P

Announcements

- **Welcome back!**
- Lecture 23 video should be posted by the end of tonight.
  - Sorry for not getting it up sooner!
- Problem session tonight in 380-380X from 7:00PM – 7:50PM.
  - Optional, but highly recommended.

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

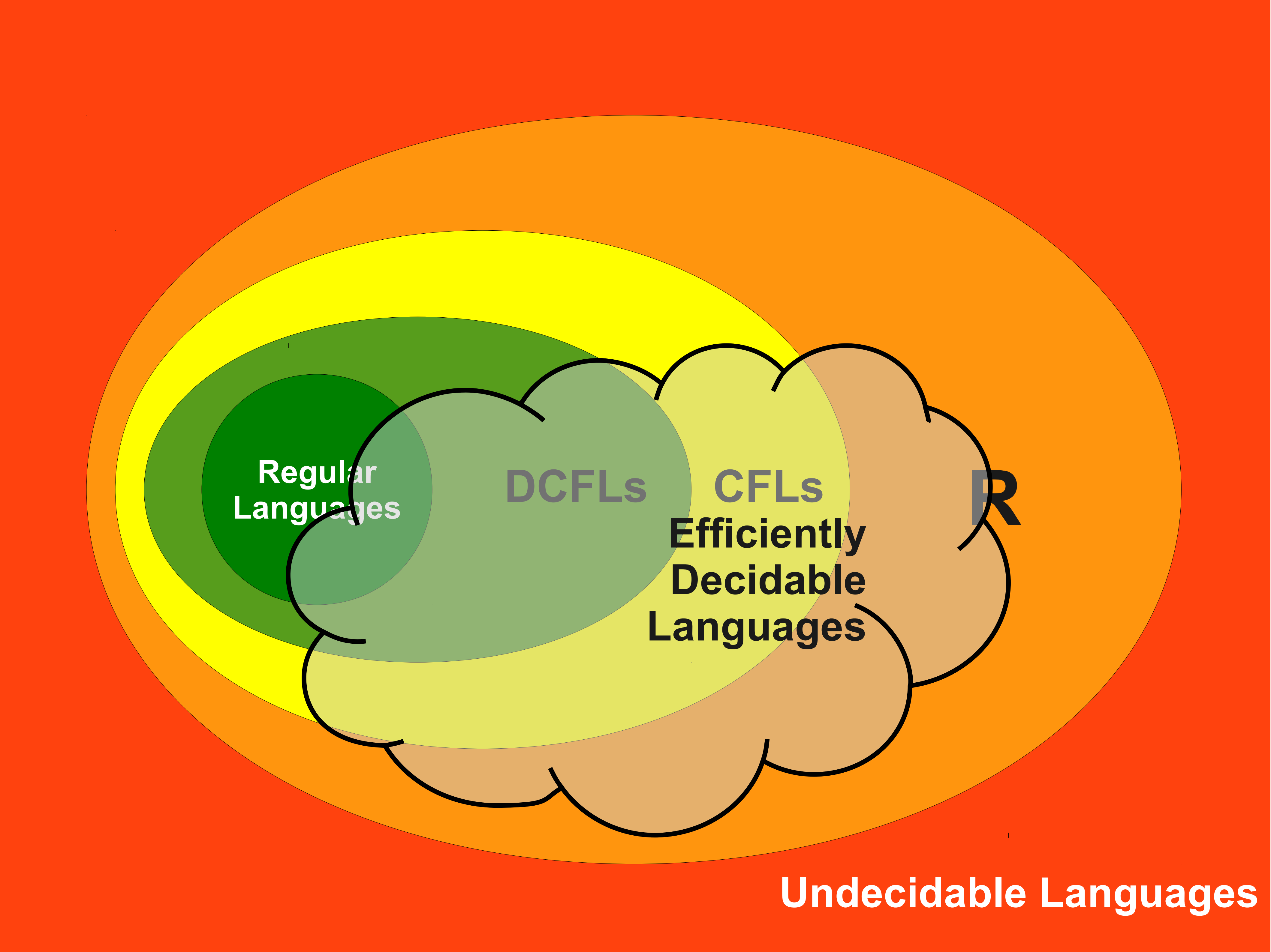
- The fact that a problem is decidable does not mean that it is *feasibly* decidable.
- In **computability theory**, we ask the question

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
DCFLs
CFLs
Efficiently Decidable Languages
R
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.
- 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

Accepting means 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	1	1	0				

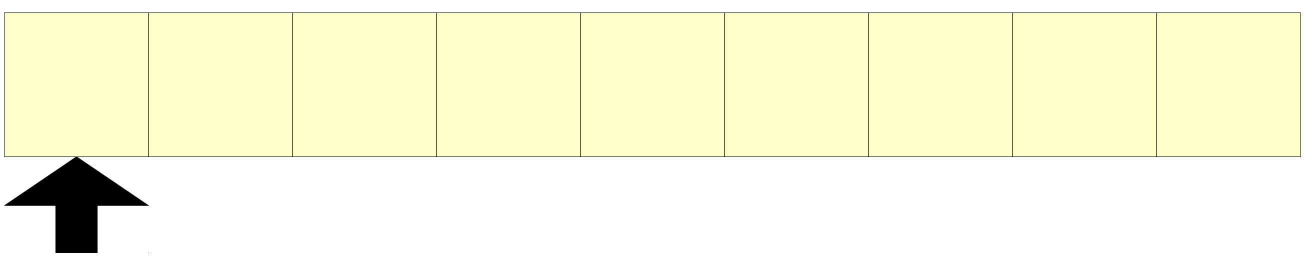
	0	1	B
q_0	0 R q_1	reject	accept
q_1	reject	1 R q_0	accept

