Intractable Problems

Time-Bounded Turing Machines
Classes **P** and **NP**
Polynomial-Time Reductions

Time-Bounded TM's

◆ A Turing machine that, given an input of length n, always halts within T(n) moves is said to be *T(n)-time bounded*.

▶ The TM can be multitape.

▶ Sometimes, it can be nondeterministic.

The class **P**

- ◆ If a DTM M is T(n)-time bounded for some polynomial T(n), then we say M is *polynomial-time* ("*polytime*") bounded.
- ◆ And L(M) is said to be in the class **P**.
- ◆ Important point: when we talk of **P**, it doesn't matter whether we mean "by a computer" or "by a TM" (next slide).

Polynomial Equivalence of Computers and TM's

◆ A multitape TM can simulate a computer that runs for time $O(T(n))$ in at most $O(T^2(n))$ of its own steps.

◆ If $T(n)$ is a polynomial, so is $T^2(n)$.

Examples of Problems in **P**

◆ Is w in L(G), for a given CFG G?

   ◗ Input = w.

   ◗ Use CYK algorithm, which is $O(n^3)$.

◆ Is there a path from node x to node y in graph G?

   ◗ Input = x, y, and G.

   ◗ Use depth-first search, which is $O(n)$ on a graph of n nodes and arcs.

Running Times Between Polynomials

◆ You might worry that something like O(n log n) is not a polynomial.

◆ However, to be in **P**, a problem only needs an algorithm that runs in time less than some polynomial.

◆ Surely O(n log n) is less than the polynomial O(n^2).

A Tricky Case: Knapsack

◆ The *Knapsack Problem* is: given positive integers i_1, i_2 ,..., i_n, can we divide them into two sets with equal sums?

◆ Perhaps we can solve this problem in polytime by a dynamic-programming algorithm:

  ◗ Maintain a table of all the differences we can achieve by partitioning the first j integers.

Knapsack – (2)

◆Basis: $j = 0$. Initially, the table has "true" for 0 and "false" for all other differences.

◆Induction: To consider i_j, start with a new table, initially all false.

◆Then, if the entry for m is "true" in the old table set the entries for $m+i_j$ and $m-i_j$ to "true" in the new table.

Knapsack – (3)

◆Suppose we measure running time in terms of the sum of the integers, say s.

◆Each table needs only space $O(s)$ to represent all the positive and negative differences we could achieve.

◆Each table can be constructed in time $O(s)$.

Knapsack – (4)

◆ Since n $\leq$ s, we can build the final table in $O(s^2)$ time.

◆ From that table, we can see if 0 is achievable and solve the problem.

Subtlety: Measuring Input Size

◆ "Input size" has a specific meaning: the length of the representation of the problem instance as it is input to a TM.

◆ For the Knapsack Problem, you cannot always write the input in a number of characters that is polynomial in the sum of the integers.

Knapsack – Bad Case

◆ Suppose we have n integers, each of which is around 2^n.

◆ We can write integers in binary, so the input takes $O(n^2)$ space to write down.

◆ But the tables require space $O(n2^n)$.

◆ All n tables in time $O(n^2 2^n)$.

 ◗ Or, since we like to use n as the input size, input of length n requires $O(n2^{sqrt(n)})$ time.

Redefining Knapsack

◆ We are free to describe another problem, call it *Pseudo-Knapsack*, where integers are represented in unary.

◆ Pseudo-Knapsack is in **P**.

The Class **NP**

- ◆ The running time of a nondeterministic TM is the maximum number of steps taken along any branch.

- ◆ If that time bound is polynomial, the NTM is said to be *polynomial-time bounded*.

- ◆ And its language/problem is said to be in the class **NP**.

Example: **NP**

◆The Knapsack Problem is definitely in **NP**, even using the conventional binary representation of integers.

◆Use nondeterminism to guess a partition of the input into two subsets.

◆Sum the two subsets and compare.

P Versus NP

◆ Originally a curiosity of Computer Science, mathematicians now recognize as one of the most important open problems the question **P** = **NP**?

◆ There are thousands of problems that are in **NP** but appear not to be in **P**.

◆ But no proof that they aren't really in **P**.

