I can't find an efficient algorithm, but neither can all these famous people.
There is a big chasm between poly-time and exp-time.

- poly-time solvable
  - testing primality
  - matrix multiplication
  - MST
  - max matching
  - shortest path
  - ...

- exp-time solvable
  - subset-sum
  - scheduling
  - TSP
  - Hamiltonian cycle
  - Pokémon
  - ...

best we can say:
Subset Sum Problem

Given a list of integers, determine if there is a subset of the integers that sum to 0.

| 4 | -3 | -2 | 7 | 99 | 5 | 1 |
Exponential running time examples

Subset Sum Problem

Given a list of integers, determine if there is a non-empty subset of the integers that sum to 0.

| 4 | -3 | -2 | 7 | 99 | 5 | 1 |

Exhaustive Search (Brute Force Search):

> Try every possible subset and see if it sums to 0.

# subsets is \(2^n\) \(\implies\) running time at least \(2^n\)

Note: checking if a given subset sums to 0 is easy.
Exponential running time examples

Theorem Proving Problem
(informal description)

Given a mathematical proposition $P$ and an integer $k$, determine if $P$ has a proof of length at most $k$.

Exhaustive Search (Brute Force Search):

$> \text{Try every possible "proof" of length at most } k, \text{ and check if it corresponds to a valid proof.}$

**Note:** checking if a given proof is correct is easy.
**Traveling Salesperson Problem (TSP)**

Is there an order in which you can visit the cities so that ticket cost is < $50000?

**Exhaustive Search (Brute Force Search):**

> Try every possible order and compute the cost.

**Note:** checking if a given solution has the desired cost is easy.
Traveling Salesperson Problem (TSP)

**Input:**
A graph $G = (V, E)$, edge weights $w_e$ (non-negative, integral) and target $t$.

**Output:**
Yes, iff there is a cycle of cost at most $t$ that visits every vertex exactly once.
**Traveling Salesperson Problem (TSP)**

**Input:**
A graph $G = (V, E)$, edge weights $w_e$ (non-negative, integral) and target $t$.

**Output:**
Yes, iff there is a cycle of cost at most $t$ that visits every vertex exactly once.
Exponential running time examples

Satisfiability Problem (SAT)

**Input:** A Boolean propositional formula.

* e.g. \( (x_1 \land \neg x_2) \lor (\neg x_1 \land x_3 \land x_4) \lor x_3 \)

**Output:** Yes iff there is an assignment to the variables that makes the formula True.

**Exhaustive Search (Brute Force Search):**

> Try every possible truth assignment to the input variables. Evaluate the formula to see the output.

**Note:** checking if a given truth assignment makes the formula True is easy.
Exponential running time examples

Circuit Satisfiability Problem (Circuit-SAT)

**Input:** A Boolean circuit.

**Output:** Yes iff there is an assignment to the input gates that makes the circuit output 1.

Exhaustive Search (Brute Force Search):

> Try every possible truth assignment to the input gates. Evaluate the circuit to see the output.

**Note:** checking if a given assignment makes the circuit output 1 is easy.
Sudoku Problem

Given a partially filled $n$ by $n$ sudoku board, determine if there is a solution.
Sudoku Problem
Given a partially filled \( n \) by \( n \) sudoku board, determine if there is a solution.

Exhaustive Search (Brute Force Search):
> Try every possible way of filling the empty cells and check if it is valid.

**Note:** checking if a given solution is correct is easy.
In our quest to understand efficient computation, (and life, the universe, and everything) we come across:

P vs NP problem

“Can creativity be automated?”

Biggest open problem in all of Computer Science. One of the biggest open problems in all of Mathematics.
So what is the P vs NP question?

The P vs NP question is the following:

Can the Sudoku problem be solved in polynomial time?

WTF?!
So what is the P vs NP question?

The P vs NP question is the following:

Can the Subset Sum problem be solved in poly-time?

4  -3  -2  7  99  5  1
The P vs NP question is the following:

Can the Traveling Salesperson (TSP) problem be solved in poly-time?
The P vs NP question is the following:

Can the Theorem Proving problem be solved in poly-time?
What the &$$%# is going on?!?

