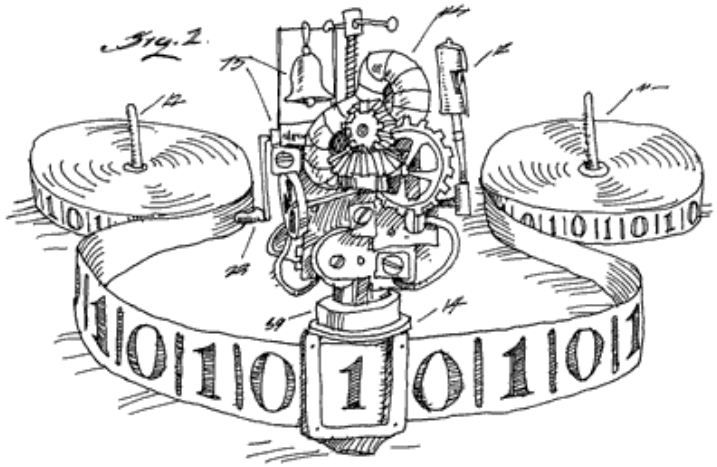
EECS 376: Foundations of Computer Science

Lecture 14 - Introduction to Complexity



More practical classification of problems

- Previously, we classified problems like this:
  - **Decidable:** solvable in finite time
  - **Undecidable:** not solvable in finite time

- "For practical purposes,
  the difference between polynomial and exponential
  is often more crucial than
  the difference between finite and non-finite."

- Jack Edmonds

(defined complexity class P, 1965)

- **Today and next class:** another classification
  - **P:** solvable in polynomial time
  - **NP-hard:** not likely solvable in polynomial time

Plan for this part of the course

Lecture 1:

- Define **P** and **NP**

Lecture 2

- Define **NP-hard** and **NP-complete.**
- Show the first NP-complete problem: SAT

Lectures 3 - 4

- Show many NP-complete problems via reductions

Lectures 5 - 6

- Show many methods to solve efficiently NP-hard Problems

Class **P:**
problems we can solve "fast"

Exponential vs. Polynomial

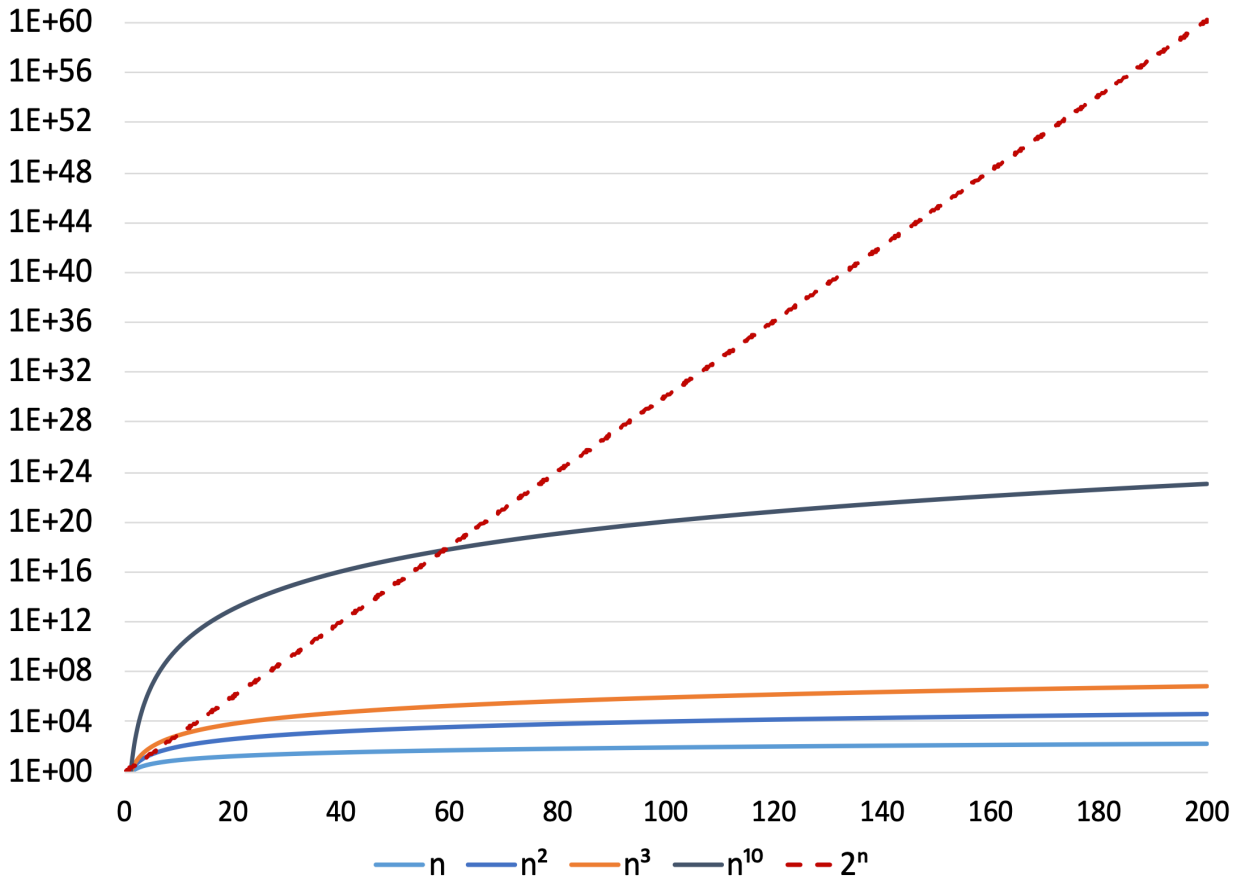
A regular Mac Pro computer performs about 10^{12} operations/sec

