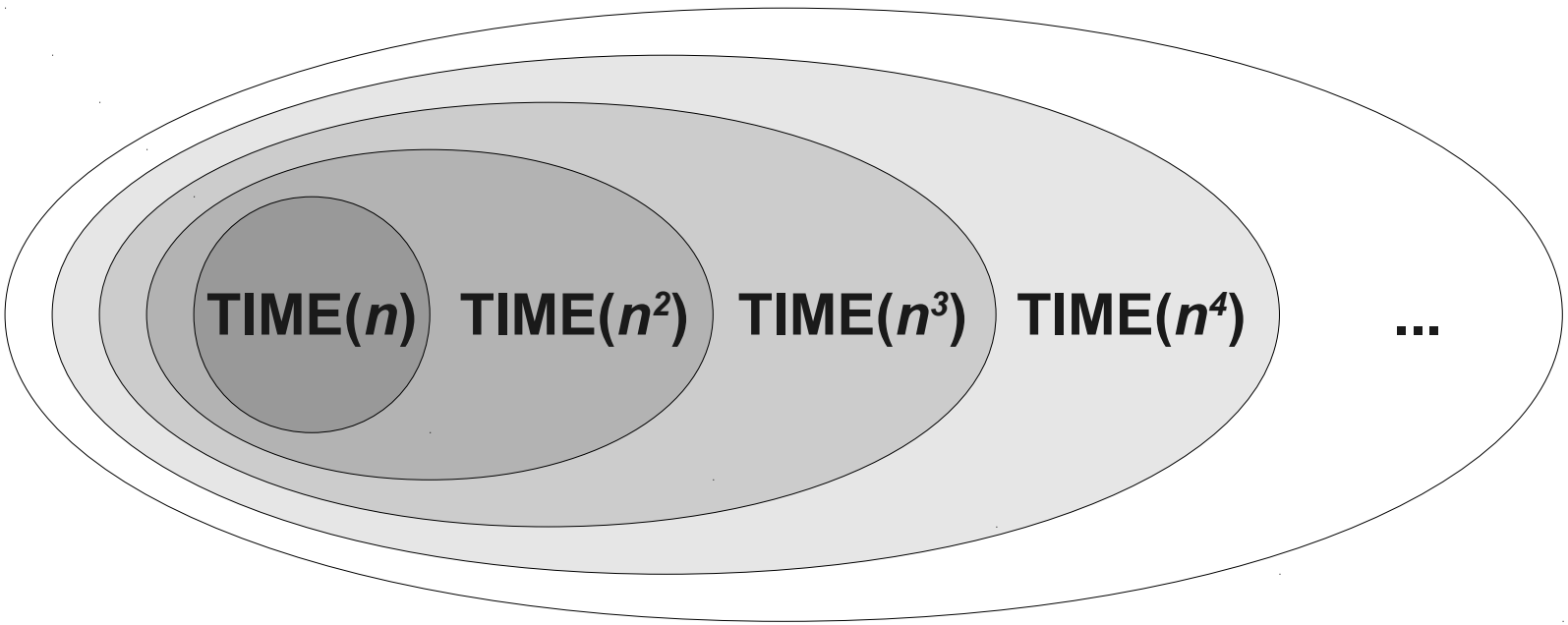
TIME(n^2)

- The language of palindromes is in TIME(n^2)
  - Snake back and forth across the tape checking whether the ends match.
- The language *NE* from Problem Set 7 is in TIME(n^2)
  - Keep checking characters from the two strings against one another.
- Any language in TIME(n) is also in TIME(n^2).
  - Since it takes at most O(n) time, it also takes at most O(n^2) time as well.

Are some problems inherently harder than others?

Weak Time Hierarchy Theorem

- **Theorem**: For any natural number $k > 1$, $\mathrm{TIME}(n^{k-1}) \subset \mathrm{TIME}(n^k)$.
- There exist infinitely many problems that require a certain amount of resources to solve.



Sketch of the Proof

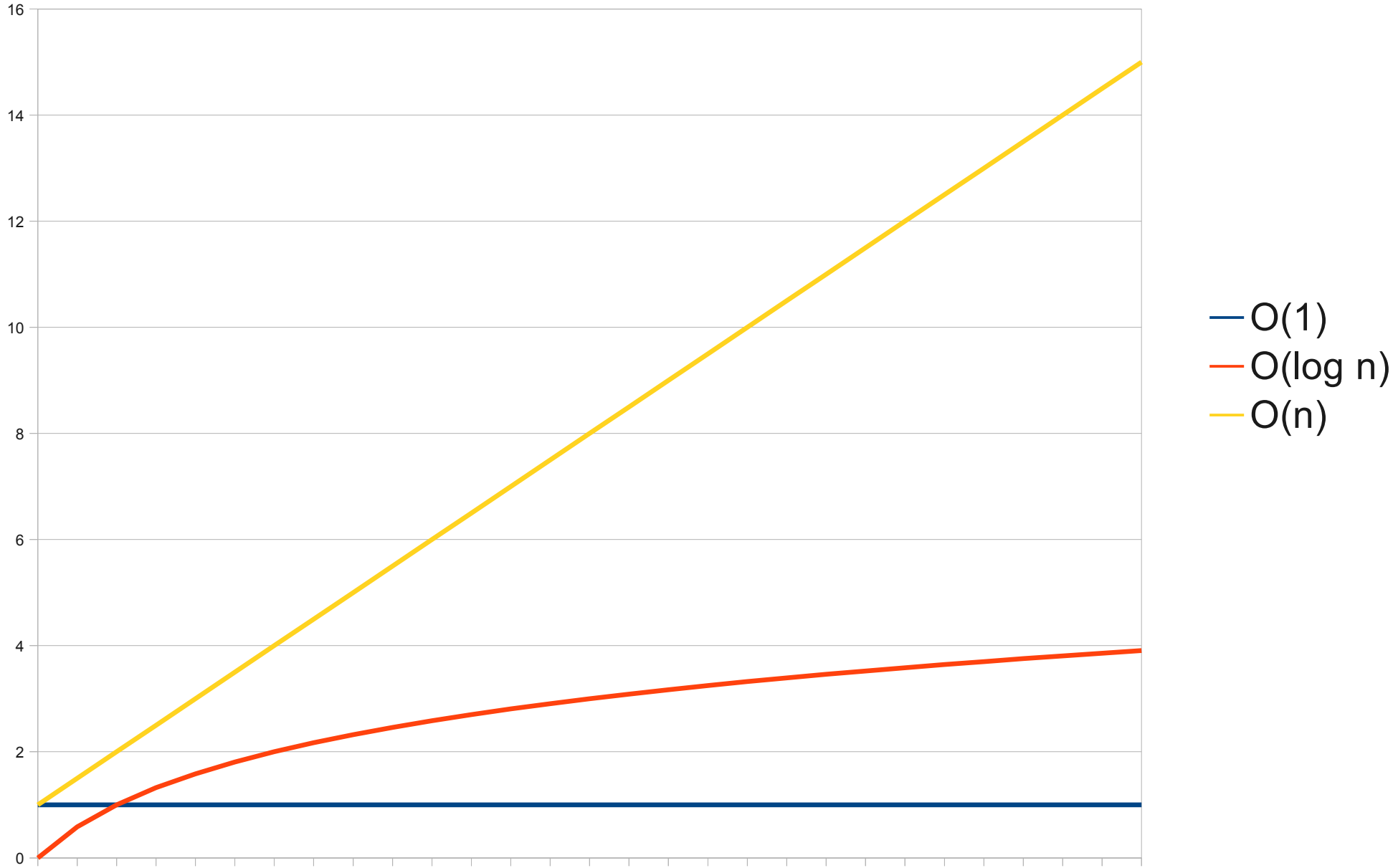
- The proof is (surprise!) another diagonalization.
- A language we can decide in time $O(n^k)$ but not time $O(n^{k-1})$ is the language

$$\{ \langle M, r \rangle \mid M \text{ rejects } \langle M \rangle \text{ within time } |r|^{k-1} \}$$