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

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 (typically denoted $f(n)$) that measures the *worst-case* number of steps M takes on any input of length n.
  - By convention, n denotes the length of the input.
  - If M loops on some input of length k, then $f(k) = \infty$.
- The previous TM has time complexity $f(n) = n + 1$.
  - 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 $\Sigma = \{\texttt{0}, \texttt{1}\}$ for the language *BALANCE* $= \{ w \in \Sigma^* \mid w$ has the same number of `0`s and `1`s $\}$:
  - M = "On input w:
    - Scan across the tape until a `0` or `1` is found.
    - If none are found, accept.
    - If one is found, continue scanning until a matching `1` or `0` is found.
    - If none is found, reject.
    - Otherwise, cross off that symbol and repeat."
- What is the time complexity of M?

A Loss of Precision

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

The Best We Can

M = "On input w:

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

At most n steps.

At most 1 step.

At most n more steps.

At most 1 step

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

At most $n/2$ loops

+

At most $3n + 2$ steps.

× At most $n/2$ loops.

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

An Easier Approach

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

Big-O Notation

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

Big-O Notation, Formally

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

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

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

Properties of Big-O Notation

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

Life is Easier with Big-O

M = "On input w:

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

$O(n)$ steps
$O(1)$ steps
$O(n)$ steps
$O(1)$ steps
$+$ $O(n)$ steps

$O(n)$ loops

$O(n)$ steps
$\times$ $O(n)$ loops

$O(n^2)$ steps

MTTMs

- 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_2 = "On input w:
  - Scan across the tape and copy all `1`s 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 `0`s and `1`s 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.
- 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 model of computation matters!***