Let’s explain from the beginning.
Toolbox of a computer scientist

1. Basic algorithmic techniques
   e.g. greedy algorithms, divide and conquer, dynamic programming, linear programming, semi-definite programming, etc…

2. Basic data structures
   e.g. queues, stacks, hash tables, binary search trees, etc…

3. Identifying and dealing with intractable problems
3. Identifying and dealing with intractable problems

After decades of research and billions of dollars of funding, no one was able to come up with poly-time algs for:

- Theorem Proving, TSP, Subset Sum, Sudoku, Tetris, …

It would be fantastic if we could prove that these cannot be solved in poly-time. But…
3. Identifying and dealing with intractable problems

But we are far from accomplishing this.

(maybe these problems are in P???)

So what can we do???

Maybe we can try to gather evidence that these problems are hard.
Goal:

Find evidence these problems are computationally hard.
Revisiting reductions

A central concept used to compare the “difficulty” of problems.

Now we are interested in poly-time decidability vs not poly-time decidability

Want to define: \( A \leq B \) (\( B \) is at least as hard as \( A \) w.r.t. poly-time decidability.)

\[
\begin{align*}
B \text{ poly-time decidable} & \implies A \text{ poly-time decidable} \\
B \in P & \implies A \in P \\
A \text{ not poly-time decidable} & \implies B \text{ not poly-time decidable} \\
A \notin P & \implies B \notin P
\end{align*}
\]
Revisiting reductions

**Notation:**  \[ A \leq_T^P B \quad (A \text{ poly-time reduces to } B) \]
if there is a poly-time machine \( M_A \) that decides \( A \)
using an oracle \( M_B \) for \( B \) as a black-box subroutine.

\[ B \text{ in } \mathbf{P} \iff A \text{ in } \mathbf{P} \]
\[ A \text{ not in } \mathbf{P} \iff B \text{ not in } \mathbf{P} \]
Revisiting reductions

def $M_B(\ldots)$:
    # some code that solves problem B

def $M_A(\ldots)$:
    # some code that solves problem A
    # that makes calls to function $M_B$ when needed

If $M_B$ poly-time $\implies M_A$ poly-time

then we would write $A \leq^{P_T} B$.

When you want to show $A \leq^{P_T} B$, you need to come up with a poly-time $M_A$. 
Revisiting reductions

Example

A: Given a graph and an integer k, does there exist at least k pairs of vertices connected to each other?

B: Given a graph and a pair of vertices (s,t), is s and t connected?

A poly-time reduces to B
A: Given a sequence of integers, and a number k, is there an increasing subsequence of length at least k?

\[3, 1, 5, 2, 3, 6, 4, 8\]

B: Given two sequences of integers, and a number k, is there a common subsequence of length at least k?

\[3, 1, 5, 2, 3, 6, 4, 8\]
\[1, 5, 7, 9, 2, 4, 1, 0, 2, 0, 3, 0, 4, 0, 8\]

A poly-time reduces to B
The two sides of reductions

1. Expand the landscape of tractable problems.

If \( A \leq^P_T B \) and \( B \) is tractable, then \( A \) is tractable.

\[
B \in \text{P} \implies A \in \text{P}
\]

Whenever you are given a new problem to solve:

- check if it is already a problem you know how to solve in disguise.

- check if it can be reduced to a problem you know how to solve.
2. Expand the landscape of intractable problems.

If \( A \leq_{PT} B \) and \( A \) is intractable, then \( B \) is intractable.

\[ A \notin \mathbf{P} \implies B \notin \mathbf{P} \]

But we are pretty lousy at showing a problem is intractable.

Maybe we can still make good use of this…
Gathering evidence for intractability

Suppose we want to gather evidence that \( A \not\in P \).

If we can show \( L \leq_P A \) for many \( L \) (including some \( L \) that we really think should not be in \( P \)) then that would be good evidence that \( A \not\in P \).
Definitions of $C$-hard and $C$-complete

**Definition:** Let $C$ be a set of languages containing $P$.

We say that language $A$ is $C$-hard if for all $L \in C$, $L \leq^P_T A$.

$A$ is at least as hard as every language in $C$.

