CS311 Computational Structures

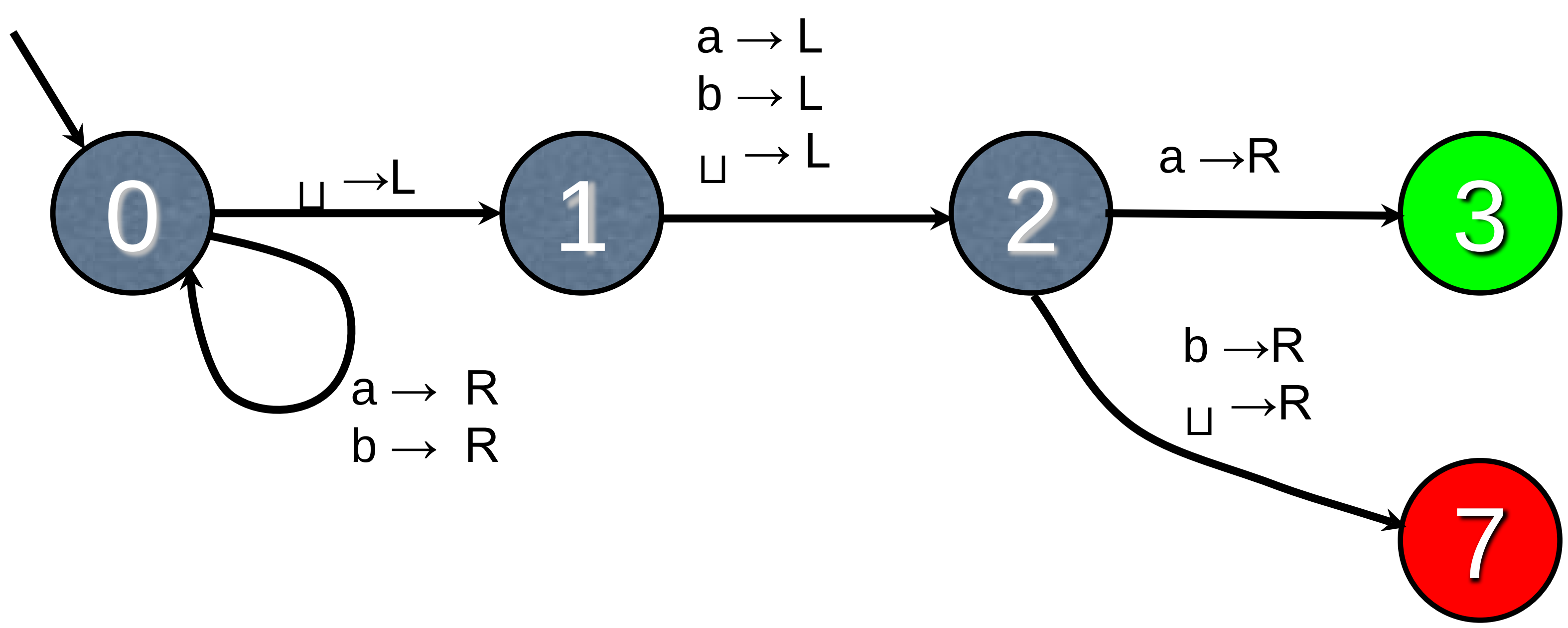
Computational Complexity

So, it’s computable!

- But at what cost?
  - Some things that are computable in principle are in practice *intractable* because of the high “cost”
  - “Cost” can measured in time, or in space, or in other resources …
- Simple time measure of decision algorithm is **number of steps taken by a (one-tape, deterministic) Turing Machine**