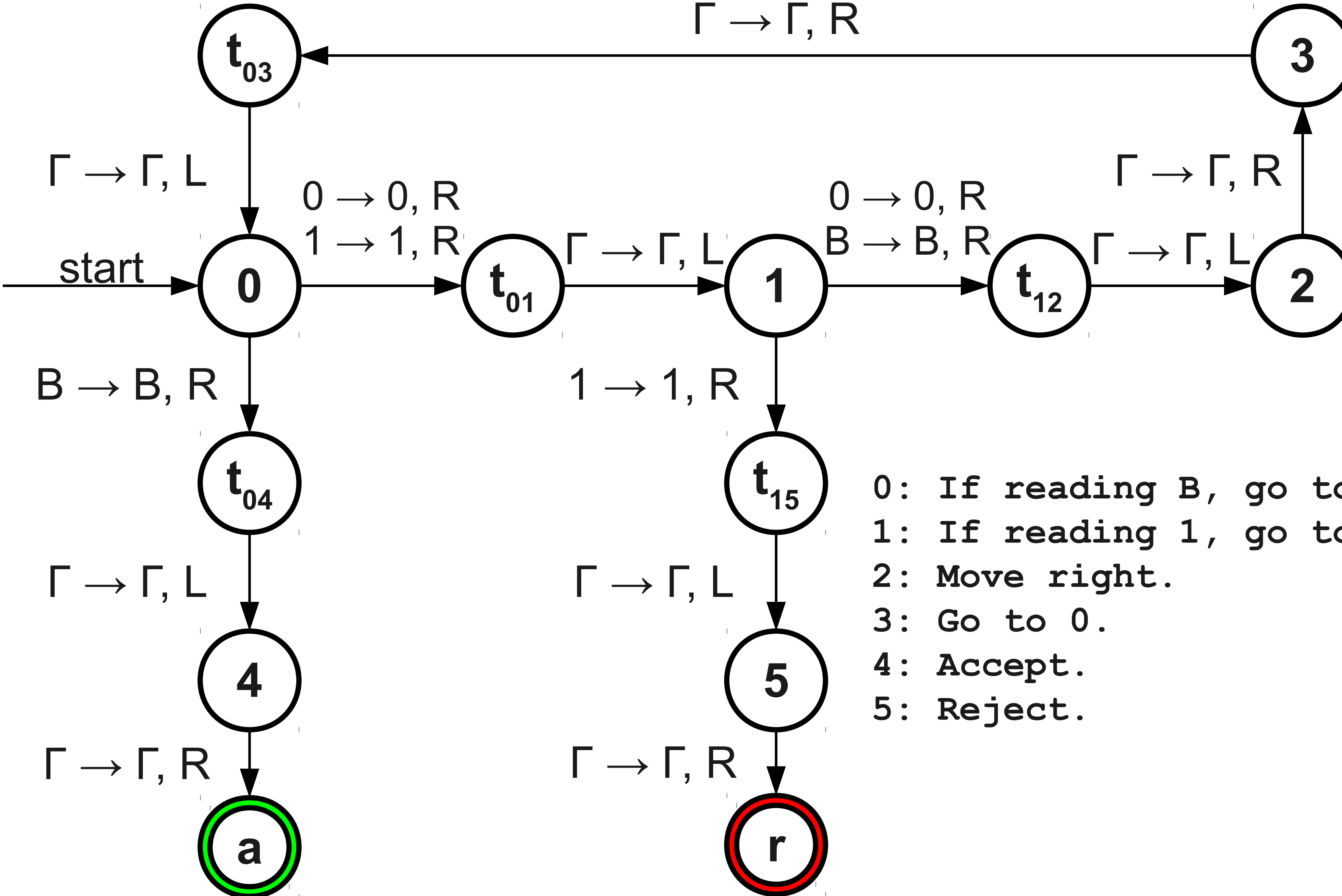
Analyzing Efficiency

- We need to reason about the efficiency of our TM equivalents.
- Questions worth considering:
  - If there is a MTTM for L that runs in time $f(n)$, can we find a TM for L that runs in time $f(n)$? $f(n)^2$? $f(n)^3$?
  - If there is a **WB** program for L that runs in time $f(n)$, can we find a TM for L that runs in time $f(n)$? $f(n)^2$? $f(n)^3$?

Our Line of Reasoning

- To analyze the relative efficiencies of MTTMs, **WB** programs, and TMs, we will do the following:
  - Show how much slowdown we get when converting a **WB** program to a TM.
  - Show how much slowdown we get when we convert a multitape **WB** program to a single-tape **WB** program.
  - Show how much slowdown we get when we convert a multitape TM to a multitape **WB** program.

From **WB** to TMs




```
0: If reading B, go to 4.
1: If reading 1, go to 5.
2: Move right.
3: Go to 0.
4: Accept.
5: Reject.
```

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 $2f(n)$.

Connecting Models of Computation

- How efficient is a multitape **WB** program compared to a single-tape **WB** program?
- **Recall:** We saw how to implement a multi*tape* **WB** program with a multi*stack* **WB** program such that each operation on the multitape **WB** program required O(1) stack operations.
- We can therefore analyze the efficiency of a multitape **WB** program by analyzing the efficiency of a multistack **WB** program.

Multitape TM Efficiency

- Time to push or pop a stack is determined by
  - how many elements are on that stack and
  - where the tape head is on the tape.
- **Important Fact #1:** After running for *n* steps, a multistack program can have at most *n* elements on any stack.
- **Important Fact #2:** After running for *n* steps, the read head of a TM can be at most *n* cells to the right of where it started.

Multitape TM Efficiency

- **Lemma**: The time required to simulate the kth step of a multitape TM is $O(k)$.
  - **Proof sketch**: We need to do at most $O(k)$ work to seek back to the start of the tape, at most $O(k)$ work to seek to the end of the stack, at most $O(1)$ work manipulating the stack, and at most $O(k)$ work moving the tape head back to where it started.
- **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 $O(f(n)^2)$. ■

What This Result Means

- We have shown that if it's possible to find an $f(n)$-time MTTM for some language L, we can also find an $O(f(n)^2)$-time single-tape TM for L.
- It might be possible to do better, though there's no guarantee.

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 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.

Why All This Matters

- **Different models of computation have different efficiencies.**
- TMs, MTTMs, **WB** programs, and computers can all solve the same problems, but may do so at different speeds.
- In many theoretical results, these differences **do not matter**.
    - We'll see why in a minute.

Time Complexity Classes

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 single-tape TM with runtime O(f(n)).
- For example:
  - TIME(n) is the set of all languages decidable in time O(n).
  - TIME(2^n) is the set of all languages decidable in time O(2^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 $n + 1$.
- Nontrivial result: A language is regular iff it is in TIME(n).
  - (This is why we can't build a single-tape TM for *BALANCE* that runs in O(n) time.)

TIME(n^2)