	n=10	**n=35**	**n=60**	**n=85**
$\mathbf{n^2}$	100 < 1 sec	1225 < 1 sec	3600 < 1 sec	7225 < 1 sec
$\mathbf{n^3}$	1000 < 1 sec	43k < 1 sec	216k < 1 sec	614k < 1 sec
$\mathbf{2^n}$	1024 < 1 sec	34×10^9 < 1 sec	> 4 years	> 120 million years

"Efficient": running time polynomial in input size

Exponential vs. Polynomial

- Consider this plot with an **adjusted scaled**



The Complexity Class **P**

Definition:

- **P** is the set of all decision problems that can be decided in polynomial time.

Formally:

- For any problem **L**, an efficient decider **Decide-L** for **L** is such that
  - **x** is a "yes" instance ⇔ **Decide-L**(**x**) accepts
  - **x** is a "no" instance ⇔ **Decide-L**(**x**) rejects (follows from above)
  - **Decide-L**(**x**) runs in poly(|**x**|) time
- **P** is the set of all decision problems that have efficient deciders

What we already know

Questions:

- Why polynomial? Why not $O(n^3)$? Why not $O(n)$?
- Why decision problems only?

Why do we use **Polynomial time** to capture the notion of efficiency?

- It is a robust definition.

- **Composable:**
  - If my program calls a polynomial number of polynomial-time algorithms, then my program runs in polynomial time
    - *Proof idea*: $(n^k)^{k'} = n^{k \cdot k'}$ is also polynomial.

- **Model-independent:** by Church-Turing thesis
  - A problem is solvable in polynomial time on a TM if and only if
  - it is solvable in polynomial time any computer.

Why do we restrict ourselves to only **Decision problems**?

- Decision problems are simpler
  - They also fit with the **language** formulation discussed in previous lectures

- To show that some problems are solvable in polytime,
  - Usually, via **binary search**,
    it is enough to check if the decision version is solvable in polytime
  - Let's see examples…

Decision version of problems

Shortest path

- Search version:
  - Given a graph and vertices s,t,
    what is the length of the shortest path from s to t?
- Decision version:
  - Given a graph, vertices s,t, and a **budget b**,
    is there a path from s to t of length **at most b**?

GCD

- Search version:
  - Given two numbers x and y, what is gcd(x,y)?
- Decision version:
  - Given two numbers x and y, and a **threshold b**, is gcd(x,y) **at most b**?

For these problems, if you can solve the decision version,
you solve the search version too. How?

Example: Solving search problems using decision problems

Suppose we have M where

- $M(x, y, b)$ accepts if $\gcd(x, y) \leq b$.
- M runs in polynomial time in the input size
  - Input size: log(x)+log(y)+log(b)

Goal: compute $\gcd(x, y)$ in polynomial time, i.e., poly(log(x)+log(y))

Bad approach:

- For $i = 0, \ldots, \min\{x, y\}$: if $M(x, y, i)$ accepts, return i.
- What's wrong?
  - It is correct, but…
  - It takes $\Omega(\min\{x, y\})$ iterations. Not polynomial in log(x)+log(y)

Good approach: **binary search** in the range of $[0,\ldots,\min\{x, y\}]$.