Counting TM Steps

- L_{ax} = {w∈{a,b}* | next to last symbol of w is a}



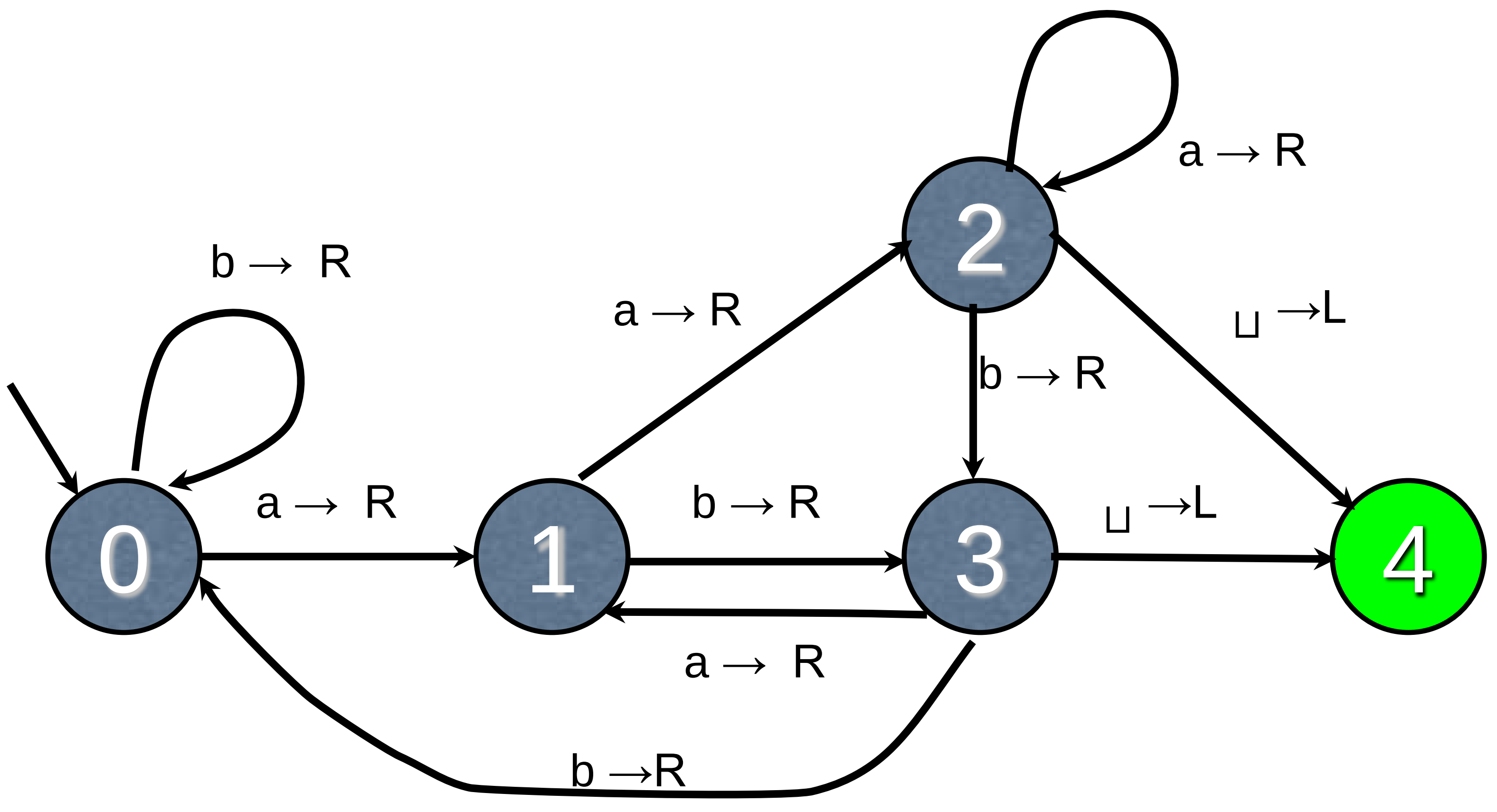
- Machine always takes exactly n+3 steps on input of length n

Time complexity class

- STEPS($t(n)$) is the class of all languages that are decidable by a (one tape deterministic) Turing machine in at most $t(n)$ steps.
  - Given input of size n, machine must halt within $t(n)$ steps with definite answer *accept* or *reject*.
- So $L_{ax} \in$ STEPS($n+3$)