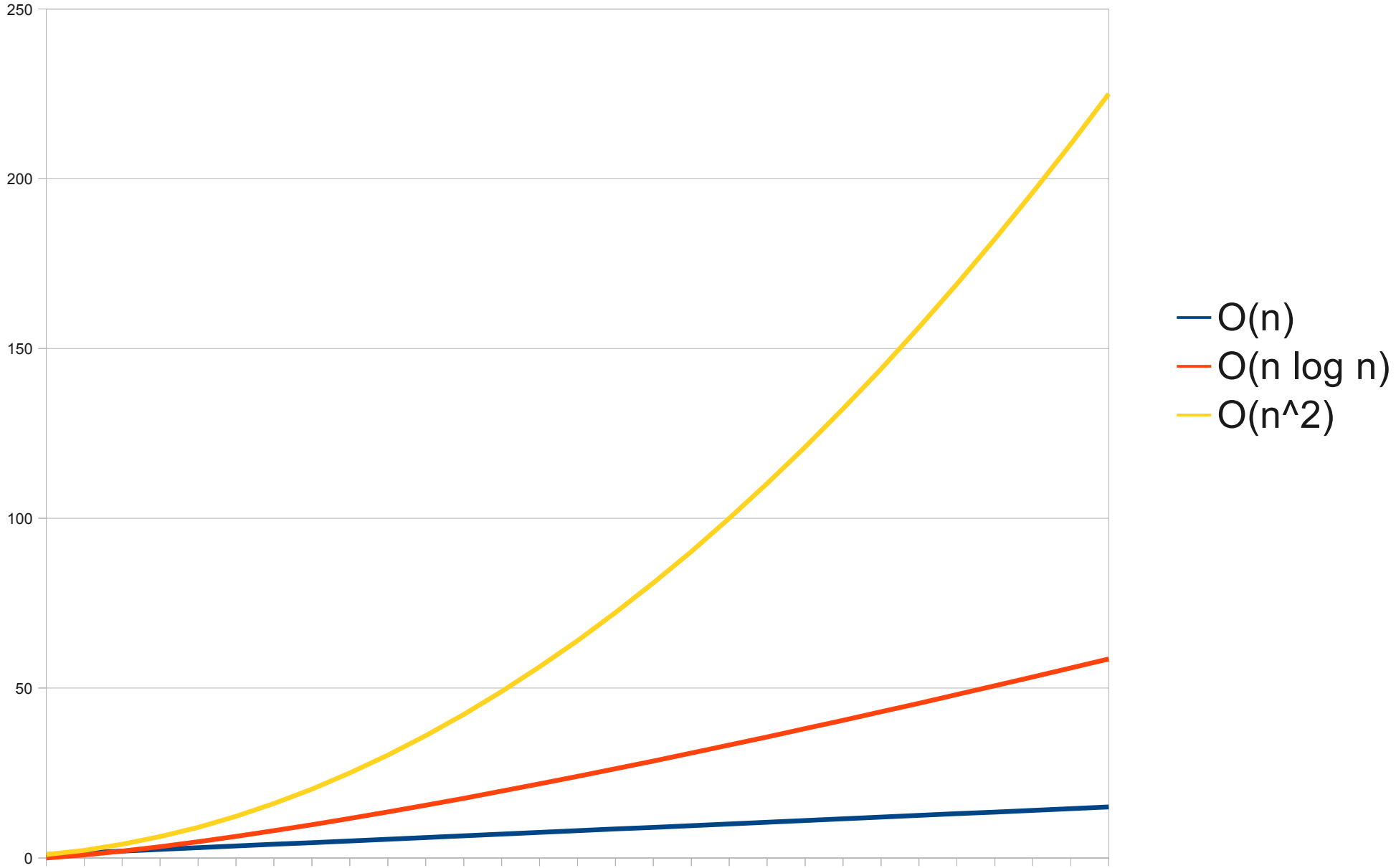
- The universal Turing machine can simulate M on w for r^{k-1} steps in time $O(r^k)$.
- Any $O(n^{k-1})$-time decider for this language breaks when run on its encoding followed by appropriate padding.
- Lots of tricky details to get right; read Sipser (Chapter 9) for more specifics.

What is Efficiency?