**Definition:** Let $C$ be a set of languages containing $P$.

We say that language $A$ is $C$-complete if
- $A$ is $C$-hard
- $A \in C$

$A$ is a representative for hardest languages in $C$. 
Observation:

Suppose $A$ is $\mathbf{C}$-complete.

- If $\mathbf{C} = \mathbf{P}$, then $A \in \mathbf{P}$.
- If $A \in \mathbf{P}$, then $\mathbf{C} = \mathbf{P}$.

If we believe $\mathbf{C} \neq \mathbf{P}$, then we must believe $A \notin \mathbf{P}$. 

$$\mathbf{C} = \mathbf{P} \iff A \in \mathbf{P}$$
Recall the goal

Good evidence that $A$ is intractable:

- $A$ is $C$-hard for a really rich set $C$
  
  (a set $C$ such that we believe $C \neq P$)

So what is a good choice for $C$?

(if we want to show TSP, Subset-Sum, Sudoku, etc… are $C$-hard?)
What if we let $\mathcal{C}$ be the set of all languages decidable using Brute Force Search (BFS)?

Can it be true that TSP is $\mathcal{C}$-hard?
A complexity class for BFS?

How can we identify the problems solvable using BFS?
What would be a reasonable definition?

What is common about TSP, Subset-Sum, Theorem Proving Problem, etc…?

Seems hard to find a correct solution (solution space is too big!)

BUT, easy to verify a given solution.
Informally:

A language is in $\textbf{NP}$ if:

whenever we have a \textbf{Yes} instance,
there is a “\textit{simple}” proof (solution) for this fact.

1. The length of the proof is polynomial in the input size.
2. The proof can be verified/checked in polynomial time.
The complexity class **NP**

**Formally:**

**Definition:**

A language \( A \) is in **NP** if

- there is a polynomial-time TM \( V \)
- a polynomial \( p \)

such that for all \( x \in \Sigma^* \):

\[
x \in A \iff \exists u \text{ with } |u| \leq p(|x|) \text{ s.t. } V(x, u) = 1
\]

If \( x \in A \), there is some proof that leads \( V \) to accept.

If \( x \notin A \), every “proof” leads \( V \) to reject.
The complexity class $\textbf{NP}$

Formally:

**Definition:**

A language $A$ is in $\textbf{NP}$ if

- there is a polynomial-time TM $V$
- a polynomial $p$

such that for all $x \in \Sigma^*$:

$x \in A \iff \exists u \text{ with } |u| \leq p(|x|) \text{ s.t. } V(x, u) = 1$

The following are synonyms in this context:

proof = solution = certificate
NP: A game between a Prover and a Verifier

Verifier

poly-time
skeptical

Prover

omniscient
untrustworthy

Given some input $x$ (known both to Verifier and Prover)

Prover wants to convince Verifier that $x \in A$.

Prover cooks up a “proof” $u$ and sends it to Verifier.

Verifier (in poly-time), should be able to tell if the proof is legit.
NP: A game between a Prover and a Verifier

Verifier: poly-time
omniscient
skeptical

Prover: omniscient
untrustworthy

“Completeness”
If \( x \in A \), there must be some proof \( u \) that convinces the Verifier.

“Soundness”
If \( x \not\in A \), no matter what “proof” Prover gives, Verifier should detect the lie.
NP: A game between a Prover and a Verifier

Verifier
- poly-time
- skeptical

Prover
- omniscient
- untrustworthy

If we have completeness and soundness, then

\[ A \in \text{NP}. \]
CLIQUE

**Input:** $\langle G, k \rangle$ where $G$ is a graph and $k$ is a positive int.

**Output:** Yes iff $G$ contains a clique of size $k$.

**Fact:** CLIQUE is in NP.
Examples of languages in $\textbf{NP}$

**Proof:** We need to show a verifier TM $V$ exists as specified in the definition of $\textbf{NP}$.

$$\text{def } V(x, u) :$$

- if $x$ is not an encoding $\langle G = (V, E), k \rangle$ of a valid graph $G$ and a positive integer $k$, **REJECT**.

- if $u$ is not an encoding of a set $S \subseteq V$ of size $k$, **REJECT**.