- Total time: poly(log(x)+log(y))

The Complexity Class **P**

Definition:

- **P** is the set of all decision problems that can be decided in polynomial time.

Formally:

- For any problem **L,** an efficient decider **Decide-L** for **L** is such that
  - **x** is a "yes" instance ⇔ **Decide-L**(**x**) accepts
  - **x** is a "no" instance ⇔ **Decide-L**(**x**) rejects (follows from above)
  - **Decide-L**(**x**) runs in poly(|**x**|) time
- **P** is the set of all decision problems that have an efficient decider

non deterministic polynomial

Class **NP:**
problems we can verify "fast"

Common mistakes:
NP does **not** stand for "Not Polynomial"

What does **verify** mean?

- **Example 1:** Given a sudoku puzzle, is there a solution?
- **Answer:** Yes.
- **Reply:** We are not convinced (i.e. you could be lying to us).
- **Reply:** Now we are convinced.

			2	6		7		1
6	8			7			9	
1	9				4	5		
8	2		1				4	
		4	6		2	9		
	5				3		2	8
		9	3				7	4
	4			5			3	6
7		3		1	8			

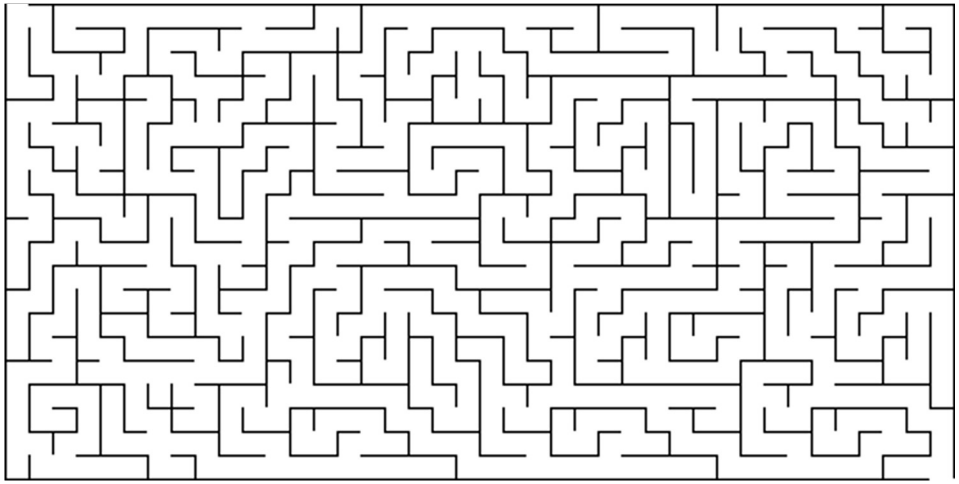
4	3	5	2	6	9	7	8	1
6	8	2	5	7	1	4	9	3
1	9	7	8	3	4	5	6	2
8	2	6	1	9	5	3	4	7
3	7	4	6	8	2	9	1	5
9	5	1	7	4	3	6	2	8
5	1	9	3	2	6	8	7	4
2	4	8	9	5	7	1	3	6
7	6	3	4	1	8	2	5	9

If there is no solution, we will not ever be convinced either.

What does **verify** mean?

Consider a maze.

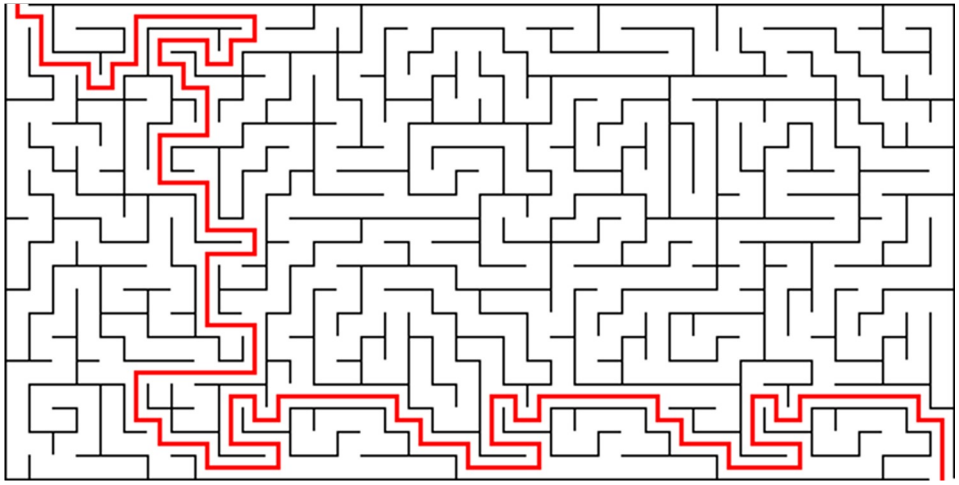
It might be hard to solve…



What does **verify** mean?