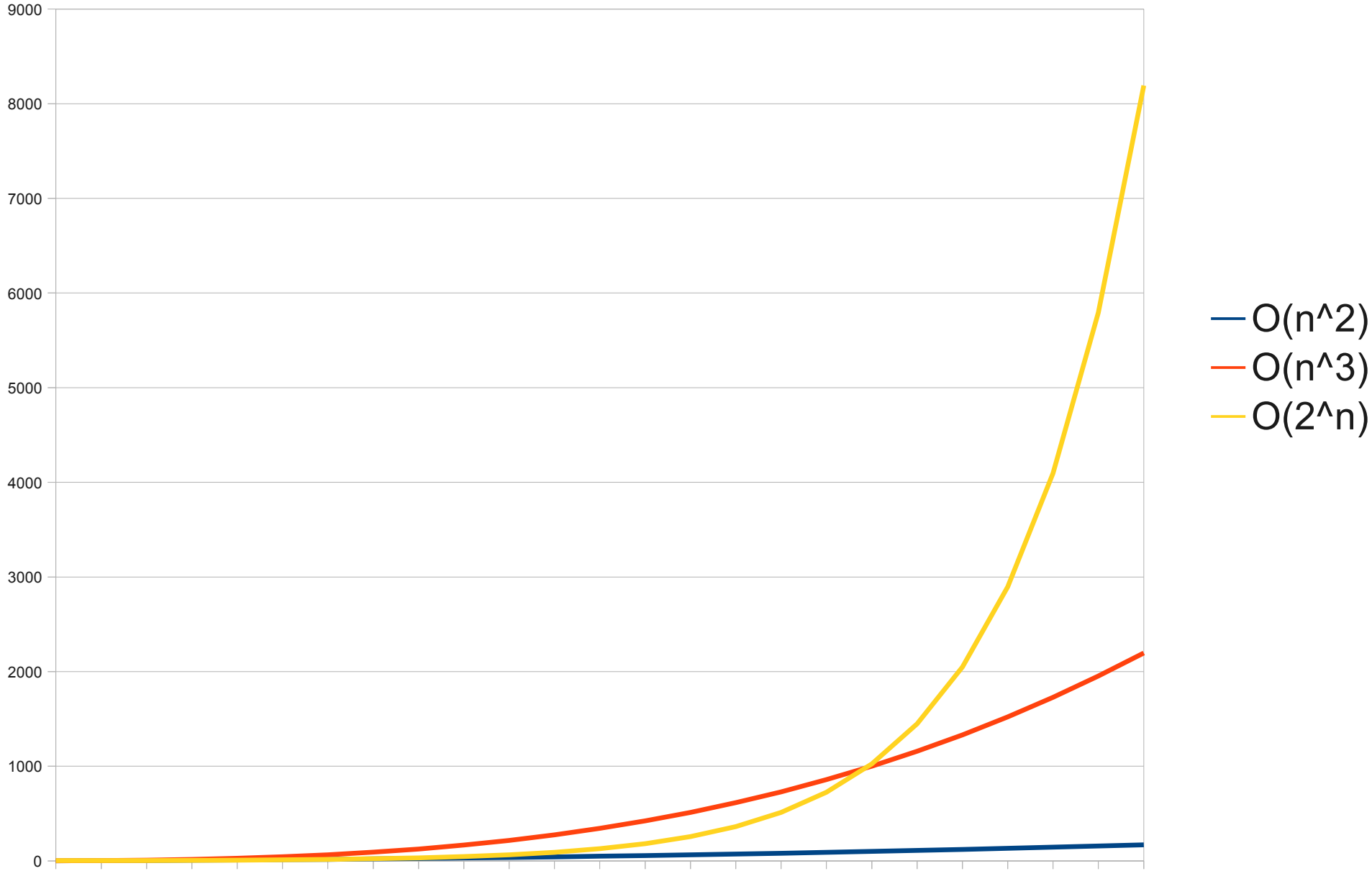
Growth Rates, Part One



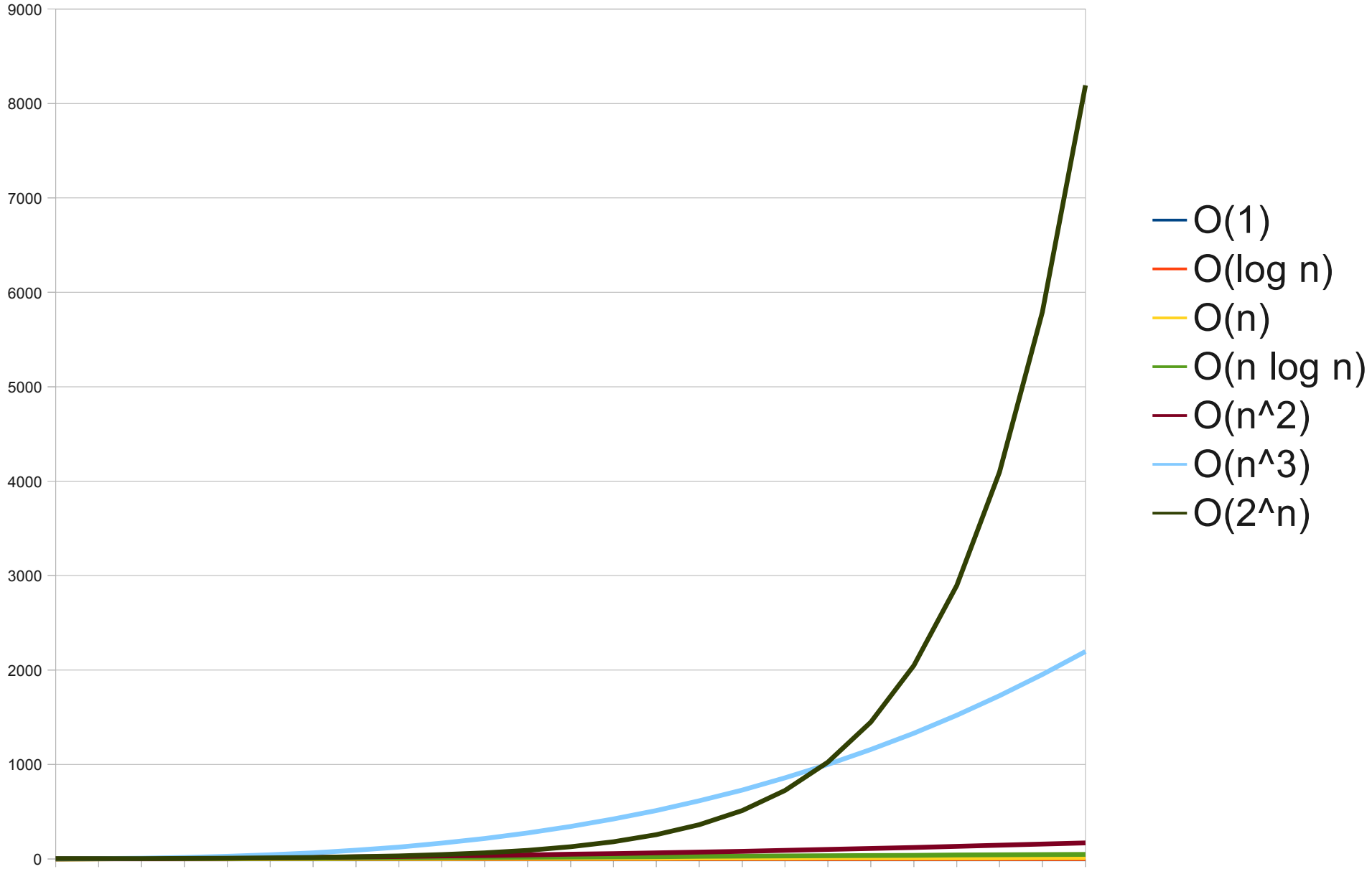
Growth Rates, Part Two



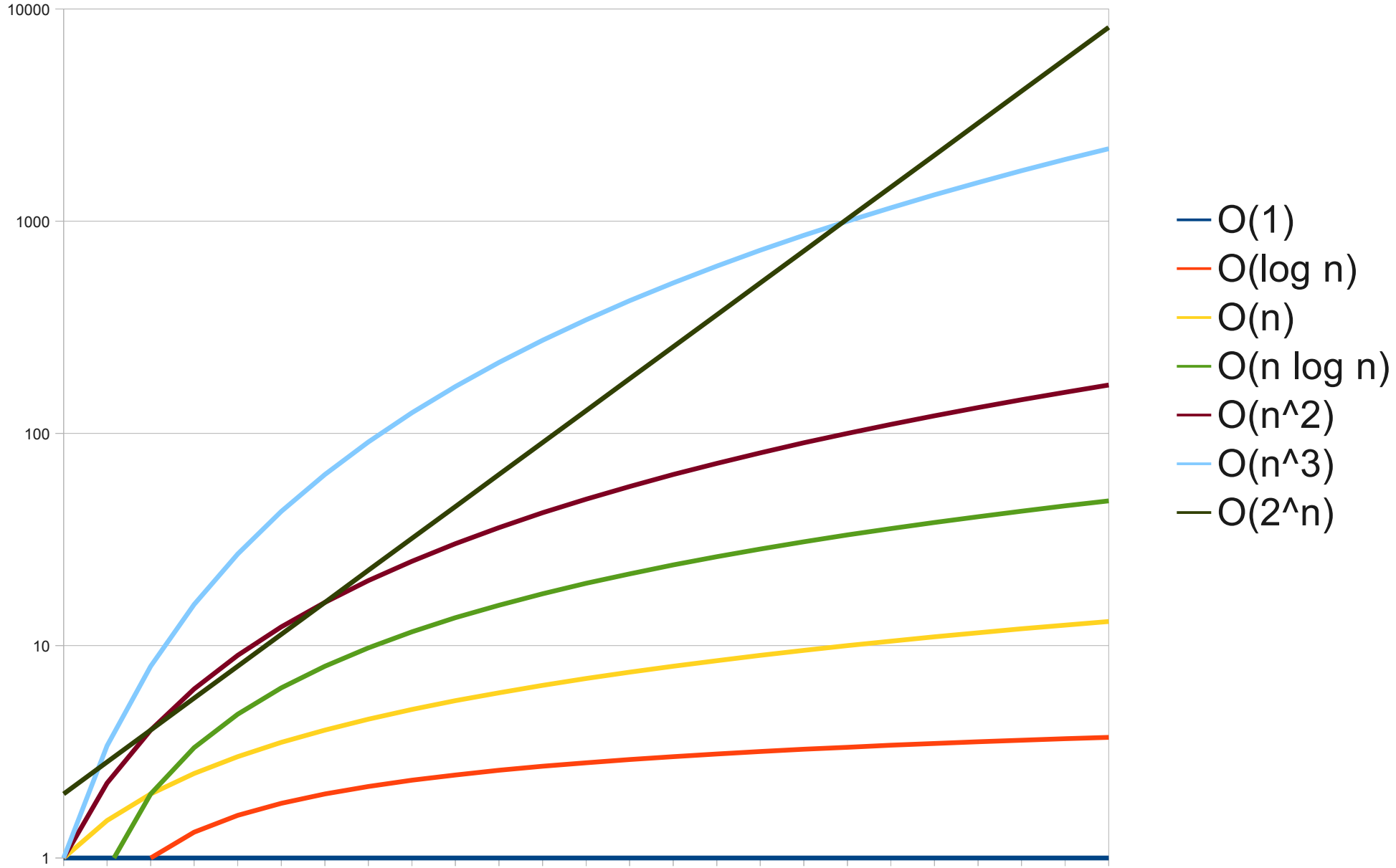
Growth Rates, Part Three



To Give You A Better Sense...



Once More with Logarithms



Comparison of Runtimes

(1 operation = 1 microsecond)

Size	1	lg n	n	n log n	n^2	n^3	2^n
100	1μs	7μs	100μs	0.7ms	10ms	<1min	40 quadrillion yrs
200	1μs	8μs	200μs	1.5ms	40ms	<1min	More than that
300	1μs	8μs	300μs	2.5ms	90ms	1min	
400	1μs	9μs	400μs	3.5ms	160ms	2min	
500	1μs	9μs	500μs	4.5ms	250ms	4min	
600	1μs	9μs	600μs	5.5ms	360ms	6min	
700	1μs	9μs	700μs	6.6ms	490ms	9min	
800	1μs	10μs	800μs	7.7ms	640ms	12min	
900	1μs	10μs	900μs	8.8ms	810ms	17min	
1000	1μs	10μs	1000μs	10ms	1000ms	22min	
1100	1μs	10μs	1100μs	11ms	1200ms	29min	
1200	1μs	10μs	1200μs	12ms	1400ms	37min	
1300	1μs	10μs	1300μs	13ms	1700ms	45min	
1400	1μs	10μs	1400μs	15ms	2000ms	56min	

Polynomials and Exponentials

- 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

The Cobham-Edmonds Thesis

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

The Cobham-Edmonds Thesis

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

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

The Cobham-Edmonds Thesis

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

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

Equivalently, $L \in \mathrm{TIME}(n^k)$ for some $k \in \mathbb{N}$

The Cobham-Edmonds Thesis

The Cobham-Edmonds Thesis

- Efficient runtimes:
  - $4n + 13$
  - $n^3 – 2n^2 + 4n$
  - $n \log \log n$

The Cobham-Edmonds Thesis

- Efficient runtimes:
  - $4n + 13$
  - $n^3 – 2n^2 + 4n$
  - $n \log \log n$
- Inefficient runtimes:
  - 2^n
  - $n!$
  - n^n

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

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**

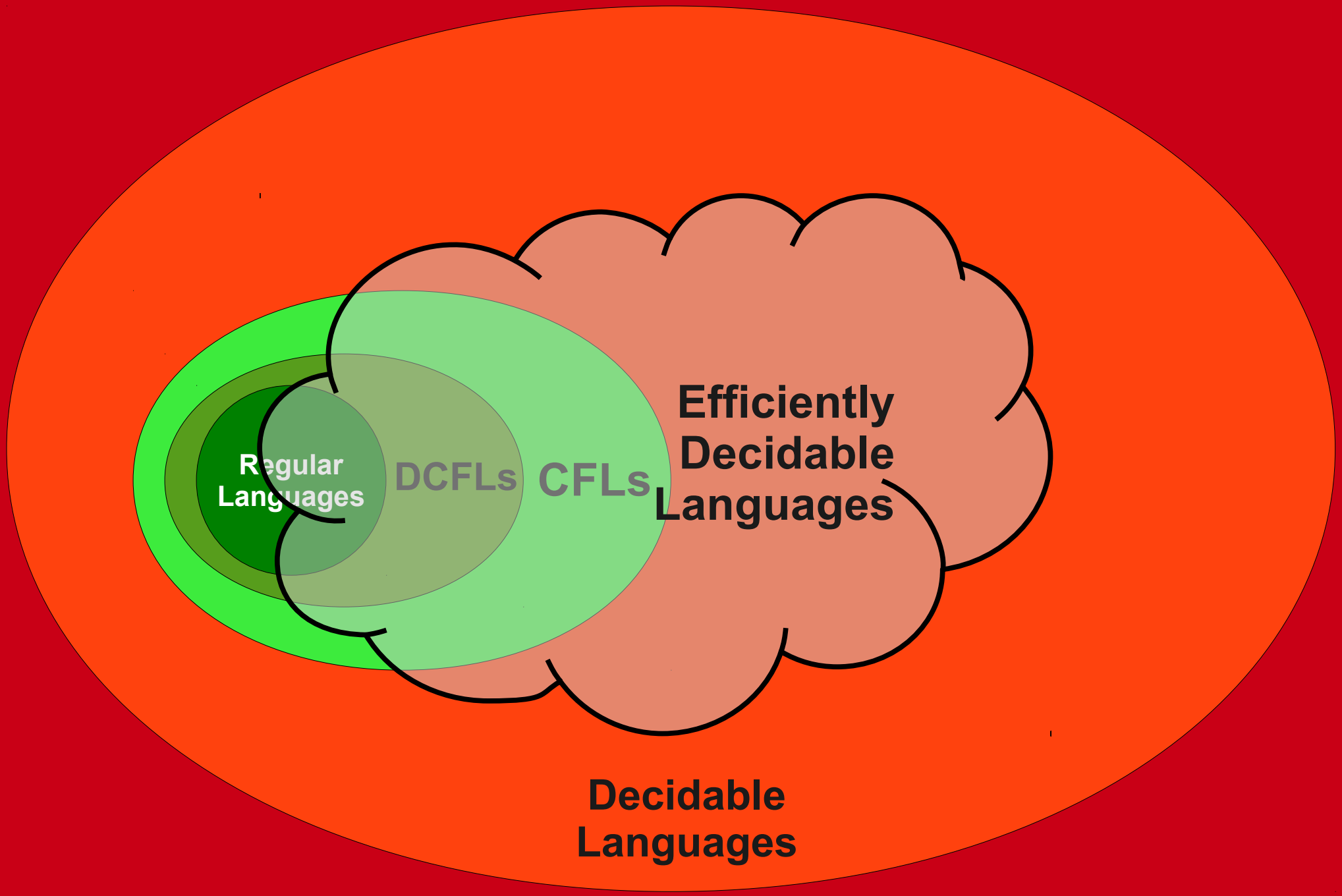
- The **complexity class P** contains all problems that can be solved in polynomial time.
- Formally:

$$\mathbf{P} = \bigcup_{k=0}^{\infty} \mathrm{TIME}(n^k)$$

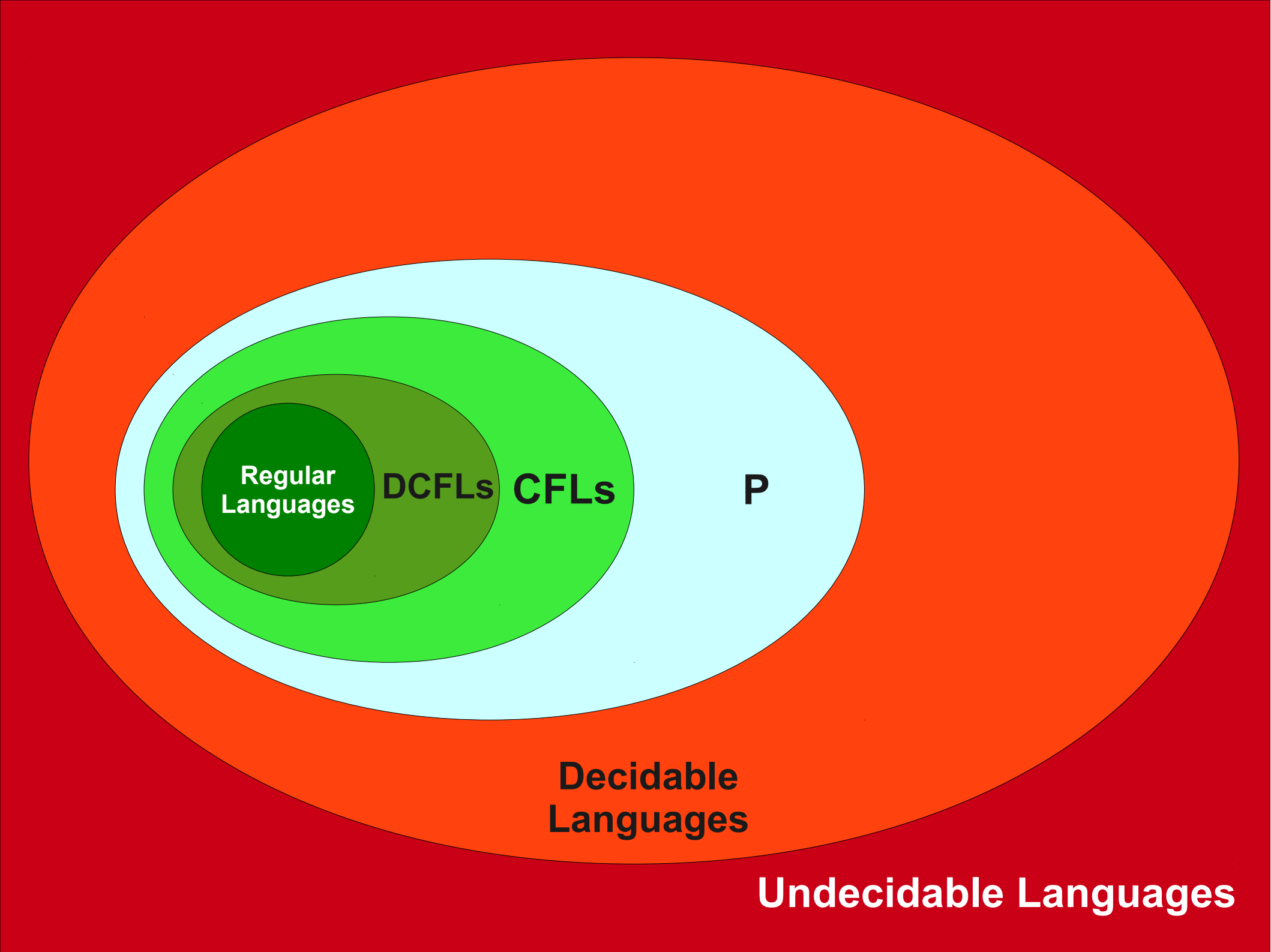
- Using our definition, a problem can be solved efficiently iff it is in **P**.

Examples of Problems in **P**

- All regular languages are in **P**.
  - Can be decided in O(n).
- All DCFLs are in **P**.
  - Can simulate a DPDA in O(n) on a two-tape TM, so can simulate DPDA on a single-tape TM in $O(n^2)$.
- All CFLs are in **P**.
  - Using more advanced techniques (the **CYK algorithm** or **Earley's algorithm**), can solve in polynomial time.
- Many other problems are in **P**:
  - *POWER2*
  - *SEARCH*



Undecidable Languages


Regular Languages
DCFLs
CFLs
P
Decidable Languages
Undecidable Languages

Problems in **P**

- **Graph connectivity:**

  Given a graph G and nodes s and t,
  is there a path from s to t?

- **Primality testing:**

  Given a number n, is n prime?

- **Maximum matching:**

  Given a set of tasks and workers who can perform those tasks, can all of the tasks be completed in under n hours?

Problems in **P**

- **Remoteness testing:**

  Given a graph *G*, are all of the nodes in *G* within distance at most *k* of one another?

- **Linear programming:**

  Given a linear set of constraints and linear objective function, is the optimal solution at least *n*?

- **Edit distance:**

  Given two strings, can the strings be transformed into one another in at most *n* single-character edits?

A Feel For Polynomial Time

- What can you do in polynomial time?
- What can you not do in polynomial time?
- Let's see some examples.

Closure under Addition

- The sum of two polynomial-bounded functions is itself a polynomial-bounded function:
  - **Theorem**: $O(n^k) + O(n^r) = O(n^{\max\{k, r\}})$.
- If you have two programs that each run in polynomial time, running them in sequence still stays within polynomial time.

```
function newCode() {
    polynomialFunctionOne();
    polynomialFunctionTwo();
}
```

Closure under Multiplication

- The product of two polynomial-bounded functions is itself a polynomial-bounded function:
  - **Theorem**: $O(n^k)\ O(n^r) = O(n^{k+r})$.
- Doing polynomial work polynomially many times stays polynomial.

```
for (int i = 0; i < poly(); i++) {
    polynomialFunction();
}
```

Closure under Composition

- The composition of polynomials (applying one polynomial to another) is itself a polynomial:
  - **Theorem**: If $f(n) = O(n^k)$ and $g(n) = O(n^r)$, then $f(g(n)) = O(n^{kr})$.
- Calling one polynomial function on the result of another stays polynomial:

```
function newCode() {
    polynomial2(polynomial1());
}
```

Other Models of Computation

- All models of computation that we've talked about so far (except for the nondeterministic TM) can be reduced to a TM in polynomial time.
- **Theorem**: $L \in$ **P** iff there is a polynomial-time TM, **WB*n*** program, multitape TM, or normal computer program (except for multiplication/bitshifting) for it.
- Essentially – a problem is in **P** iff you could solve it on a normal computer without doing multiplies or bitshifts.

Proving Languages are in **P**

- To prove that a language is regular, we could
  - Design a DFA for it.
  - Design an NFA for it.
  - Design a regular expression for it.
  - Use closure properties.
- To prove that a language is a CFL, we could
  - Design a CFG for it.
  - Design a PDA for it.
  - Use closure properties.
- How do we prove that a language is in **P**?

Proving Languages are in **P**

- **Directly prove the language is in P**.
  - Build a decider for the language L.
  - Prove that the decider runs in time $O(n^k)$.
- **Use closure properties**.
  - Prove that the language can be formed by appropriate transformations of languages in P.
- **Reduce the language to a language in P**.
  - Show how a polynomial-time decider for some language L' can be used to decide L.

Proving Languages are in **P**

- **Directly prove the language is in P**.
  - Build a decider for the language L.
  - Prove that the decider runs in time $O(n^k)$.

Use closure properties.

Prove that the language can be formed by appropriate transformations of languages in P.

Reduce the language to a language in P.

Show how a polynomial-time decider for some language L' can be used to decide L.

SEARCH is in **P**

- M = “On input p?t:
  - Copy p onto one tape of the TM (the **pattern tape**)
  - Copy t onto another tape of the TM (the **text tape**)
  - Move the tape heads for the pattern and text tapes back to the start.
  - Repeat the following:
    - Check whether the contents of the pattern tape appear at the current position on the text tape.
    - If so, accept.
    - Otherwise, back the tape head for the pattern tape to the beginning.
    - Advance the tape head in the text tape, rejecting if the end of the text is found.

SEARCH is in **P**

- M = “On input p?t:
  - Copy p onto one tape of the TM (the **pattern tape**)
  - Copy t onto another tape of the TM (the **text tape**)
  - Move the tape heads for the pattern and text tapes back to the start.
  - Repeat the following:
    - Check whether the contents of the pattern tape appear at the current position on the text tape.
    - If so, accept.
    - Otherwise, back the tape head for the pattern tape to the beginning.
    - Advance the tape head in the text tape, rejecting if the end of the text is found.

O(n) steps

SEARCH is in **P**

- M = “On input p?t:
  - Copy p onto one tape of the TM (the **pattern tape**)
  - Copy t onto another tape of the TM (the **text tape**)
  - Move the tape heads for the pattern and text tapes back to the start.
  - Repeat the following:
    - Check whether the contents of the pattern tape appear at the current position on the text tape.
    - If so, accept.
    - Otherwise, back the tape head for the pattern tape to the beginning.
    - Advance the tape head in the text tape, rejecting if the end of the text is found.

O(n) steps

This takes O(n) time, because we need to verify the structure of the input.

