NP-Completeness

Goal: We want some way to classify problems that are hard to solve, i.e. problems for which we can not find polynomial time algorithms.

For many interesting problems

- we cannot find a polynomial time algorithm
- we cannot prove that no polynomial time algorithm exists
- the best we can do is formalize a class of NP-complete problems that either all have polynomial time algorithms or none have polynomial time algorithms

NP-completeness arises in many fields including

- biology
- chemistry
- economics
- physics
- engineering
- sports
- etc.

Goal in class:

To learn how to prove that problems are NP-complete.

We need a formalism for proving problems hard.

Turing Machine (simplified description)

A Turing Machine has

- **Finite state control**
- **Infinite tape (each square can hold 0, 1, $, or be blank.**
- **Read-Write head**

Each step of the finite state control is a function

$$f(\textbf{current state}, \textbf{tape symbol}) \rightarrow (\textbf{new state}, \textbf{symbol to write}, \textbf{movement of head})$$

Example

Program to test if a binary number is even. Input is \$ terminated. Output is written immediately after \$, 1 for yes, 0 for no.

- Read until \$ (state q_0)
- Back up, check last digit (state q_1)
- if even, write a 1 (states q_2, q_3, q_F)
- if odd, write a 0 (states q_4, q_5, q_F)

Here is a program. Each cell is (new state, write symbol move)

state	input 0	input 1	input \$
(q_0)	$(q_0, -, R)$	$(q_0, -, R)$	$(q_1, -, L)$
(q_1)	$(q_2, -, R)$	$(q_4, -, R)$	error
(q_2)	error	error	$(q_3, -, R)$
(q_3)	$(q_F, 1, -)$	$(q_F, 1, -)$	$(q_F, 1, -)$
(q_4)	error	error	$(q_5, -, R)$
(q_5)	$(q_F, 0, -)$	$(q_{\cancel{0}F}, \cancel{1}0, -)$	$(q_{\cancel{0}F}, \cancel{1}0, -)$
(q_F)	halt	halt	halt

Church Turing Thesis The set of things that can be computed on a TM is the same as the set of things that can be computed on any digital computer.

P

Definition Let P be defined as the set of problems that can be solved in polynomial time on a TM (On an input of size n, they can be solved in time $O(n^k)$ for some constant k)

Theorem P is the set of problems that can be solved in polynomial time on the model of computation used in CSOR 4231 and on every modern non-quantum digital computer.

Technicalities

- We assume a reasonable (binary) encoding of input
- Note that all computers are related by a polynomial time transformation. Think of this as a "compiler"

Further details

- We restrict attention to "yes-no" questions
- Shortest path is now "Given a graph G and a number b does the shortest path from s to t have length at most b.
- We do not use the language framework from the book in class

Verification

Verification Given a problem X and a possible solution S, is S a solution to X.

Example X is shortest paths and S is an s-t path in S that is claimed to have length at most b, check whether the path really is of length at most b

Example X is sorting and S is an allegedly sorted list. Is the list really sorted?

Claim Verification is no harder than solving a problem from scratch.
We write

$$\text{Verification} \leq \text{Solving}$$

Def: NP is the set of problems that can be verified in polynomial time

Formally: Problem X with input of size n is in NP if there exists a "certificate" y, $|y| = \text{poly}(n)$ such that, using y, one can verify whether a solution x is really a solution in polynomial time. (Think of y as the "answer")

Some problems

Longest Path Given a graph G, and number k is the longest simple path from s to t of length $\geq k$.