But if you give me the solution,

I can verify that it's a valid solution.



If there is no solution, we will not ever be convinced either.

The Complexity Class **NP**

Definition:

- **NP** is the set of all decision problems whose yes-instances can be verified in polynomial time.

Common mistakes: NP does **not** stand for "Not Polynomial"

- NP stands for "Nondeterministic Polynomial"
- We will not talk about non-determinism in this class though.

"A better name would have been **VP: verifiable** in polynomial time."

-Clyde Kruskal

The Complexity Class **NP**

Definition:

- **NP** is the set of all decision problems whose yes-instances can be verified in polynomial time.

Formally:

- For any problem **L**, an efficient verifier **Verify-L** for **L** is such that
  - **x** is a "yes" instance ⇔ ∃**C** **Verify-L**(**x**, **C**) accepts
  - **x** is a "no" instance ⇔ ∀**C** **Verify-L**(**x**, **C**) rejects (follows from above)
  - **Verify-L**(**x**, **C**) runs in poly(|**x**|) time

- **NP** is the set of all decision problems that have efficient verifiers
- If **Verify-L**(**x**, **C**) accepts, then **C** is called a certificate.

Intuition:

- If the input has a ~~solution~~,
  then we can efficiently verify that given some *additional information*
- If there is *no solution*,
  then no additional information (even maliciously produced) could convince us

Nondeterministic Turing Machines

Definition:

A nondeterministic Turing machine is defined in the expected way. At any point in a computation, the machine may proceed according to several possibilities. The transition function for a nondeterministic Turing machine has the form

$$\delta: Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}).$$

The computation of a nondeterministic Turing machine is a tree whose branches correspond to different possibilities for the machine. If some branch of the computation leads to the accept state, the machine accepts its input.

otherwise: $\forall c$, $V(x, c)$ 都 rej
$\Rightarrow$ no.

The Running Time of a Deterministic Turing Machine

Definition:

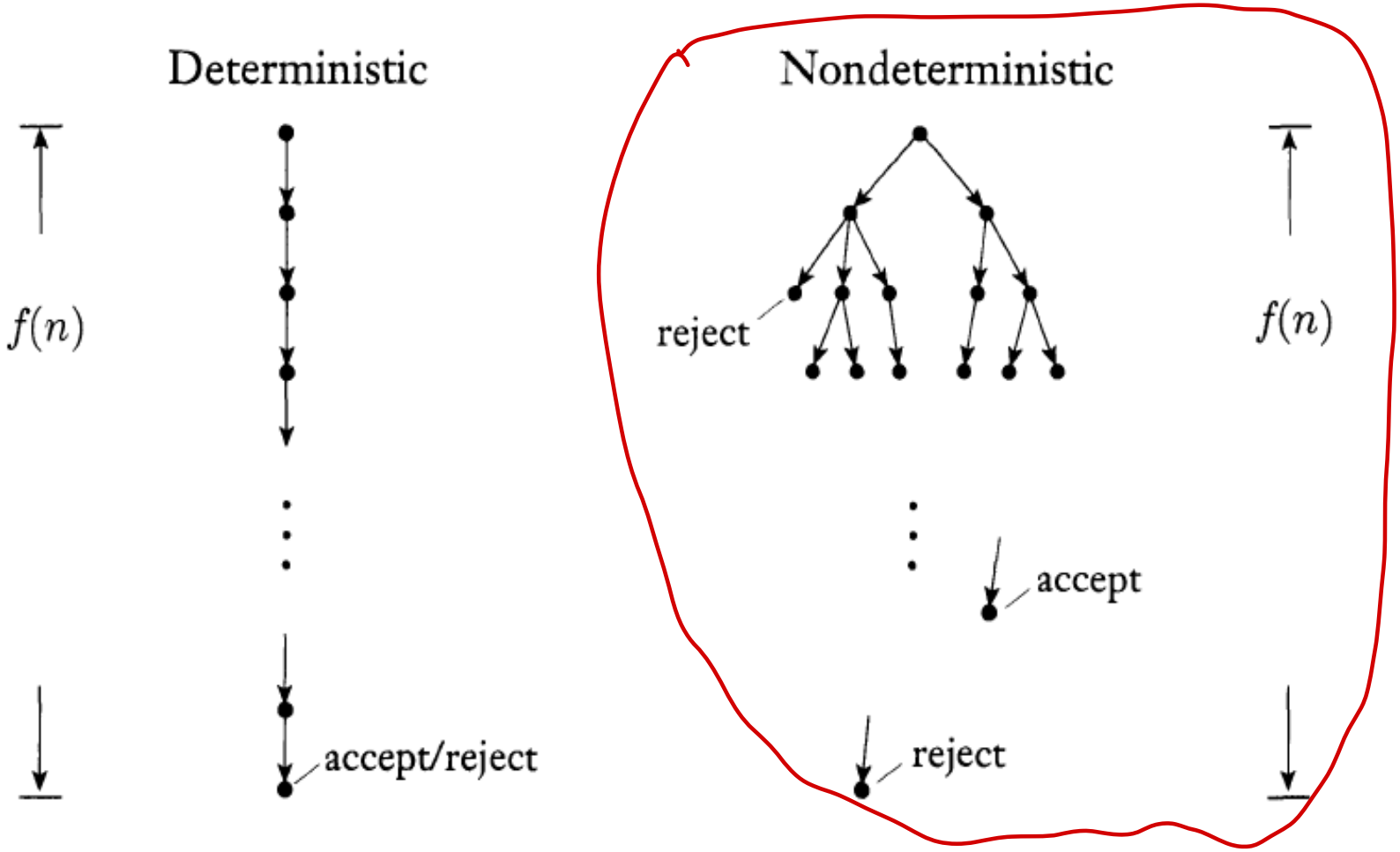
Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