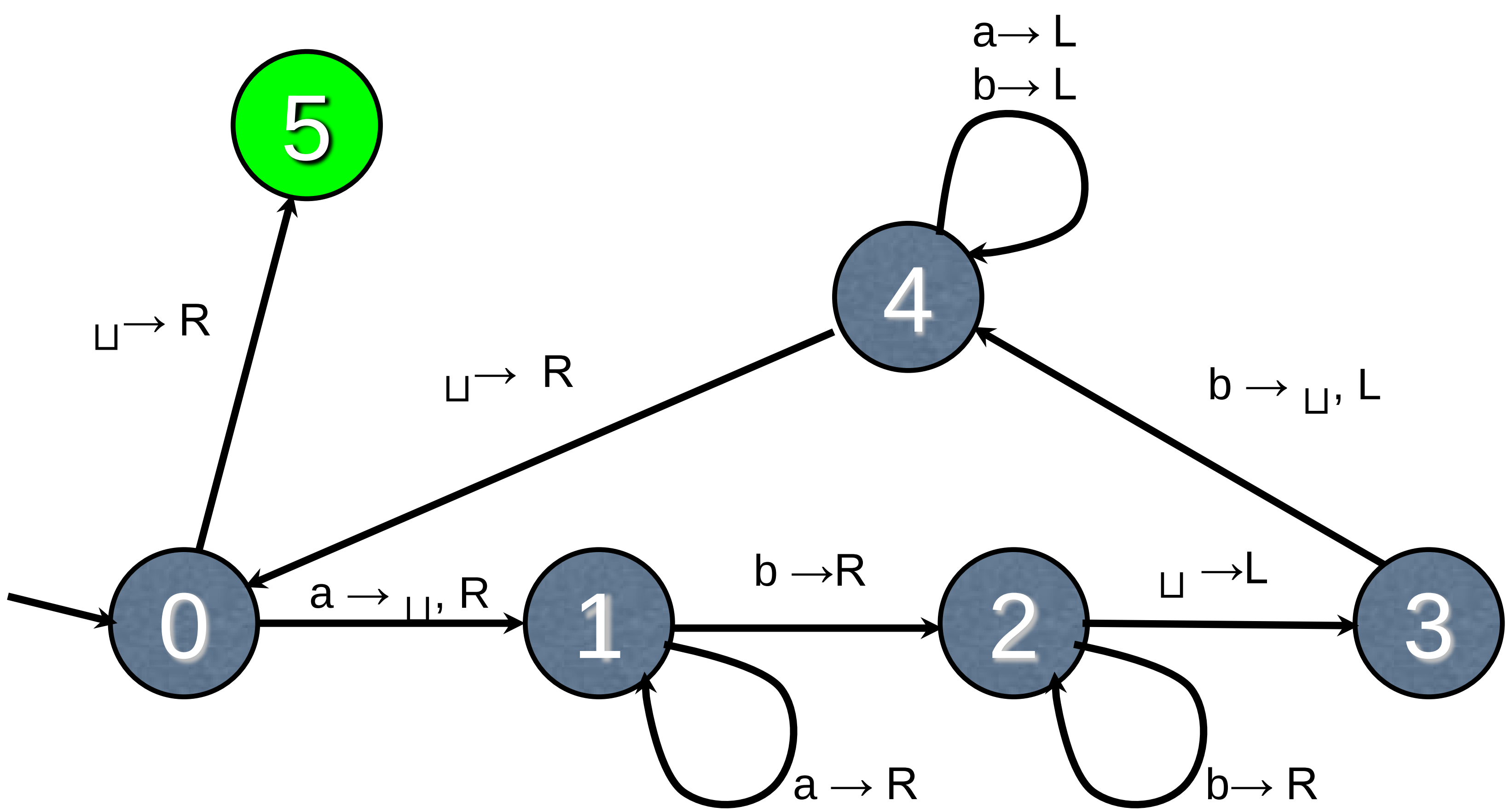
Portland State University

A smarter machine for L_{ax}



- Machine takes at most n+1 steps on input of length n
- So $L_{ax} \in$ STEPS($n+1$)