- for each pair of vertices in $S$:
  - if the vertices are not neighbors, **REJECT**.
  - **ACCEPT**
Examples of languages in \textbf{NP}

\textbf{Proof (continued):}

Need to show:

1. if $x \in \text{CLIQUE}$, there is some proof $u$ (of poly-length) that makes $V$ ACCEPT.

2. if $x \notin \text{CLIQUE}$, no matter what $u$ is, $V$ REJECTS.

3. $V$ is polynomial-time.

\text{(we leave 3 as an exercise)}
Examples of languages in \textbf{NP}

\textbf{Proof (continued)}:

Need to show:

1. If \( x \in \text{CLIQUE} \), there is some proof \( u \) (of poly-length) that makes \( V \) ACCEPT.

If \( x \in \text{CLIQUE} \), then \( x = \langle G, k \rangle \) is a valid encoding, and \( G \) contains a clique of size \( k \).

Then when \( u \) is a valid encoding of this clique, the verifier will accept.
Examples of languages in NP

Proof (continued):

Need to show:

2. if \( x \notin \text{CLIQUE} \), no matter what \( u \) is, \( V \) REJECTS.

if \( x \notin \text{CLIQUE} \), then there are 2 options:

- \( x \) is not a valid encoding \( \langle G, k \rangle \).

- \( x \) is a valid encoding, but \( G \) does not contain a clique of size \( k \).

In either case, \( V \) rejects for any \( u \).

(add a couple of lines of justification)
This would be the proper way of showing that a language is in \textbf{NP}.

However, we usually don’t write it this way.

- We assume implicitly that inputs are automatically checked to be of the correct type.
- Instead of starting with the description of \(V\), we start with the description of the expected proof.
- We describe things at a very high level and skip many details.
3COL

**Input:** \( \langle G \rangle \) where \( G \) is a graph.

**Output:** Yes iff \( G \) is 3-colorable.

**Fact:** 3COL is in NP.
Examples of languages in NP

Proof (sort of):
The proof string is a valid coloring of the vertices with 3 colors.

The verifier goes through each edge one by one and checks that the endpoints are different colors.

If the input graph is 3-colorable, this check will succeed for a valid 3-coloring of the vertices.

If the input graph is not 3-colorable, then no matter what 3-coloring is given, the verifier will be able to find an edge whose endpoints are colored the same.

The verifier is poly-time since going through each edge and checking their colors takes poly-time.
Examples of languages in **NP**

**CIRCUIT-SAT**

**Input:** \(\langle C \rangle\) where \(C\) is a Boolean circuit.

**Output:** Yes iff \(C\) is satisfiable.

**Fact:** CIRCUIT-SAT is in **NP**.
The complexity class \( \textbf{NP} \)

2 Observations:

1. Every decision problem in \( \textbf{NP} \) can be solved using BFS.
   - Go through all possible proofs \( u \), and run \( V(x, u) \)

2. This is a pretty BIG class!
   
   Contains everything in \( \textbf{P} \).  (recitation)

\( \textbf{NP} \) contains much more than \( \textbf{P} \).
Coming back to our goal

We wanted to find evidence that TSP, Subset-Sum, Theorem Proving problem, etc. are not in $P$.

Could it be true that one of them is $\text{NP}$-complete?

Is there any language that is $\text{NP}$-complete?

Is $\text{NP}$-completeness a useful definition?
The Cook-Levin Theorem

Theorem (Cook 1971 - Levin 1973):
SAT is \( \textbf{NP} \)-complete.

So SAT is in \( \textbf{NP} \). (easy)

And for every \( L \) in \( \textbf{NP} \), \( L \leq_{PT}^{P} \text{SAT} \).
Karp’s 21 NP-complete problems

1972: “Reducibility Among Combinatorial Problems”

0-1 Integer Programming
Clique
Set Packing
Vertex Cover
Set Covering
Feedback Node Set
Feedback Arc Set
Directed Hamiltonian Cycle
Undirected Hamiltonian Cycle
3SAT

Partition
Clique Cover
Exact Cover
Hitting Set
Knapsack
Steiner Tree
3-Dimensional Matching
Job Sequencing
Max Cut
Chromatic Number
Today

Thousands of problems are known to be $\text{NP}$-complete.
(including the languages mentioned in this lecture)

1979
Some other “interesting” examples

Super Mario Bros
Given a Super Mario Bros level, is it completable?