The Running Time of a Nondeterministic Turing Machine

Definition:

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.

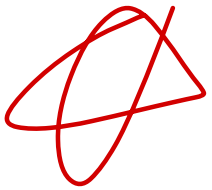
Measuring Deterministic and Nondeterministic Running Time



Equivalent Definition of NP

Theorem:

A language is in NP if and only if some nondeterministic polynomial time Turing machine decides it.



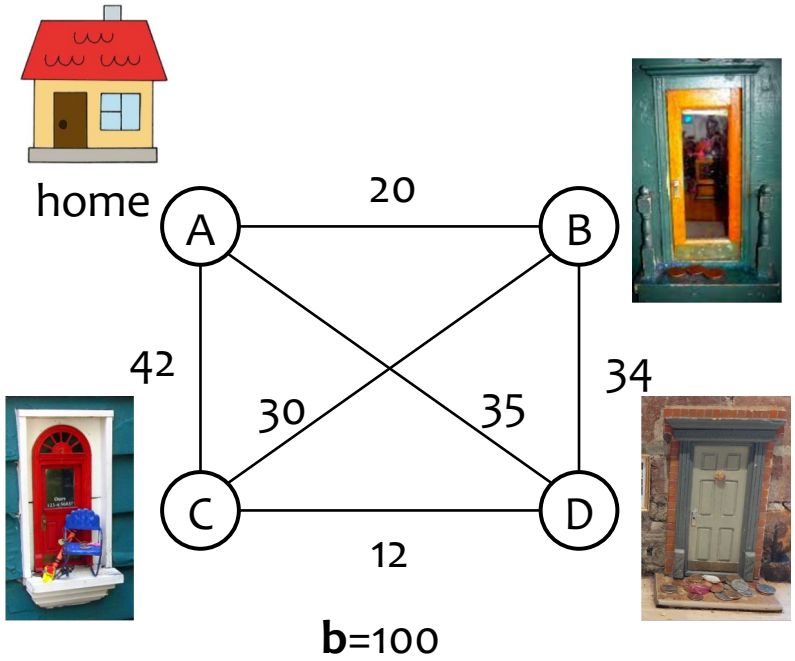
Prove that a problem is in **NP:**

Showing efficient verifiers

Traveling Salesperson Problem (TSP)

Input: n vertices, distances between each pair of vertices, budget b

Output: Is there a length ≤**b** cycle containing every vertex exactly once?