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**)
  - Move the tape heads for the pattern and text tapes back to the start.
  - Repeat the following:
    - Check whether the contents of the pattern tape appear at the current position on the text tape.
    - If so, accept.
    - Otherwise, back the tape head for the pattern tape to the beginning.
    - Advance the tape head in the text tape, rejecting if the end of the text is found.

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**) **O(|t|) steps**
  - Move the tape heads for the pattern and text tapes back to the start.
  - Repeat the following:
    - Check whether the contents of the pattern tape appear at the current position on the text tape.
    - If so, accept.
    - Otherwise, back the tape head for the pattern tape to the beginning.
    - Advance the tape head in the text tape, rejecting if the end of the text is found.

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**) **O(|t|) steps**
  - Move the tape heads for the pattern and text tapes back to the start. **O(|p| + |t|) steps**
  - Repeat the following:
    - Check whether the contents of the pattern tape appear at the current position on the text tape.
    - If so, accept.
    - Otherwise, back the tape head for the pattern tape to the beginning.
    - Advance the tape head in the text tape, rejecting if the end of the text is found.

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**) **O(|t|) steps**
  - Move the tape heads for the pattern and text tapes back to the start. **O(|p| + |t|) steps**
  - Repeat the following:
    - Check whether the contents of the pattern tape appear at the current position on the text tape. **O(|p|) steps**
    - If so, accept.
    - Otherwise, back the tape head for the pattern tape to the beginning.
    - Advance the tape head in the text tape, rejecting if the end of the text is found.

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**) **O(|t|) steps**
  - Move the tape heads for the pattern and text tapes back to the start. **O(|p| + |t|) steps**
  - Repeat the following:
    - Check whether the contents of the pattern tape appear at the current position on the text tape. **O(|p|) steps**
    - If so, accept. **O(1) steps**
    - Otherwise, back the tape head for the pattern tape to the beginning.
    - Advance the tape head in the text tape, rejecting if the end of the text is found.

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**) **O(|t|) steps**
  - Move the tape heads for the pattern and text tapes back to the start. **O(|p| + |t|) steps**
  - Repeat the following:
    - Check whether the contents of the pattern tape appear at the current position on the text tape. **O(|p|) steps**
    - If so, accept. **O(1) steps**
    - Otherwise, back the tape head for the pattern tape to the beginning. **O(|p|) steps**
    - Advance the tape head in the text tape, rejecting if the end of the text is found.

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**) **O(|t|) steps**
  - Move the tape heads for the pattern and text tapes back to the start. **O(|p| + |t|) steps**
  - Repeat the following:
    - Check whether the contents of the pattern tape appear at the current position on the text tape. **O(|p|) steps**
    - If so, accept. **O(1) steps**
    - Otherwise, back the tape head for the pattern tape to the beginning. **O(|p|) steps**
    - Advance the tape head in the text tape, rejecting if the end of the text is found. **O(1) steps**

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**) **O(|t|) steps**
  - Move the tape heads for the pattern and text tapes back to the start. **O(|p| + |t|) steps**
  - Repeat the following: **O(|p| |t|) steps**
    - Check whether the contents of the pattern tape appear at the current position on the text tape. **O(|p|) steps**
    - If so, accept. **O(1) steps**
    - Otherwise, back the tape head for the pattern tape to the beginning. **O(|p|) steps**
    - Advance the tape head in the text tape, rejecting if the end of the text is found. **O(1) steps**

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**) **O(|t|) steps**
  - Move the tape heads for the pattern and text tapes back to the start. **O(|p| + |t|) steps**
  - Repeat the following: **O(|p| |t|) steps**
    - Check whether the contents of the pattern tape appear at the current position on the text tape. **O(|p|) steps**
    - If so, accept. **O(1) steps**
    - Otherwise, back the tape head for the pattern tape to the beginning. **O(|p|) steps**
    - Advance the tape head in the text tape, rejecting if the end of the text is found. **+ O(1) steps**

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**) **O(|t|) steps**
  - Move the tape heads for the pattern and text tapes back to the start. **O(|p| + |t|) steps**
  - Repeat the following: **O(|p| |t|) steps**
    - Check whether the contents of the pattern tape appear at the current position on the text tape. **O(|p|) steps**
    - If so, accept. **O(1) steps**
    - Otherwise, back the tape head for the pattern tape to the beginning. **O(|p|) steps**
    - Advance the tape head in the text tape, rejecting if the end of the text is found. **+ O(1) steps**

O(|p| |t| + |p| + |t|) steps

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**) **O(|t|) steps**
  - Move the tape heads for the pattern and text tapes back to the start. **O(|p| + |t|) steps**
  - Repeat the following: **O(|p| |t|) steps**
    - Check whether the contents of the pattern tape appear at the current position on the text tape. **O(|p|) steps**
    - If so, accept. **O(1) steps**
    - Otherwise, back the tape head for the pattern tape to the beginning. **O(|p|) steps**
    - Advance the tape head in the text tape, rejecting if the end of the text is found. **+ O(1) steps**

$O(n^2 + n)$ steps

SEARCH is in **P**

- M = “On input p?t: **O(n) steps**
  - Copy p onto one tape of the TM (the **pattern tape**) **O(|p|) steps**
  - Copy t onto another tape of the TM (the **text tape**) **O(|t|) steps**
  - Move the tape heads for the pattern and text tapes back to the start. **O(|p| + |t|) steps**
  - Repeat the following: **O(|p| |t|) steps**
    - Check whether the contents of the pattern tape appear at the current position on the text tape. **O(|p|) steps**
    - If so, accept. **O(1) steps**
    - Otherwise, back the tape head for the pattern tape to the beginning. **O(|p|) steps**
    - Advance the tape head in the text tape, rejecting if the end of the text is found. **+ O(1) steps**

$O(n^2)$ steps

Proving Languages are in **P**

- **Directly prove the language is in P**.
  - Build a decider for the language L.
  - Prove that the decider runs in time $O(n^k)$.
- **Use closure properties**.
  - Prove that the language can be formed by appropriate transformations of languages in P.
- **Reduce the language to a language in P**.
  - Show how a polynomial-time decider for some language L' can be used to decide L.

Proving Languages are in **P**

Directly prove the language is in P.

Build a decider for the language L.

Prove that the decider runs in time $O(n^k)$.

- **Use closure properties.**
  - Prove that the language can be formed by appropriate transformations of languages in P.

Reduce the language to a language in P.

Show how a polynomial-time decider for some language L' can be used to decide L.

Closure Properties of **P**

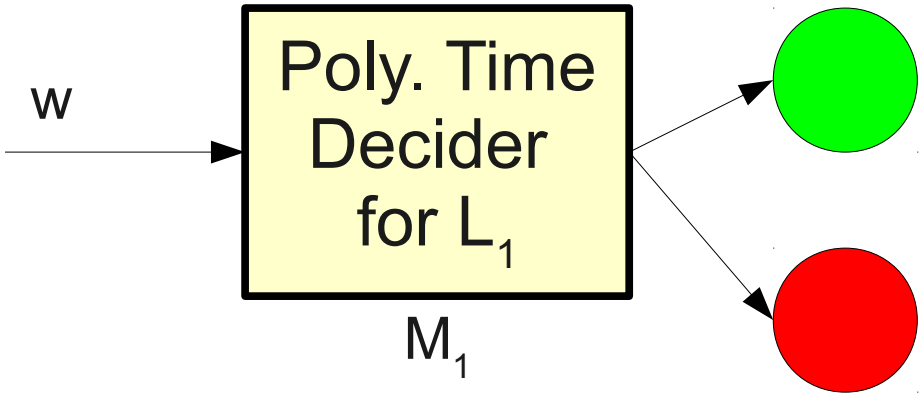
- The complexity class **P** is closed under many familiar operations:
  - Union
  - Intersection
  - Complement
  - Difference
  - Reversal
  - Concatenation
  - Kleene Star

Closure Properties of **P**

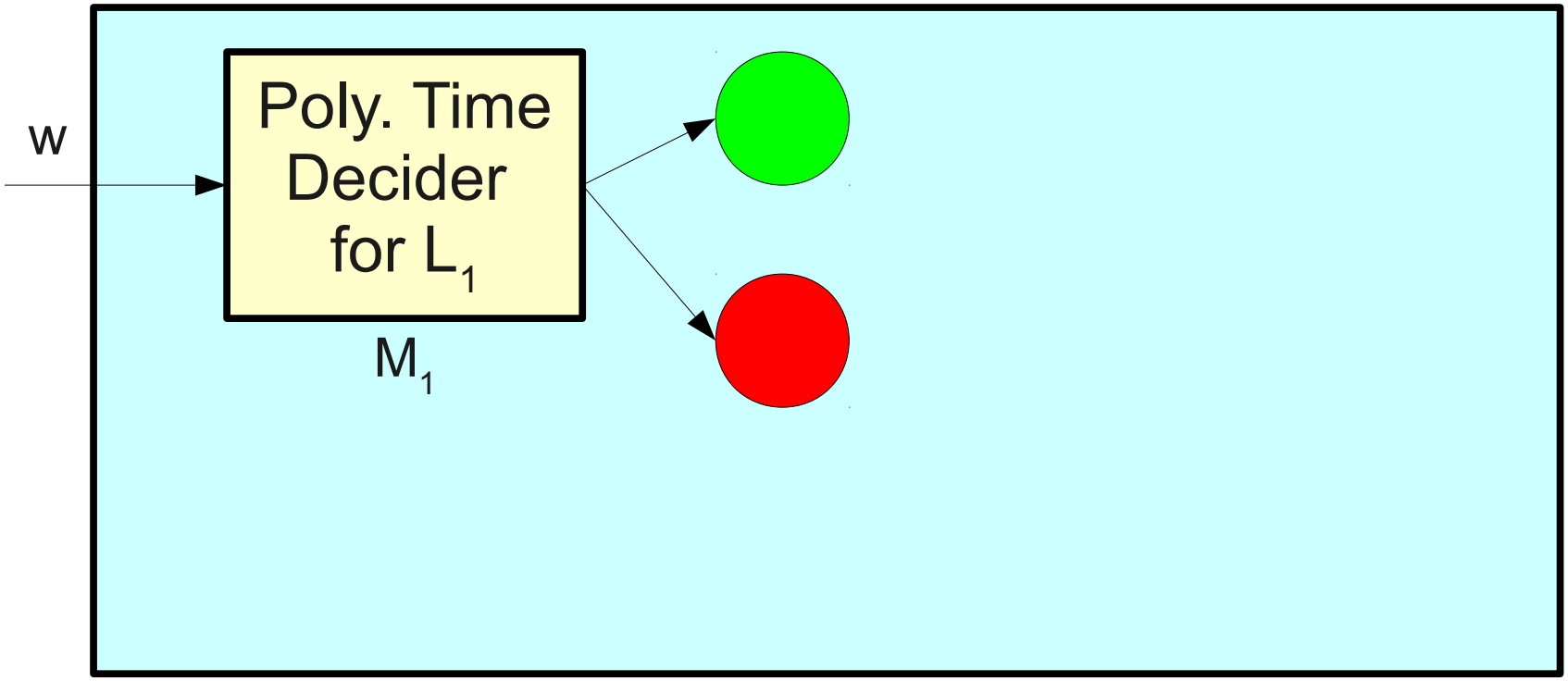
- The complexity class **P** is closed under many familiar operations:
  - **Union**
  - Intersection
  - Complement
  - Difference
  - Reversal
  - Concatenation
  - Kleene Star