Complete Problems

◆ One way to address the **P** = **NP** question is to identify *complete problems* for NP.

◆ An *NP-complete problem* has the property that it is in **NP**, and if it is in **P**, then every problem in **NP** is also in **P**.

◆ Defined formally via "polytime reductions."

Complete Problems – Intuition

◆ A complete problem for a class embodies every problem in the class, even if it does not appear so.

◆ Compare: PCP embodies every TM computation, even though it does not appear to do so.

◆ Strange but true: Knapsack embodies every polytime NTM computation.

Polytime Reductions

◆Goal: find a way to show problem L to be NP-complete by reducing every language/problem in **NP** to L in such a way that if we had a deterministic polytime algorithm for L, then we could construct a deterministic polytime algorithm for any problem in **NP**.

Polytime Reductions – (2)

◆ We need the notion of a *polytime transducer* – a TM that:

1. Takes an input of length n.
2. Operates deterministically for some polynomial time p(n).
3. Produces an output on a separate *output tape*.

◆ Note: output length is at most p(n).

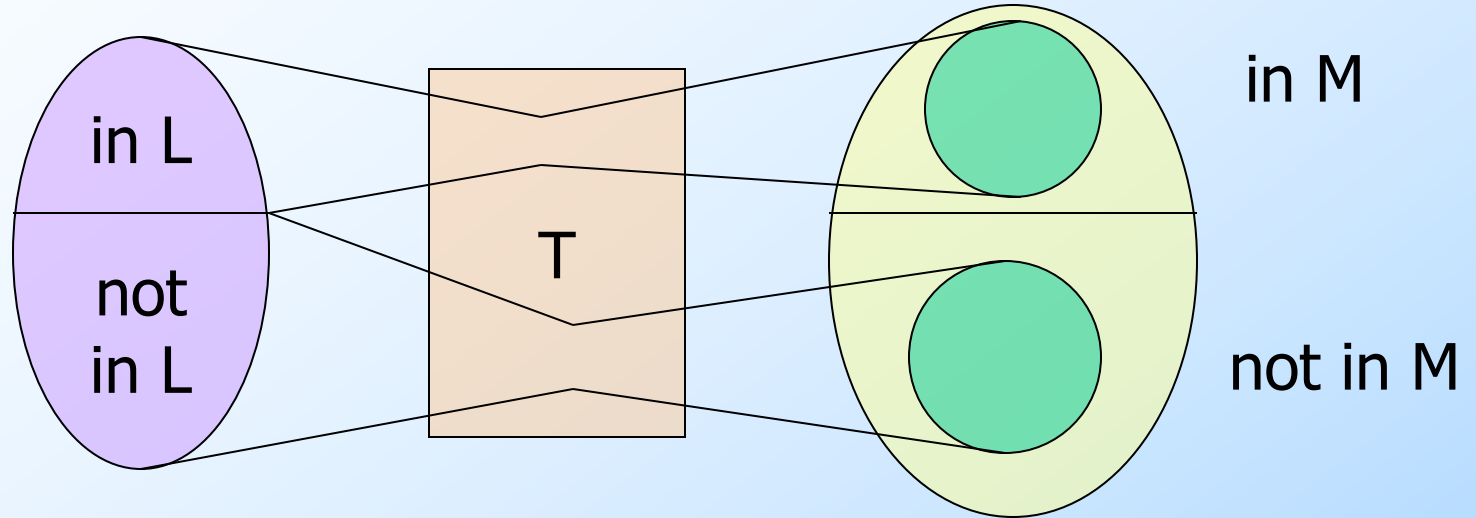# Polytime Transducer

state

input $\longleftrightarrow$ n

scratch
tapes
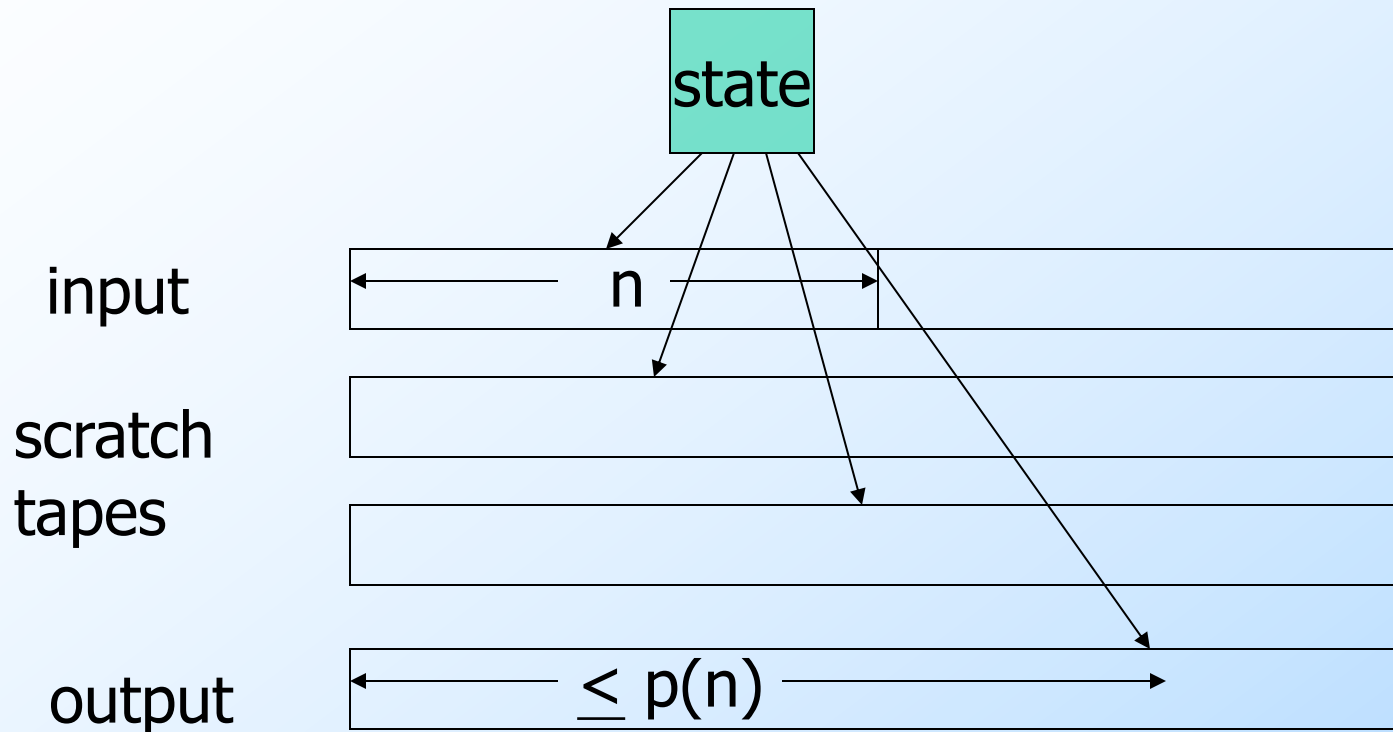
output $\longleftrightarrow \underline{\le} p(n)$

Polytime Reductions – (3)

◆ Let L and M be langauges.

◆ Say L is *polytime reducible* to M if there is a polytime transducer T such that for every input w to T, the output x = T(w) is in M if and only if w is in L.