TSP is in NP

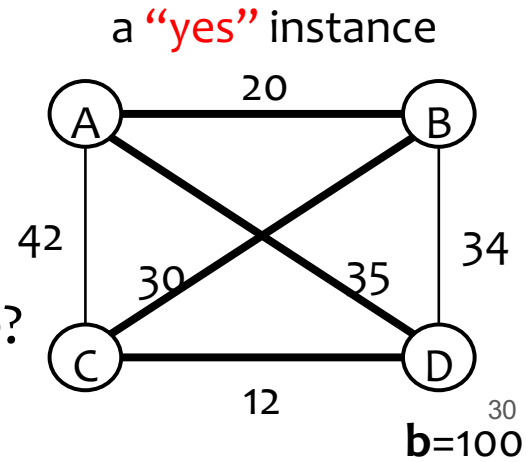
Recall:

NP is the set of all decision problems **L** that have efficient verifiers **Verify-L**

- **x** is a "yes" instance ⇔ ∃**C** **Verify-L**(**x, C**) accepts
- **Verify-L**(**x, C**) runs in poly(|**x**|) time

Example: TSP:

- **Certificate C:** Length ≤**b** cycle with all vertices.
- Efficient verifier **Verify-TSP**(⟨**G, b**⟩, **C**):
  - Is **C** a cycle in **G**?
  - Does **C** contain every vertex in **G** exactly once?
  - Do the edge weights of **C** add up to ≤**b**?



TSP is in NP

Recall:

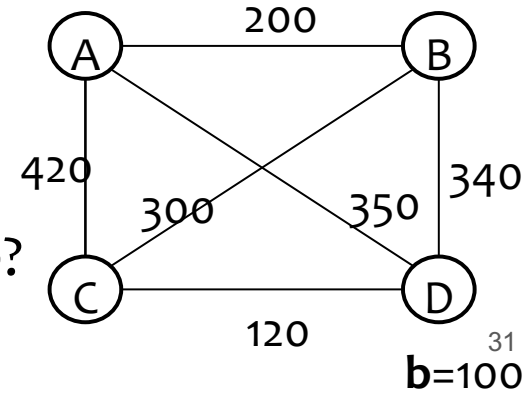
NP is the set of all decision problems **L** that have an efficient verifier **Verify-L**

- **x** is a "yes" instance ⇔ ∃**C** **Verify-L**(**x, C**) accepts
- **Verify-L**(**x, C**) runs in poly(|**x**|) time

Example: TSP:

- **Certificate C:** Length ≤**b** cycle with all vertices.
- Efficient verifier **Verify-TSP**(⟨**G, b**⟩, **C**):
  - Is **C** a cycle in **G**?
  - Does **C** contain every vertex in **G** exactly once?
  - Do the edge weights of **C** add up to ≤**b**?

a "no" instance



TSP is in NP

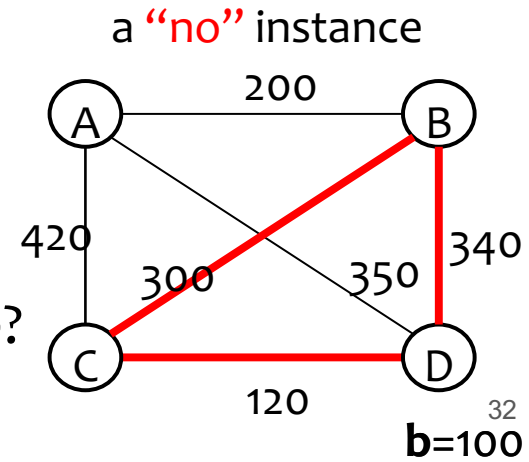
Recall:

NP is the set of all decision problems **L** that have an efficient verifier **Verify-L**

- **x** is a "yes" instance ⇔ ∃**C** **Verify-L**(**x, C**) accepts
- **Verify-L**(**x, C**) runs in poly(|**x**|) time

Example: TSP:

- **Certificate C:** Length ≤**b** cycle with all vertices.
- Efficient verifier **Verify-TSP**(⟨**G, b**⟩, **C**):
  - Is **C** a cycle in **G**?
  - Does **C** contain every vertex in **G** exactly once?
  - Do the edge weights of **C** add up to ≤**b**?