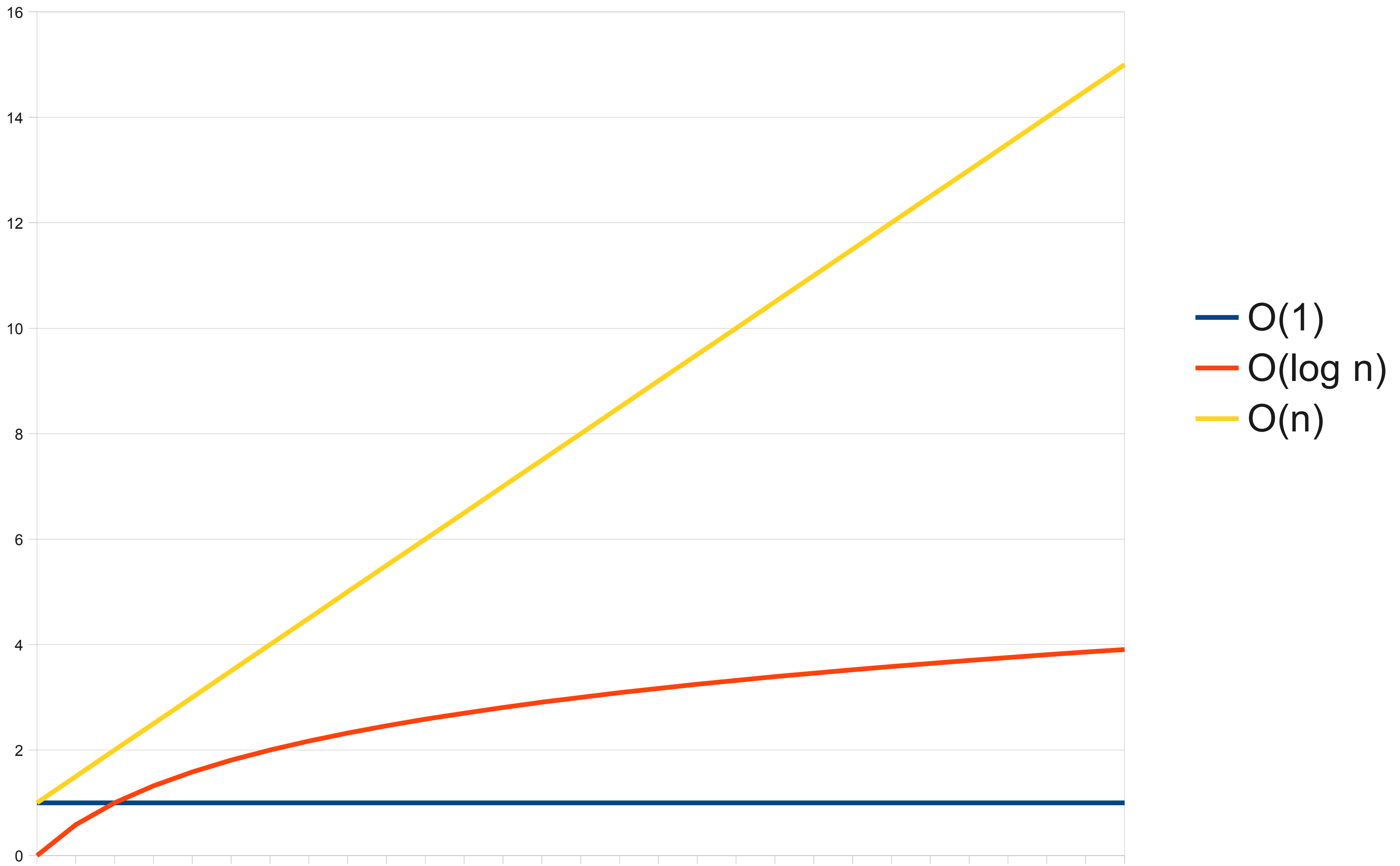
- The language of palindromes is in TIME(n^2)
  - Snake back and forth across the tape checking whether the ends match.
- The language of balanced parentheses is in TIME(n^2).
  - Use an MTTM to track unmatched open parentheses on a second tape.
- All DCFLs are in TIME(n^2).
  - Simulate a DCFL with a multitape TM in time O(n).
  - Convert to a single-tape TM in O(n^2).
- 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.

TIME(n^{18})

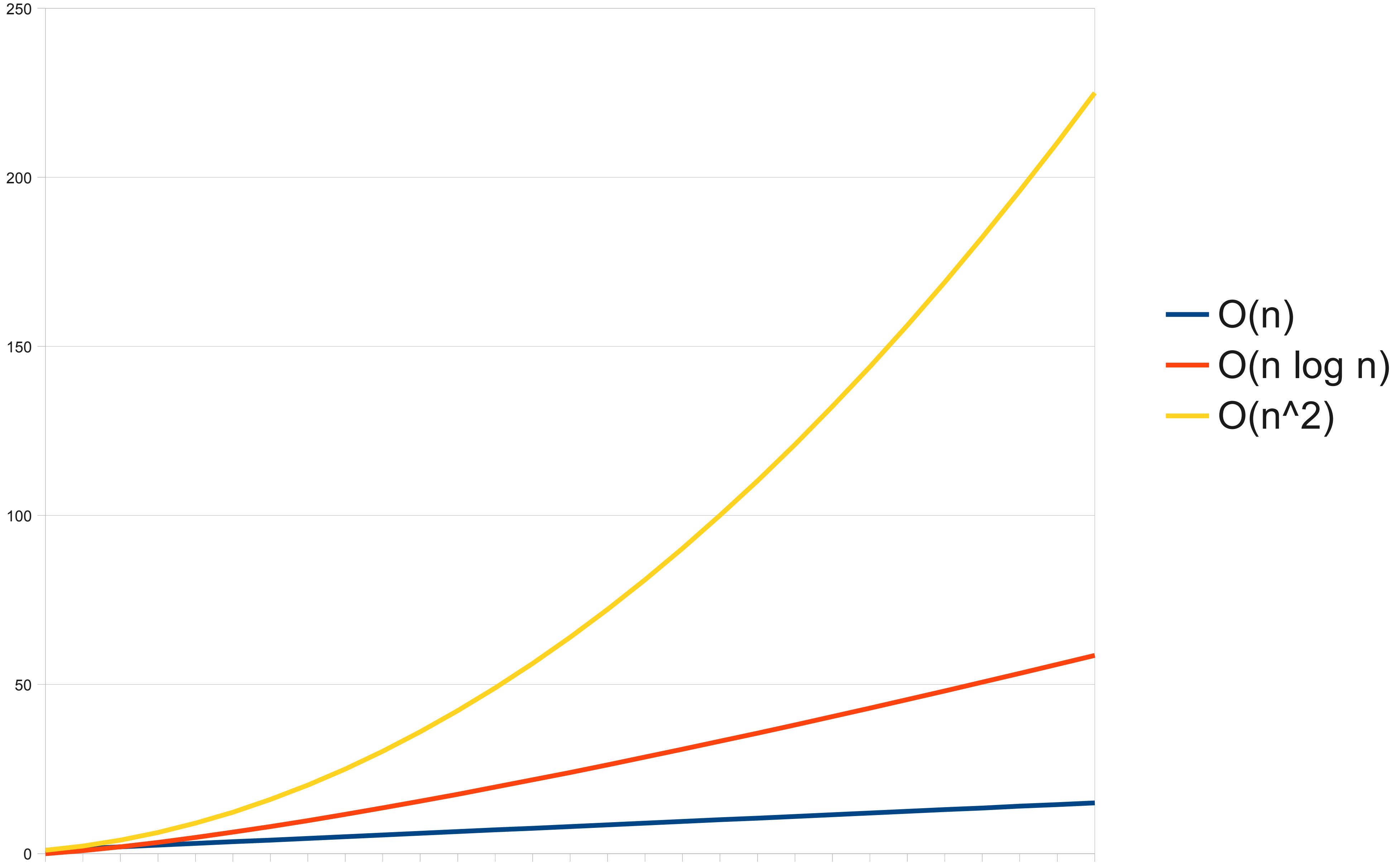
- All CFLs are in TIME(n^{18}).
  - Given a grammar G, there exists an algorithm on a standard computer that can decide whether G generates w in time O(n^3).
  - Since an $f(n)$-time computer program can be simulated in time O($f(n)^6$) on a TM, this means all CFLs are in TIME(n^{18}).