A machine for $\{a^k b^k \mid k \geq 0\}$



- Main loop matches a from start and b from end.
- e.g. 0aabb → 1abb → a1bb → ab2b → abb2 → ab3b → a4b →4ab →4⊔ab → 0ab

Calculating machine time

- Main loop on $a^k b^k$ takes 4k+1 steps and reduces k by 1
  - e.g. for k = 2: 0aabb → 1abb → a1bb → ab2b → abb2 → ab3b → a4b →4ab →4⊔ab → 0ab
- So overall time for "yes" decision on $a^k b^k$ is
  - $(4k+1) + (4(k-1)+1) + ... + (4+1) + 1$
  - $= 4(k+(k-1)+...+1) + k + 1 = 4k(k+1)/2 + k+1$
  - $= (2k+1)(k+1) = 2k^2 + 3k + 1$
- By inspection, reaching "no" can't take longer than "yes", so machine decides in at most $(1/2)n^2 + (3/2)n + 1$ steps

Problems vs. Algorithms

- So $\{a^k b^k \mid k \geq 0\} \in \mathrm{STEPS}((1/2)n^{2} + (3/2)n + 1)$.
- Does this mean that deciding $\{a^k b^k \mid k \geq 0\}$ always takes this much time?
- No! There are faster algorithms (machines) for deciding this language.
  - *e.g.*, can be done in time proportional to $n \log n$

Portland State University

Let's approximate

- We'd like to discuss time for algorithms even if they are only described informally
  - And even when we do have a precise TM, counting steps exactly is tedious!
- Also, we often care only about the asymptotic ("big O") behavior of an algorithm or problem.
- From now on, we'll be loose about describing and counting steps
  - e.g. $\{a^k b^k \mid k \geq 0\} \in \mathrm{TIME}(n^2)$

Review from Hein textbook for CS250

The Meaning of Big Oh **(5.42)**

The notation $f(n) = O(g(n))$ means that there are positive numbers c and m such that

$$|f(n)| \leq c\,|g(n)| \text{ for all } n \geq m.$$

The Meaning of Big Omega **(5.44)**

The notation $f(n) = \Omega(g(n))$ means that there are positive numbers c and m such that