To show that a problem is in **NP**, you need to specify (e.g. for the HW):

1. Certificate
2. Efficient verifier
3. Proof of correctness of verification algorithm

Example: TSP:

1. **Certificate C:** Length ≤**b** cycle containing every vertex.
2. Efficient verifier: **Verify-TSP**(⟨**G, b**⟩, **C**):
   Is **C** a cycle in **G**?
   Does **C** contain every vertex in **G** exactly once?
   Do the edge weights of **C** add up to ≤**b**?
   Accept if all 3 answers are "yes"
   (you'd need to analyze the running time)
3. TSP "yes" instance ⇒ exists a length ≤**b** cycle containing every vertex
   ⇒ ∃**C** **Verify-TSP**(⟨**G, b**⟩, **C**) accepts

   TSP "no" instance ⇒ no length ≤**b** cycle containing every vertex
   ⇒ ∀**C** **Verify-TSP**(⟨**G, b**⟩, **C**) rejects

Useful fact: Certificate has poly length

Reminder:

- For any problem **L,** an efficient verifier **Verify-L** for **L** is such that
  - **x** is a "yes" instance ⇔ ∃**C** **Verify-L**(**x, C**) accepts
  - **Verify-L**(**x, C**) runs in poly(|**x**|) time

- If **Verify-L**(**x, C**) accepts, then **C** is called a certificate.

Claim: without loss of generality, we can assume |**C**| ≤ poly(|**x**|).

Proof:

- **Verify-L**(**x, C**) runs in poly(|**x**|) time.
- It reads only poly(|**x**|) symbols of **C**. So, we can remove the rest.

Subset Sum is in NP

Input: a set S of integers and a target t.

Output: Is there a subset of the integers in S whose sum is exactly t?

Prove that Subset Sum is in **NP**.

- Certificate: a subset **C** ⊆ S whose sum is t.
- Verifier: **Verify**(**S, t, C**): Check that **C** ⊆ S and the sum of **C** is t.
- Analysis:
  - **(S, t)** is a "yes" instance ⇔ ∃**C** **Verify**(**S,t, C**) accepts
  - **Verify**(**x, C**) runs in poly(**|S| log t**) time

Terminology on Satisfiability (SAT)

A **Boolean formula** Φ is made up of:

- "literals": variables and their negations (e.g. x, y, z, ¬x, ¬y, ¬z)
- OR: ∨
- AND: ∧

Example:

$$\Phi_1 = (x \vee y) \wedge (\neg y \vee x \vee \neg z) \wedge (\neg x \vee (y \wedge \neg z))$$

Φ is ***satisfiable*** if

- ∃ a true/false assignment **A** to the variables so that Φ(**A**) = true
- For example, Φ_1 is satisfiable.
  - Assign x = F, y = T, z = F

SAT is in NP

Input: A Boolean formula Φ

Output: Is Φ *satisfiable*?

Prove that SAT is in **NP**.

- Certificate: a true/false assignment **C** to variables where Φ(**C**) = true
- Verifier: **Verify**(Φ, **C**): Check that Φ(**C**) = true
- Analysis:
  - Φ is a "yes" instance ⇔ ∃**C** **Verify**(Φ, **C**) accepts
  - **Verify**(Φ, **C**) runs in poly(|Φ|) time