What is Efficiency?

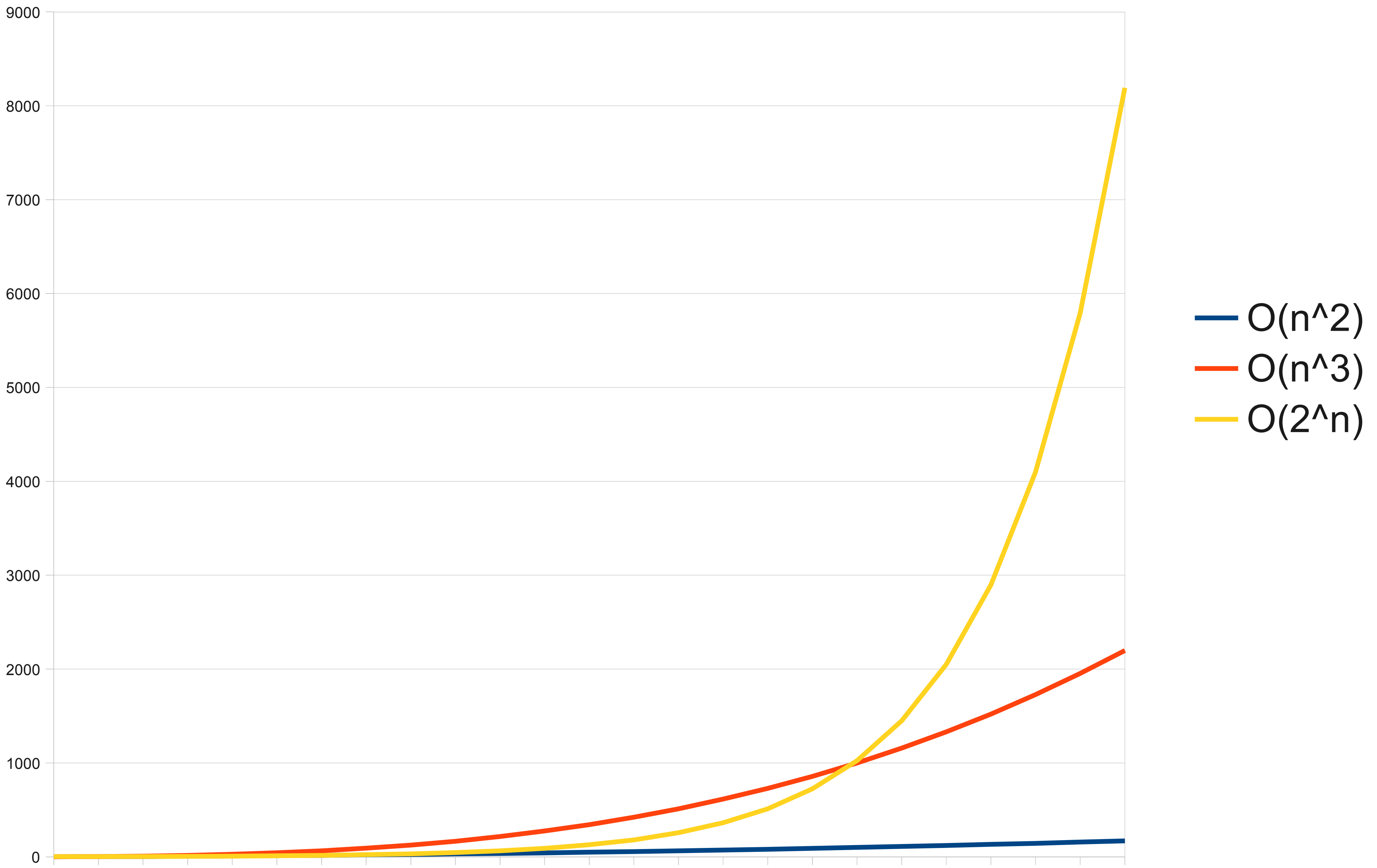
Growth Rates, Part One



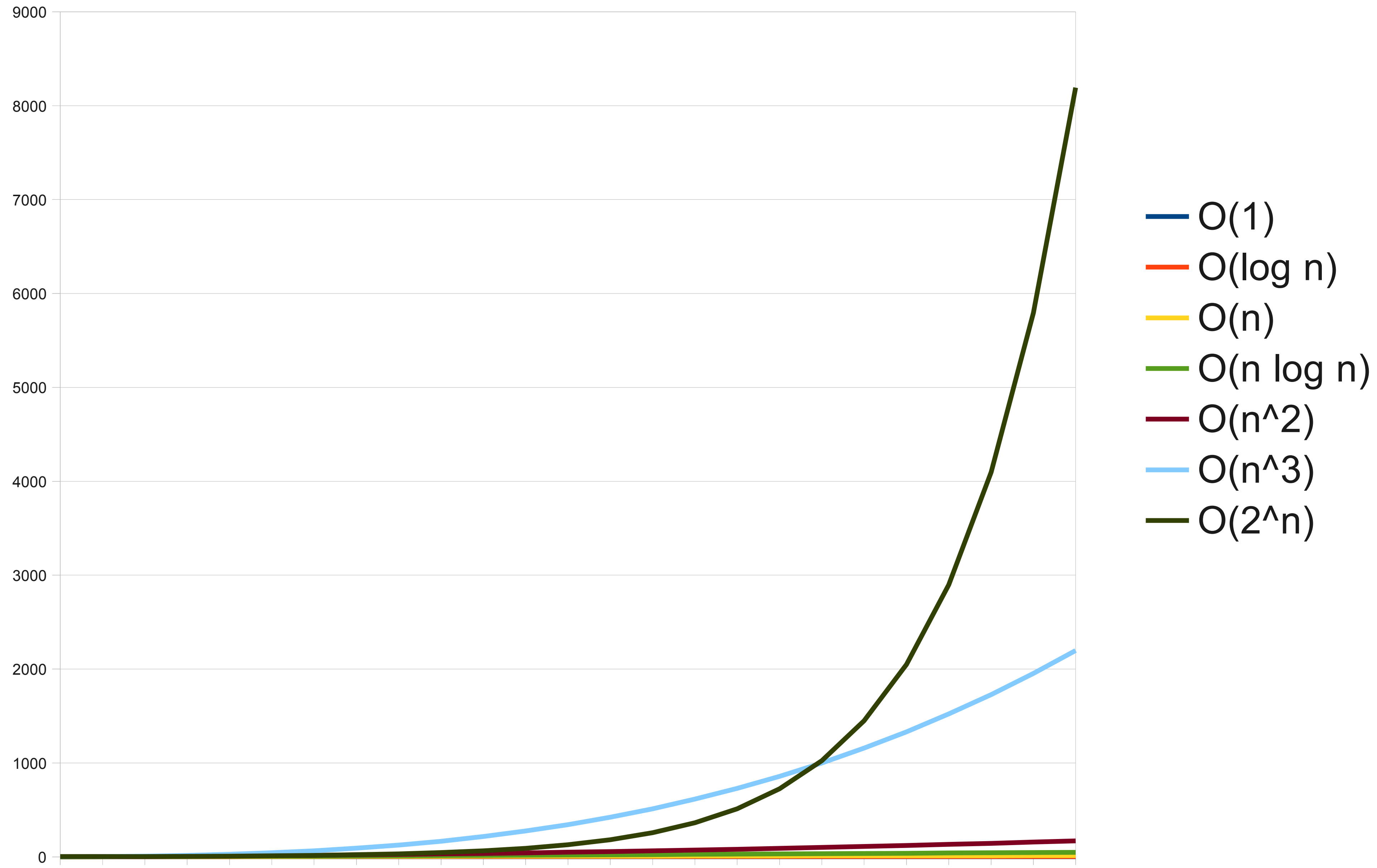
Growth Rates, Part Two



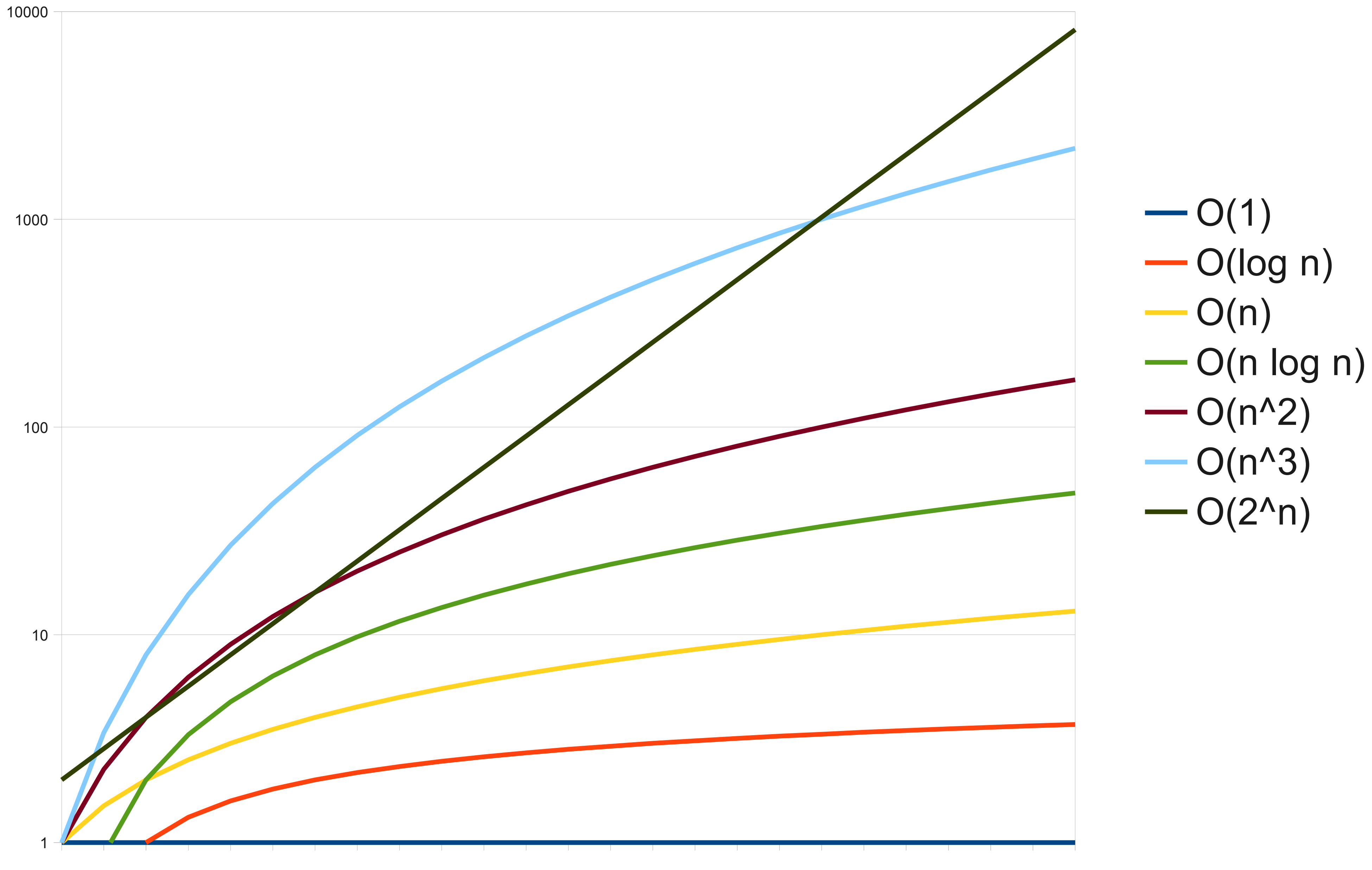