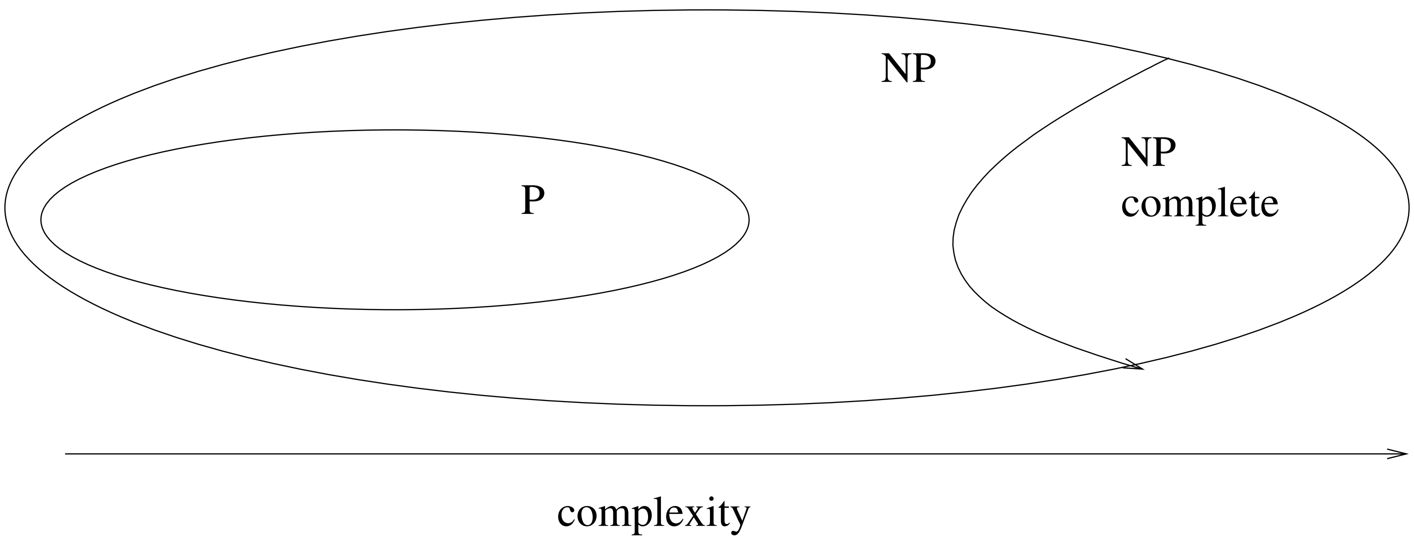
Satisfiability Given a formula Φ in CNF (conjunctive normal form), does there exist a satisfying assignment to Φ, i.e. an assignment of the variables that evaluates to true.

Big Question

$$P = NP??$$

Is solving a problem no harder than verifying?

Don't know answer. Instead we will identify "hardest" problems in NP. If any of these are in P then all of NP is in P.



NP-complete

Definition **Problem X is NP-complete if**

1. $X \in NP$

2. $Y \leq X \ \ \forall Y \in NP$

Definition $Y \leq X$ **means**

- **Y is polynomial time reducible to X, which means**

there exists a polynomial time computable function f **that maps inputs to** Y **to inputs to** X**, such that**

input y **to problem** Y **returns "Yes" iff input** $f(y)$ **to problem** X **returns "Yes"**

Informally $Y \leq X$ **means that Y is "not much harder than" ("easier than") X**

Theorem

If $Y \leq X$ **then** $X \in P \Rightarrow Y \in P$

Contrapositive

If $Y \leq X$ **then** $Y \notin P \Rightarrow X \notin P$

SAT

Theorem SAT is NP-complete

Proof idea: The turing machine program for any problem in NP can be verified by a polynomial sized SAT instance that encodes that the input is well formed and that each step follows legally from the next.

Implication We now have one NP-complete problem. We will now reduce other problems to it.

Reductions

- **If I want to show that X is easy, I show that in polynomial time I can reduce X to Y, where I already know that Y is easy.**
- **If I want to show that X is hard, then I reduce Y to X, where I already know that Y is hard.**
- **So if SAT $\leq X$, then X is hard.**

Showing X is NP-complete

To show that X is NP-complete, I show:

1. $X \in NP$

2. For some problem Z that I know to be NP-complete $Z \leq X$

Showing X is NP-complete

To show that X is NP-complete, I show:

1. $X \in NP$

2. For some problem Z that I know to be NP-complete $Z \leq X$

Expanded version: **To show that X is NP-complete, I show:**

1. $X \in NP$

2. Find a known NP-complete problem Z.

3. Describe f, which maps input z to Z to input $f(z)$ to X.

4. Show that Z with input z returns "yes" iff X with input $f(z) returns “yes'$

5. Show that f runs in polynomial time.