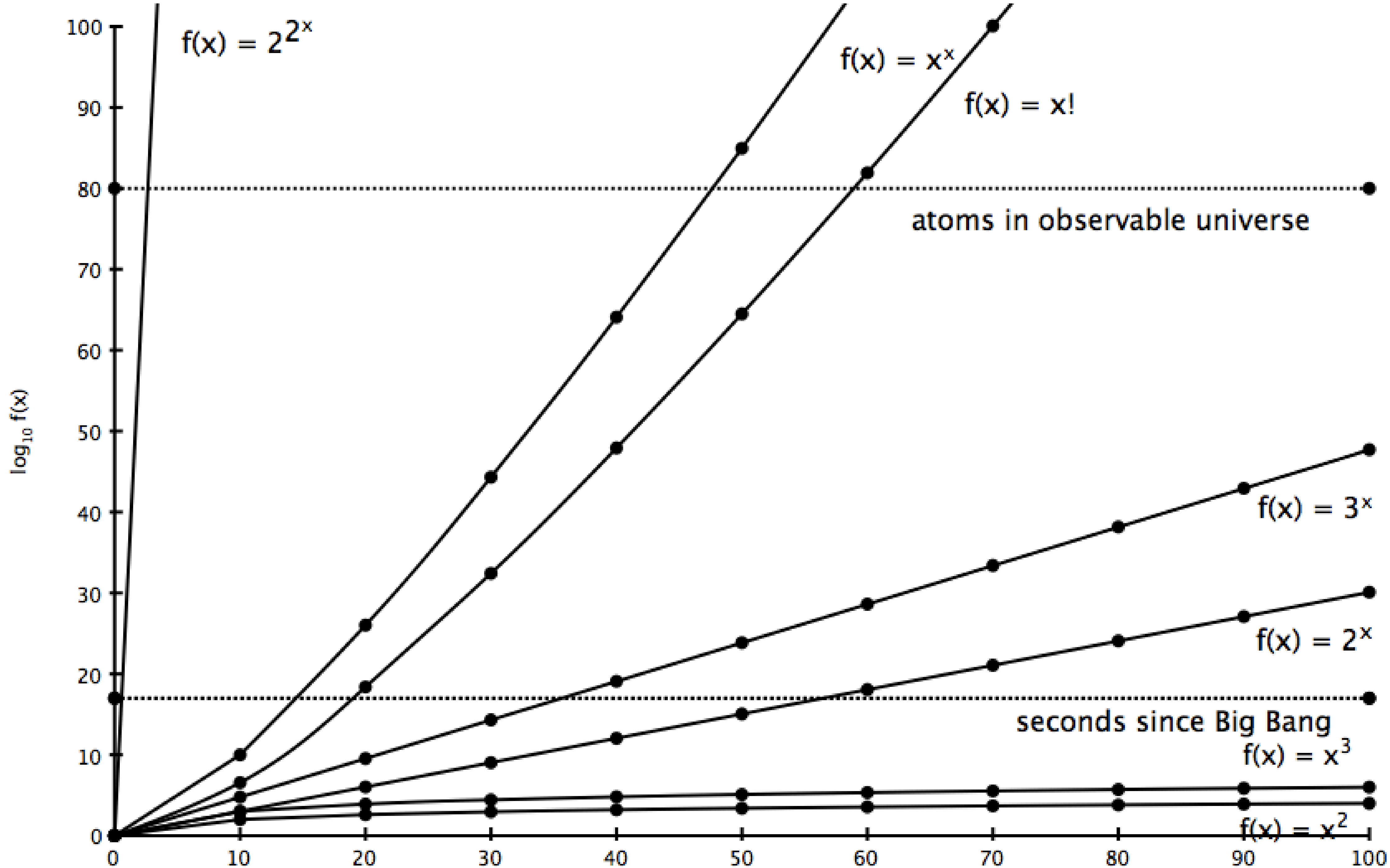
$$|f(n)| \geq c\,|g(n)| \text{ for all } n \geq m.$$

In other words...

- $O(g)$ is the set of functions whose asymptotic behavior is **bounded above** by that of g
- $\Omega(g)$ is the set of functions whose asymptotic behavior is **bounded below** by that of g
- Also, we define $\Theta(g)$ to be the set of functions with the **same** asymptotic behavior as g — *i.e.*, both $O(g)$ and $\Omega(g)$

Portland State University

Comparative growth rates of some functions



Portland State University

Avoiding Exponential Time

- Algorithms requiring exponential time are too slow to be practical except for very small input sizes.
- Indeed, problems requiring more than polynomial time are often called **intractable**.
  - Note: this is just a convenient term. A “tractable” problem that requires $O(n^{100})$ time is likely not practically solvable.

Portland State University

Another problem: PATH

- PATH = {⟨G,s,t⟩ | G is a directed graph that has a directed path from s to t }
  - We assume ⟨G⟩ is encoded as an adjacency matrix of size $O(n^2)$. (Any other **reasonable** encoding will also work.)
- There's an obvious **brute force** algorithm to decide this language: try **each** possible sequence of nodes in G of length up to n, and see if it forms a path in G
  - We consider each possible path only once
  - But there are still *n*! possible paths, so time of this algorithm is $\Omega(2^n)$.

A faster PATH finder

- PATH = {⟨G,s,t⟩ | G is a directed graph that has a directed path from s to t }
- A faster algorithm operates like this:
  - 1. Place a mark on node *s*
  - 2. Repeat until no additional nodes are marked
    - 2.1. Scan all edges of G. If an edge *(a,b)* is found from a marked node *a* to an unmarked node *b*, mark *b*.
  - 3. If *t* is marked *accept*; otherwise *reject*.
  - Time: step 2 repeats O(n) times; each step takes $O(n^2)$ time, so overall time is $O(n^3)$. PATH is tractable after all.