Growth Rates, Part Three



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

A language L can be **decided efficiently** iff 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

- 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**

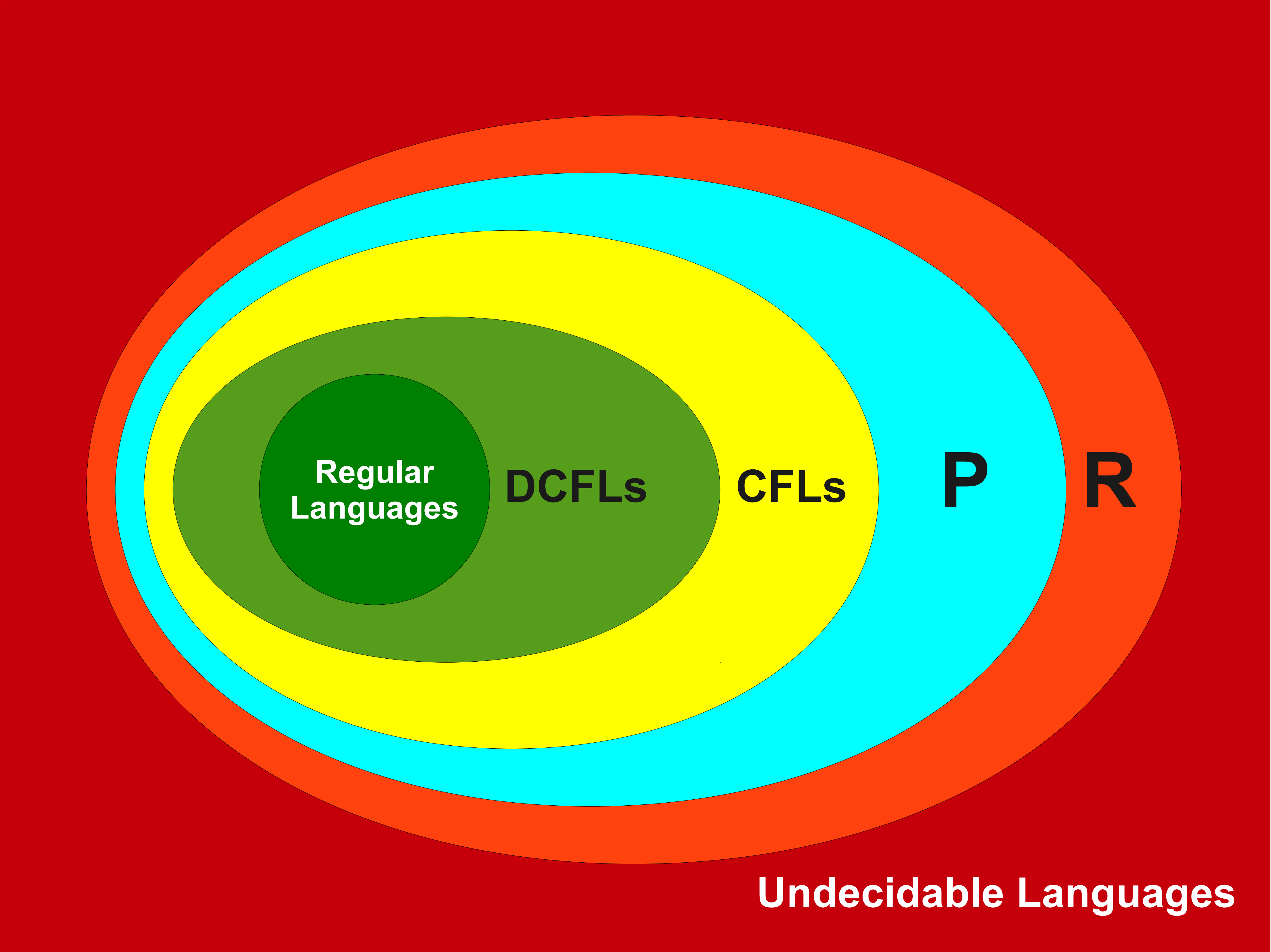
- The **complexity class P** contains all problems that can be solved in polynomial time.
- Formally:

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

- 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**.
  - Contained in TIME(n).
- All DCFLs are in **P**.
  - Contained in TIME(n^2).
- All CFLs are in **P**.
  - Contained in TIME(n^{18})
- Many other problems are in **P**.
  - *POWER2*
  - *SEARCH*


Regular
Languages
DCFLs
CFLs
P
R
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? (Best known TM for this takes time $O(n^{72})$.)

- **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?

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**: $L \in \mathbf{P}$ iff there is a polynomial-time TM, **WB*n*** program, multitape TM, or normal computer program for it.
- Essentially – a problem is in **P** iff you could solve it on a normal computer in polynomial time.

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

- **Theorem**: $O(n^k) + O(n^r) = O(n^{\max\{k, r\}})$.
  - The sum of two polynomial-bounded functions is itself a polynomial-bounded function.
- 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

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

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

Closure under Composition

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

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

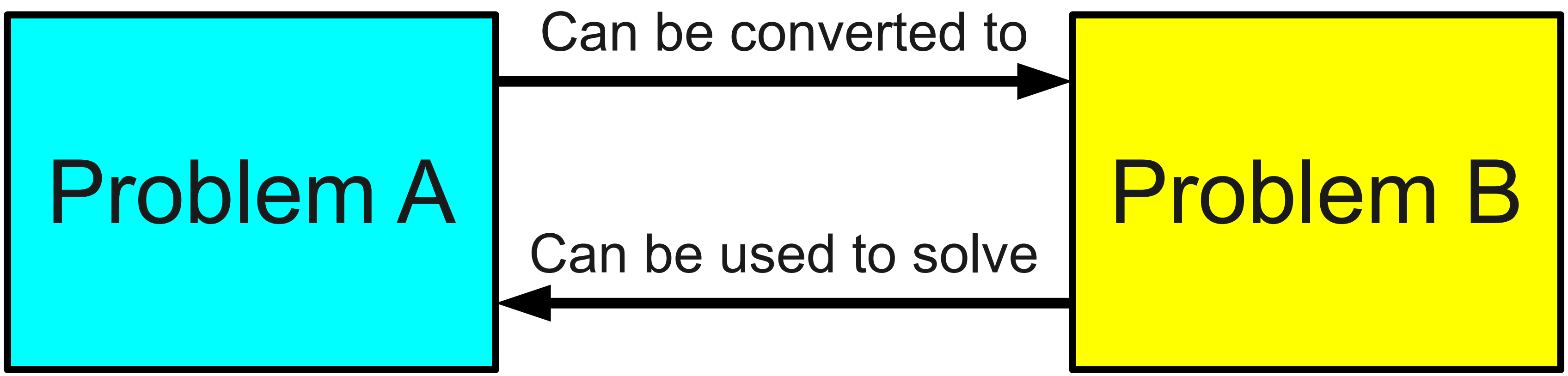
Proving Languages are in **P**

- 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.

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 Reductions

- Let $A \subseteq \Sigma_1^*$ and $B \subseteq \Sigma_2^*$ be languages.
- A **polynomial-time mapping reduction** is a function $f : \Sigma_1^* \to \Sigma_2^*$ with the following properties:
  - $f(w)$ can be computed in polynomial time.
  - $w \in A$ iff $f(w) \in B$.
- Informally:
  - A way of turning inputs to A into inputs to B
  - **that can be computed in polynomial time**
  - that preserves the correct answer.
- Notation: $\boldsymbol{A \leq_{\mathrm{P}} B}$ iff there is a polynomial-time mapping reduction from A to B.

Next Time

- **Polynomial-Time Reductions**
  - What do these reductions look like?
- **NP**
  - What can we *verify* quickly?
- **P $\stackrel{?}{=}$ NP**
  - How are these classes related?