P is Closed Under Union

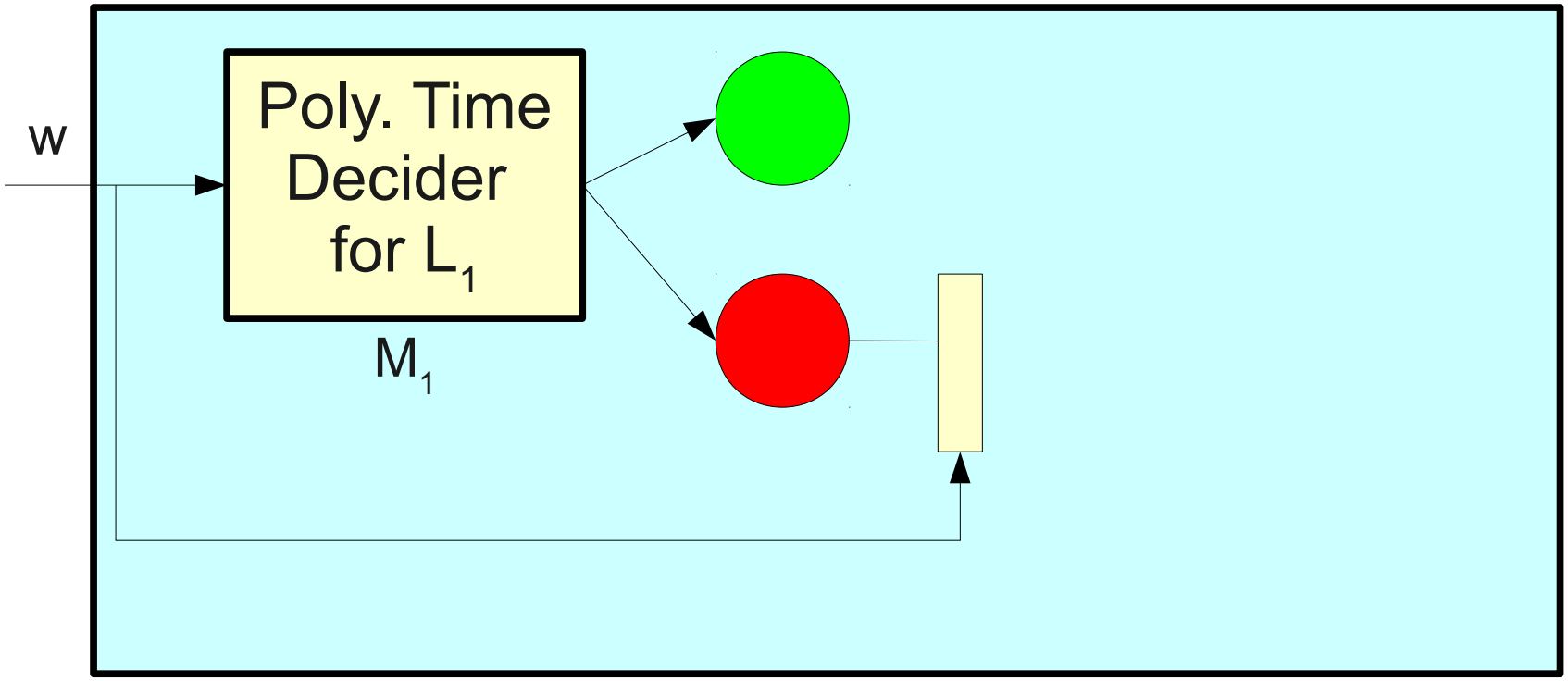
P is Closed Under Union



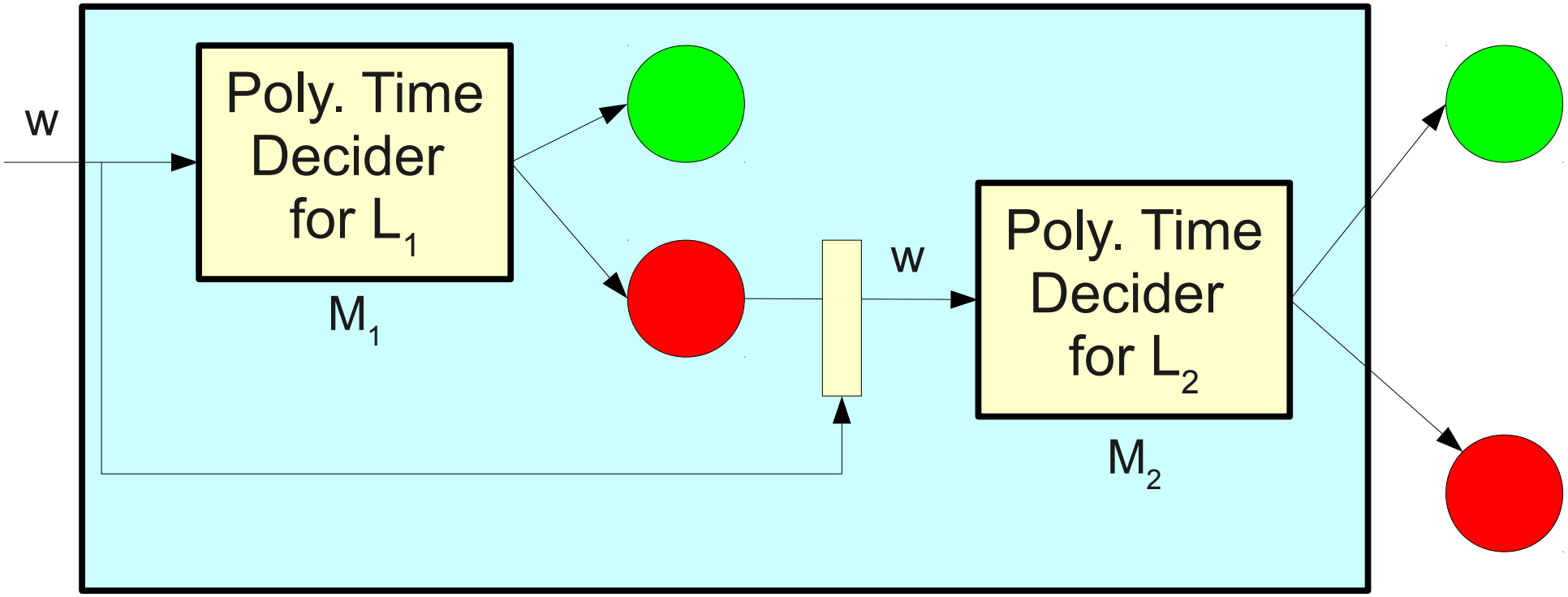
P is Closed Under Union



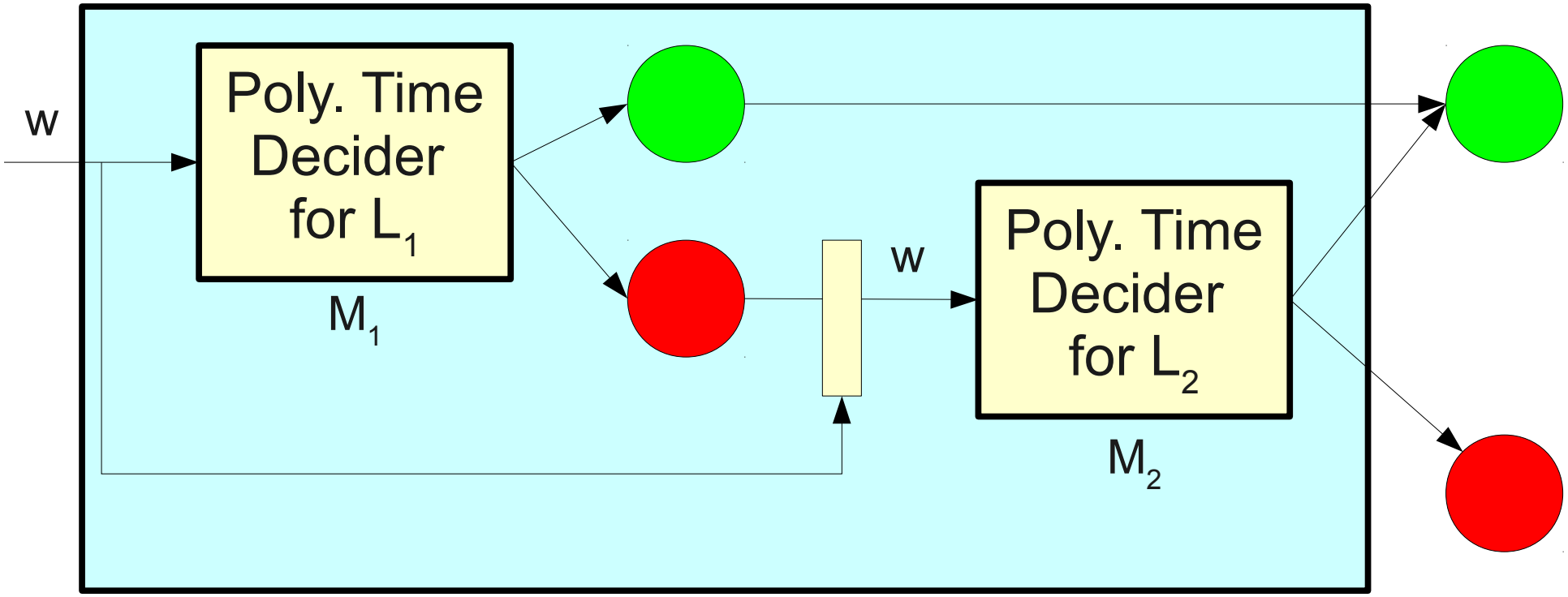
P is Closed Under Union



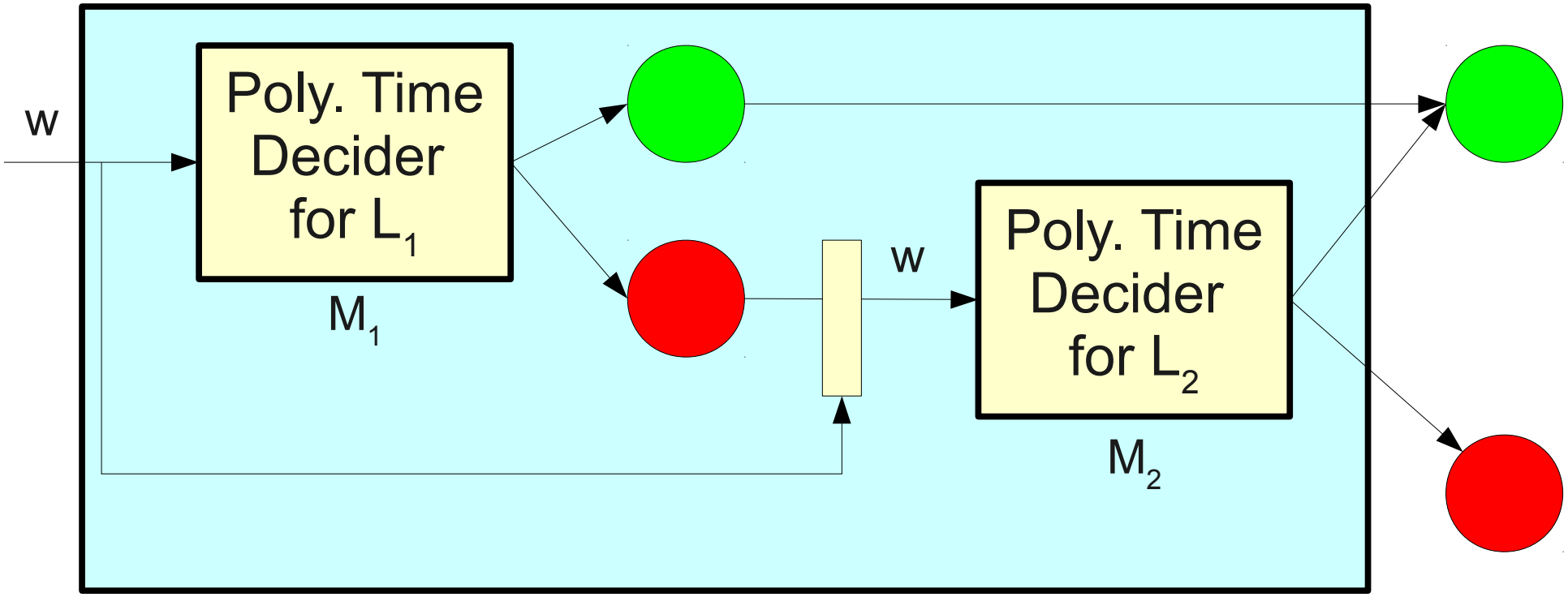
P is Closed Under Union



P is Closed Under Union



P is Closed Under Union



M' = “On input w:

Run M_1 on w.

If M_1 accepts, accept.

Otherwise, run M_2 on w.

If M_2 accepts, accept.

Otherwise, reject.”

Theorem: P is closed under union.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively.

Just as languages in R have deciders and languages in RE have recognizers, since L_1 and L_2 are in P, they must have polynomial-time deciders.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$.

Note that our goal is to show that this machine is a polynomial-time decider, not just a regular decider or TM.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$. To see that M' is a polynomial-time decider, note that when we run M_1 on w, it takes $O(n^k)$ steps for some $k \in \mathbb{N}$.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$. To see that M' is a polynomial-time decider, note that when we run M_1 on w, it takes $O(n^k)$ steps for some $k \in \mathbb{N}$. If M_1 accepts, M' accepts in $O(n^k)$.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$. To see that M' is a polynomial-time decider, note that when we run M_1 on w, it takes $O(n^k)$ steps for some $k \in \mathbb{N}$. If M_1 accepts, M' accepts in $O(n^k)$. Otherwise, we run M_2 on w.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$. To see that M' is a polynomial-time decider, note that when we run M_1 on w, it takes $O(n^k)$ steps for some $k \in \mathbb{N}$. If M_1 accepts, M' accepts in $O(n^k)$. Otherwise, we run M_2 on w. This takes $O(n^r)$ steps for some $r \in \mathbb{N}$.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$. To see that M' is a polynomial-time decider, note that when we run M_1 on w, it takes $O(n^k)$ steps for some $k \in \mathbb{N}$. If M_1 accepts, M' accepts in $O(n^k)$. Otherwise, we run M_2 on w. This takes $O(n^r)$ steps for some $r \in \mathbb{N}$. The total runtime of the two machines is thus $O(n^k + n^r)$.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$. To see that M' is a polynomial-time decider, note that when we run M_1 on w, it takes $O(n^k)$ steps for some $k \in \mathbb{N}$. If M_1 accepts, M' accepts in $O(n^k)$. Otherwise, we run M_2 on w. This takes $O(n^r)$ steps for some $r \in \mathbb{N}$. The total runtime of the two machines is thus $O(n^k + n^r)$. In order to run the two machines, we must also store w so we can use it twice.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$. To see that M' is a polynomial-time decider, note that when we run M_1 on w, it takes $O(n^k)$ steps for some $k \in \mathbb{N}$. If M_1 accepts, M' accepts in $O(n^k)$. Otherwise, we run M_2 on w. This takes $O(n^r)$ steps for some $r \in \mathbb{N}$. The total runtime of the two machines is thus $O(n^k + n^r)$. In order to run the two machines, we must also store w so we can use it twice.