Tetris
Given a sequence of Tetris pieces, and a number k, can you clear more than k lines?
How do you show a language is \textbf{NP}-complete?

Seems like an unbelievably strong statement.
How could one possibly prove such a thing?!?

Once you have one, things are easier.

If \( \text{SAT} \leq_{PT} L \), then \( L \) is \textbf{NP}-hard.

\( \text{(transitivity of } \leq_{PT} \text{)} \)
How do you show a language is \textbf{NP}-complete?

It is similar to showing undecidability.

- we need an initial direct proof that a language is \textbf{NP}-hard. (Cook-Levin Theorem)

- to show other languages are \textbf{NP}-hard, we use poly-time reductions.

This is the topic of Thursday’s lecture.
Good evidence for intractability?

If $A$ is $\text{NP}$-hard, that seems to be good evidence that $A \notin \text{P}$.

(if you believe $\text{P} \neq \text{NP}$)

But is $\text{P} \neq \text{NP}$??
The P vs NP Question
The P vs NP question

After years of research:

We are pretty confident that this is one of the deepest questions we have ever asked.
The two possible worlds

The left diagram shows two possible worlds.

In the left world, the complexity classes satisfy the relations:

- \( \text{NP} \subseteq \text{NP-hard} \)
- \( \text{NP} \cap \text{NP-hard} = \text{NP-c} \)
- \( \text{NP} \cap \text{NP-c} = \text{P} \)

In the right world, the complexity classes satisfy the relations:

- \( \text{P} = \text{NP} = \text{NP-hard} \)

In the left world, \( \text{P} \neq \text{NP} \).

In the right world, \( \text{P} \subseteq \text{NP} \).

The right diagram shows the possible world where \( \text{P} = \text{NP} \).

The left diagram shows the possible world where \( \text{P} \neq \text{NP} \).
What do experts think?

Two polls from 2002 and 2012

# respondents in 2002: 100
# respondents in 2012: 152

<table>
<thead>
<tr>
<th></th>
<th>P ≠ NP</th>
<th>P = NP</th>
<th>Ind</th>
<th>DC</th>
<th>DK</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>61(61%)</td>
<td>9(9%)</td>
<td>4(4%)</td>
<td>1(1%)</td>
<td>22(22%)</td>
</tr>
<tr>
<td>2012</td>
<td>126(83%)</td>
<td>12(9%)</td>
<td>5(3%)</td>
<td>5(3%)</td>
<td>1(0.6%)</td>
</tr>
</tbody>
</table>
What does NP stand for anyway?

Not Polynomial?
None Polynomial?
No Polynomial?
No Problem?
Nurse Practitioner?

It stands for **Nondeterministic Polynomial time**.

Languages in NP are the languages decidable in polynomial time by a nondeterministic TM.

DFA  ↔  SFA (actually called NFA)
TM  ↔  NTM
What does NP stand for anyway?

Other contenders for the name of the class:

- Herculean
- Formidable
- Hard-boiled
- PET
  - “possibly exponential time”
  - “provably exponential time”
  - “previously exponential time”
Summary
Summary

- How do you identify intractable problems? (problems not in $\mathbb{P}$) e.g. SAT, TSP, …

- We are not able to prove they are intractable. Can we gather some sort of evidence?

- Poly-time reductions $A \leq_{PT}^P B$ are useful to compare hardness of problems.

- Evidence for intractability of $A$:
  Show $L \leq_{PT}^P A$, for all $L \in C$, for a large class $C$.

- Definitions of $C$-hard, $C$-complete.

- What is a good choice for $C$, if we want to show, say, SAT is $C$-hard?
Summary

• The complexity class **NP** (take **C = NP**)

• **NP**-hardness, **NP**-completeness

• Cook-Levin Theorem: SAT is **NP**-complete

• Many other languages are **NP**-complete.

• If L is **NP**-hard, is this good evidence it is intractable (i.e., L not in **P**)?

• The **P** vs **NP** question
How did Cook-Levin show SAT is $\mathbf{NP}$-complete?

And examples of poly-time reductions that show other problems are $\mathbf{NP}$-complete.