Finding a Hamiltonian path

- HAMPATH = {⟨G,s,t⟩ | G is a directed graph with a path from s to t passing through each node exactly once}
- The brute force algorithm for PATH works here too, so HAMPATH is in TIME(n!) (where n is number of nodes in graph).
- Nobody knows for sure whether there is a polynomial-time algorithm for HAMPATH.

What if we vary the model?

- When we're talking about timing, our precise choice of TM model matters.
  - Unlike for decidability results.
- Example: Given a two-tape TM, we can recognize $\{a^n b^n \mid n \geq 0\}$ in O(n) time. How?
- Example: On a nondeterministic TM, the brute force algorithm for PATH runs in polynomial time. Why?

Portland State University

Time cost of simulation

- We can relate execution times on fancy TM's to those on the standard TM.
- For every multitape TM that runs in time t(n), there is an equivalent single tape TM that runs in time $O(t^2(n))$.
  - IALC Thm. 8.10
- For every ND TM that decides in time t(n), there is an equivalent single tape TM that decides in time $2^{O(t(n))}$.
  - Straightforward from simulation method

Portland State UNIVERSITY

Summary

- Sometimes there's an algorithm that is asymptotically faster than the "obvious" one.
- Sometimes there isn't (and we may not know).
- The distinction between polynomial and exponential time algorithms matters
- The underlying computation model matters

The class P: Tractable Problems

- P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,
- $P = \bigcup_k \text{TIME}(n^k)$
- where the time complexity class $\text{TIME}(t(n))$ is the collection of languages that are decidable by a deterministic single-tape TM in $O(t(n))$ steps

Tractable Problems

- Most of the computer programs in common use solve a problem in P.
  - If it weren't, the program would probably run too slowly to be useful!
  - Exception: some problems can be solved for interesting special cases even if they aren't tractable in general
- But sometimes finding a polynomial time algorithm is challenging
  - And for a large class of problems, we don't know for sure whether such a algorithm exists.

Verifying vs. Solving

- Often, it seems easier to **verify** an alleged solution to a problem than it is to **determine** from scratch whether there is a solution.
  - Example: Consider the Hamiltonian Path problem; no polynomial time algorithm for this is known.
  - But if we have a proposed solution (*i.e.*, a path that visits each node once), it is simple to **verify** whether the path is correct in polynomial time. (How ?)
- The proposed solution is described by a **certificate**
  - *e.g.*, for Hamiltonian Path, the proposed path
- A **verifier** is a TM that takes a problem and a certificate and answers “OK” or “fake”

The Class NP

- The class NP contains all the decision problems that can be *verified* in Polynomial time.

Equivalently

- The class NP contains all the decision problems that can be *solved* in Polynomial time by a *nondeterministic* algorithm
  - It may make arbitrary (nondeterministic) choices
  - The number of steps must be bounded by some polynomial in *n*, where *n* is the length of the input
  - NTIME($t(n)$) = languages decidable in $O(t(n))$ time by a NDTM
  - NP = $\bigcup_k$ NTIME(n^k)