THE Major Open Problem in Computer Science

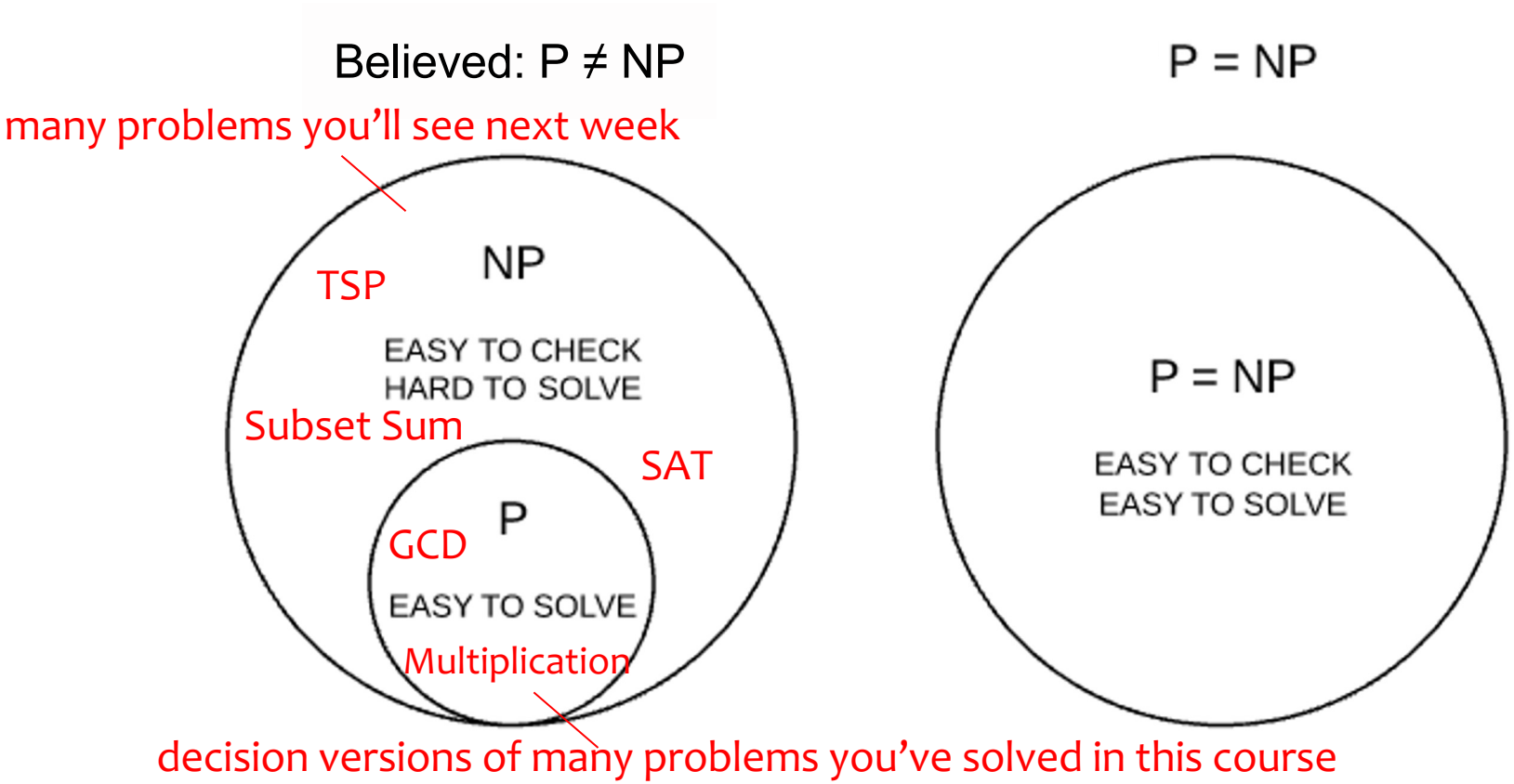
$$P \stackrel{?}{=} NP$$

"Is every efficiently verifiable problem also efficiently solvable?"

"If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in 'creative leaps', no fundamental gap between solving a problem and recognizing the solution once it's found."

- Scott Aaronson

Two Possible Worlds



Millennium Prize Problems

- The **Millennium Prize Problems** are seven well-known complex mathematical problems selected by the Clay Mathematics Institute in 2000. The Clay Institute has pledged a US$1 million prize for the first correct solution to each problem.

CMI

ABOUT PROGRAMS MILLENNIUM PROBLEMS PEOPLE PUBLICATIONS EVENTS EUCLID

Millennium Problems

Yang–Mills and Mass Gap

Experiment and computer simulations suggest the existence of a "mass gap" in the solution to the quantum versions of the Yang-Mills equations. But no proof of this property is known.

Riemann Hypothesis

The prime number theorem determines the average distribution of the primes. The Riemann hypothesis tells us about the deviation from the average. Formulated in Riemann's 1859 paper, it asserts that all the 'non-obvious' zeros of the zeta function are complex numbers with real part 1/2.

P vs NP Problem

If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

Navier–Stokes Equation

This is the equation which governs the flow of fluids such as water and air. However, there is no proof for the most basic questions one can ask: do solutions exist, and are they unique? Why ask for a proof? Because a proof gives not only certitude, but also understanding.

Hodge Conjecture

The answer to this conjecture determines how much of the topology of the solution set of a system of algebraic equations can be defined in terms of further algebraic equations. The Hodge conjecture is known in certain special cases, e.g., when the solution set has dimension less than four. But in dimension four it is unknown.

Poincaré Conjecture

In 1904 the French mathematician Henri Poincaré asked if the three dimensional sphere is characterized as the unique simply connected three manifold. This question, the Poincaré conjecture, was a special case of Thurston's geometrization conjecture. Perelman's proof tells us that every three manifold is built from a set of standard pieces, each with one of eight well-understood geometries.

Birch and Swinnerton-Dyer Conjecture

Supported by much experimental evidence, this conjecture relates the number of points on an elliptic curve mod p to the rank of the group of rational points. Elliptic curves, defined by cubic equations in two variables, are fundamental mathematical objects that arise in many areas: Wiles' proof of the Fermat Conjecture, factorization of numbers into primes, and cryptography, to name three.