Picture of Polytime Reduction

in L

not
in L

T

in M

not in M

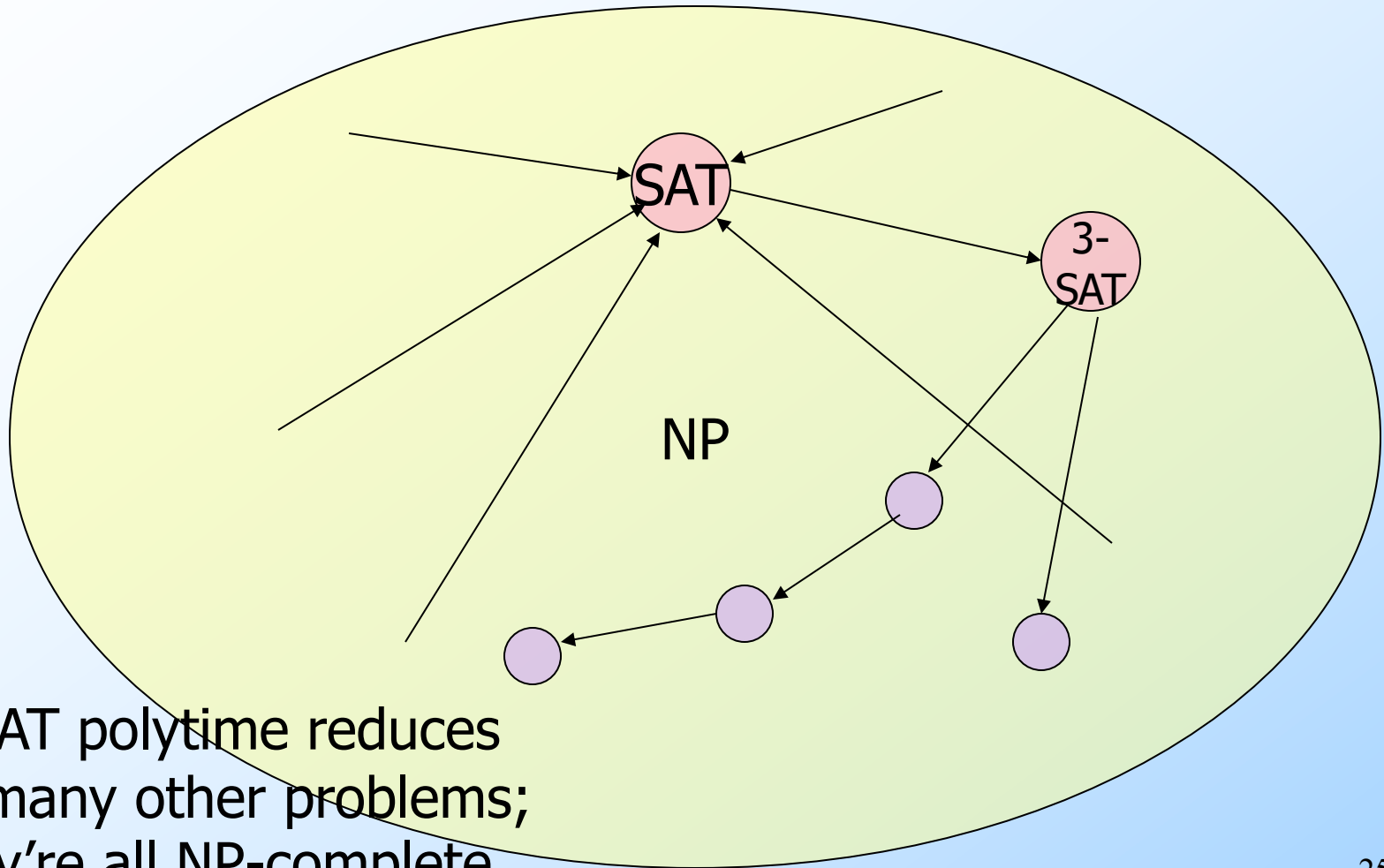# NP-Complete Problems

◆A problem/language M is said to be *NP-complete* if it is in **NP**, and for every language L in **NP**, there is a polytime reduction from L to M.

◆Fundamental property: if M has a polytime algorithm, then so does L.

◗ I.e., if M is in **P**, then every L in **NP** is also in **P**, or "**P** = **NP**."

The Plan

All of **NP** polytime reduces to SAT, which is therefore NP-complete

SAT polytime reduces to 3-SAT

NP

3-SAT polytime reduces to many other problems; they're all NP-complete

25

Proof That Polytime Reductions "Work"

◆ Suppose M has an algorithm of polynomial time q(n).

◆ Let L have a polytime transducer T to M, taking polynomial time p(n).

◆ The output of T, given an input of length n, is at most of length p(n).

◆ The algorithm for M on the output of T takes time at most q(p(n)).

Proof – (2)

◆ We now have a polytime algorithm for L:

1. Given w of length n, use T to produce x of length $\leq p(n)$, taking time $\leq p(n)$.

2. Use the algorithm for M to tell if x is in M in time $\leq q(p(n))$.

3. Answer for w is whatever the answer for x is.

◆ Total time $\leq p(n) + q(p(n))$ = a polynomial.