This detail is glossed over in the above TM description, and from a computability perspective is irrelevant. However, from a complexity point of view, it's critical to get this right.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$. To see that M' is a polynomial-time decider, note that when we run M_1 on w, it takes $O(n^k)$ steps for some $k \in \mathbb{N}$. If M_1 accepts, M' accepts in $O(n^k)$. Otherwise, we run M_2 on w. This takes $O(n^r)$ steps for some $r \in \mathbb{N}$. The total runtime of the two machines is thus $O(n^k + n^r)$. In order to run the two machines, we must also store w so we can use it twice. This can be done in O(n) time if we use two tapes, for a net runtime of $O(n^k + n^r + n)$ time on a multitape TM.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$. To see that M' is a polynomial-time decider, note that when we run M_1 on w, it takes $O(n^k)$ steps for some $k \in \mathbb{N}$. If M_1 accepts, M' accepts in $O(n^k)$. Otherwise, we run M_2 on w. This takes $O(n^r)$ steps for some $r \in \mathbb{N}$. The total runtime of the two machines is thus $O(n^k + n^r)$. In order to run the two machines, we must also store w so we can use it twice. This can be done in O(n) time if we use two tapes, for a net runtime of $O(n^k + n^r + n)$ time on a multitape TM. Thus M' is a polynomial-time decider.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$. To see that M' is a polynomial-time decider, note that when we run M_1 on w, it takes $O(n^k)$ steps for some $k \in \mathbb{N}$. If M_1 accepts, M' accepts in $O(n^k)$. Otherwise, we run M_2 on w. This takes $O(n^r)$ steps for some $r \in \mathbb{N}$. The total runtime of the two machines is thus $O(n^k + n^r)$. In order to run the two machines, we must also store w so we can use it twice. This can be done in O(n) time if we use two tapes, for a net runtime of $O(n^k + n^r + n)$ time on a multitape TM. Thus M' is a polynomial-time decider.

To see that $\mathcal{L}(M') = L_1 \cup L_2$, note that M' accepts w iff M_1 accepts w or M_2 accepts w. M_1 accepts w iff $w \in L_1$ and M_2 accepts w iff $w \in L_2$. Thus M' accepts w iff $w \in L_1$ or $w \in L_2$, so M' accepts w iff $w \in L_1 \cup L_2$. Thus $\mathcal{L}(M') = L_1 \cup L_2$.

Theorem: P is closed under union.
Proof: Suppose L_1 and L_2 are in P, and let M_1 and M_2 be polynomial-time deciders for L_1 and L_2, respectively. Then construct the machine M' as follows:

M' = "On input w:
Run M_1 on w.
If M_1 accepts, accept.
Otherwise, run M_2 on w.
If M_2 accepts, accept.
Otherwise, reject."