Portland State University

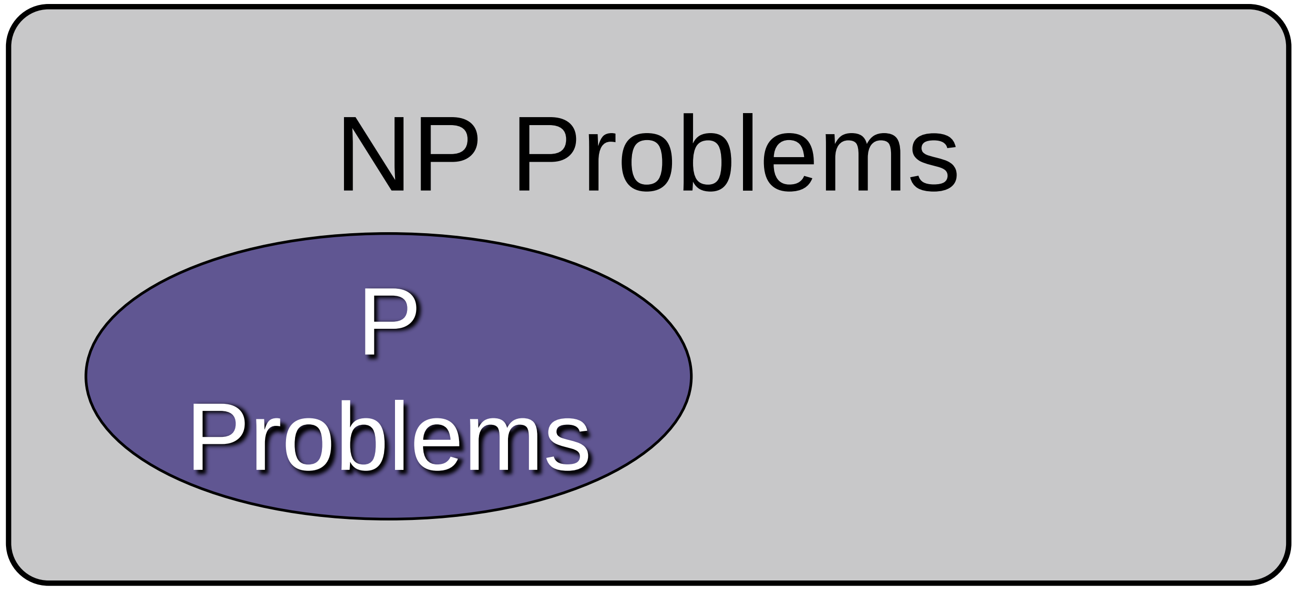
23

Equivalence of Two Definitions of NP

- Suppose that we have a deterministic verifier …
- then we build a non-deterministic solver that:
  - non-deterministically generates all putative certificates (effectively in parallel)
  - runs the deterministic verifier to check them (each in polynomial time)
  - if there is a solution, we'll find and approve its certificate

- Conversely:
  - Suppose we have a non-deterministic solver, e.g., a non-deterministic TM M
    - At each of its polynomially-many steps, it may branch at most a constant number of ways.
    - We can use the path of choices made as the certificate; valid certificates lead to accept states.
  - So: we can build a deterministic verifier that, given the certificate, simulates M on that path, and checks that it is an accept path.
    - This takes polynomially-many steps

Relationship



- Anything in P is also in NP
  - because any deterministic algorithm is also a non-deterministic algorithm
- Does P = NP ?
  - currently not known, but widely suspected P ≠ NP

Beyond NP

- Many interesting and natural problems are in NP
  - Typically show membership in NP by exhibiting a polynomial-time verifier.
- But some (natural) problems are not...
- EXPTIME = $\bigcup_k$ TIME(2^{n^k}) is believed to be larger than NP (and known to be larger than P)
- 2-EXPTIME = $\bigcup_k$ TIME$(2^{2^{n^k}})$ is larger than NP

Portland State UNIVERSITY

What about Space?

- We can measure the space use of a TM as the maximum number of tape cells it scans on an input of length n.
- Define SPACE(f(n)) as the class of languages decided by a (deterministic) TM using O(f(n)) space.
- Define NSPACE similarly for non-deterministic TM.

NP ⊆ PSPACE

- By analogy with time classes, we define PSPACE = $\bigcup_k$ SPACE(n^k) and NPSPACE = $\bigcup_k$ NSPACE(n^k)
- Then NP ⊆ PSPACE.
  - Clearly NP ⊆ NPSPACE, because a machine that takes t steps can access at most t tape squares.
  - It turns out that NPSPACE = PSPACE
    - Consequence of Savitch's theorem (IALC 11.5)
  - It also turns out that NPSPACE ⊆ EXPTIME
- But it is unknown whether NP ⊊ PSPACE or NSPACE ⊊ EXPTIME