We show that M' is a polynomial-time decider, then show that $\mathcal{L}(M') = L_1 \cup L_2$. To see that M' is a polynomial-time decider, note that when we run M_1 on w, it takes $O(n^k)$ steps for some $k \in \mathbb{N}$. If M_1 accepts, M' accepts in $O(n^k)$. Otherwise, we run M_2 on w. This takes $O(n^r)$ steps for some $r \in \mathbb{N}$. The total runtime of the two machines is thus $O(n^k + n^r)$. In order to run the two machines, we must also store w so we can use it twice. This can be done in O(n) time if we use two tapes, for a net runtime of $O(n^k + n^r + n)$ time on a multitape TM. Thus M' is a polynomial-time decider.

To see that $\mathcal{L}(M') = L_1 \cup L_2$, note that M' accepts w iff M_1 accepts w or M_2 accepts w. M_1 accepts w iff $w \in L_1$ and M_2 accepts w iff $w \in L_2$. Thus M' accepts w iff $w \in L_1$ or $w \in L_2$, so M' accepts w iff $w \in L_1 \cup L_2$. Thus $\mathcal{L}(M') = L_1 \cup L_2$. ■

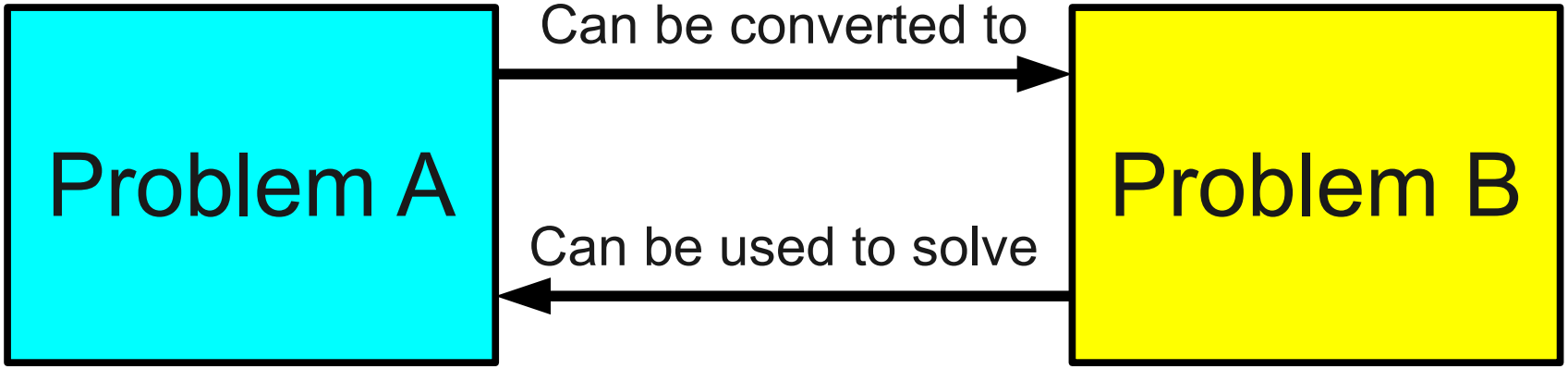
Proving Languages are in P

- **Directly prove the language is in P**.
  - Build a decider for the language L.
  - Prove that the decider runs in time $O(n^k)$.
- **Use closure properties**.
  - Prove that the language can be formed by appropriate transformations of languages in P.
- **Reduce the language to a language in P**.
  - Show how a polynomial-time decider for some language L' can be used to decide L.

Proving Languages are in P

- **Directly prove the language is in P.**
  - Build a decider for the language L.
  - Prove that the decider runs in time $O(n^k)$.
- **Use closure properties.**
  - Prove that the language can be formed by appropriate transformations of languages in P.
- **Reduce the language to a language in P.**
  - Show how a polynomial-time decider for some language L' can be used to decide L.

Reductions



If any instance of A can be converted into an instance of B, we say that A **reduces** to B.

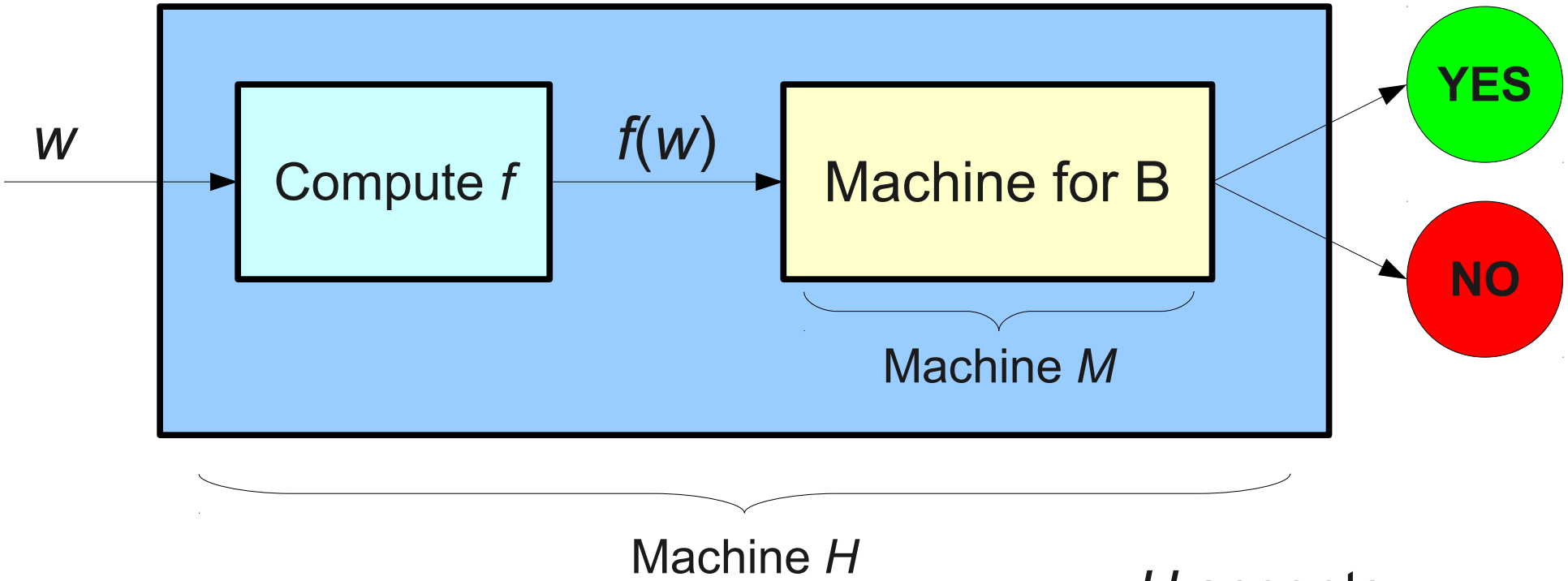
Mapping Reductions and **P**

- When studying whether problems were in **R**, **RE**, or co-**RE**, we used mapping reductions.
- We cannot use mapping reductions when talking about the class **P**.
  - The reduction can do more than polynomial work.
- We will need to introduce a new kind of reduction.

Polynomial-Time Mapping Reductions

- A function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ is a **polynomial-time mapping reduction** from A to B iff
  - f is computable in polynomial time, and
  - $w \in A$ iff $f(w) \in B$.
- If f is a polynomial-time mapping reduction from A to B, then we say that A is **polynomial-time mapping reducible** to B.
- We denote this $\mathbf{A \leq_P B}$.
- Informally:
  - Instances of A can be turned into instances of B
  - In polynomial time
  - While preserving the correct answer.

Sketch of the Proof



H = "On input w:
Compute $f(w)$.
Run M on $f(w)$.
If M accepts $f(w)$, accept w.
If M rejects $f(w)$, reject w."

H accepts w
iff
M accepts $f(w)$
iff
$f(w) \in B$
iff
$w \in A$

Doing the Math

- Initial input to A: Size n.
- After doing the polynomial-time mapping reduction: Size $O(n^k)$.
  - If we can do at most $O(n^k)$ steps, then we can't construct a string any longer than that.
- Time to run the polynomial-time decider for B on that string: Size $O(n^{kr})$.
- Overall time is still polynomial.

Theorem: If B $\in$ **P** and A $\leq_P$ B, then A $\in$ **P**.

Proof: Let H be a polynomial-time decider for B. Consider the following TM:

M = "On input w:
Compute $f(w)$.
Run H on $f(w)$.
If H accepts, accept; if H rejects, reject."

We claim that M is a polynomial-time decider for A. To see this, we prove that M is a polynomial-time decider, then that $\mathscr{L}$(M) = B. To see that M is a polynomial-time decider, note that because f is a polynomial-time reduction, computing $f(w)$ takes time $O(n^k)$ for some k. Moreover, because computing $f(w)$ takes time $O(n^k)$, we know that $|f(w)| = O(n^k)$. M then runs H on $f(w)$. Since H is a polynomial-time decider, H halts in $O(m^r)$ on an input of size m for some r. Since $|f(w)| = O(n^k)$, H halts after $O(|f(w)|^r) = O(n^{kr})$ steps. Thus M halts after $O(n^k + n^{kr})$ steps, so M is a polynomial-time decider.

To see that $\mathscr{L}(M)$ = A, note that M accepts w iff H accepts $f(w)$ iff $f(w) \in$ A. Since f is a polynomial-time reduction, $f(w) \in$ B iff w $\in$ A. Thus M accepts w iff $w \in$ A, so $\mathscr{L}(M)$ = A. ■

Next Time

- **Nondeterministic Turing Machines**
  - An absurdly powerful computing device.
- **NP**
  - What can we *verify* quickly?
- **P vs. NP**
  - How are these classes related?