TIME
COMPLEXITY
AND BIG-O
NOTATION
"Time complexity" = Running time for programs

- Consider only computable functions. → Decidable (Always halt)

- Consider only deterministic machines
  That "guess the right thing" or "try all possibilities" is a questionable operation on a real machine.

- Consider some input, w.
  For TMs: Just count the transitions.

- Consider all inputs of size N.

  What is the maximum time the Turing machine might take?

Our goal: Find a function of N to describe the running time.

\[ f(N) = \ldots \]
Often, the function can be ugly!

\[ f(N) = 17N^3 + 5N^2 + 3\log N + 29 \]

For large values of \( N \), we only care about \( N^3 \).

We want the **asymptotic upper bound**.

The "order" (or "Big-O") notation:

\[ f(N) = O(N^3) \]

Also: Ignore constant factors.

\[ \Rightarrow \text{Ignore } 17N^3 \]
Let $M$ be a deterministic Turing machine that always halts.

Let $n$ be the size of an input.

**Defn**

The "**time complexity**" (i.e., the running time) of $M$ is a function $f$.

$$f(n) = \text{the maximum number of steps that } M \text{ takes on any input of size } n.$$ 

**Note**

"Size of input" usually means the length of the input.

... but may sometimes mean something else, such as
- number of nodes in a graph.
- number of rules in a CFG.
- etc.
Big-O Notation

\[ 17n^3 + 5n^2 + 3n + 29 \]

\[ O(n^3) \]

For polynomial functions.
* Take the highest order term
* Ignore the coefficient.

\[ f(n) = 17n^3 + 5n^2 + 3n + 29 \]

We say:
\[ f(n) = O(n^3) \]

Also:
\[ f(n) = O(n^4) = O(n^5) = O(2^n) \]
Let $f(n)$ be some running time function of interest.

We say $f(n) = O(n^3)$ if, for all $n \geq$ some value ($n_0$) (i.e., for all $n$ large enough) $f$ behaves like $n^3$, ignoring constant factors.

More precisely:

$f(n) = O(g(n))$

if $\exists c$ and $\exists n_0$ such that

$f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. 
TYPICAL COMPLEXITY CLASSES

\[ N = \text{linear} \]
\[ \log N \]
\[ N/\log N \]
\[ N^2 \]
\[ N^k \]
\[ 2^N \]  
EXponential

\[ \{ \]
POLYNOMIAL TIME
\[ \} \]

\[ O(N) = \text{linear time algorithms} \]
\[ O(N \log N) \]
\[ O(N^2) \]
\[ O(N^3) \]
\[ O(N^4) \]
\[ O(2^N) \]
Q: Why aren't there many $O(\log N)$ algorithms?

A: The input has size $n$. Just to read all the input requires $O(n)$.
TIME COMPLEXITY CLASSES

TIME(n)
  The set of all languages/problems that can be DECIDED in $O(n)$ time.

TIME($n^2$)
  ... that can be DECIDED in $O(n^2)$ time.

TIME($n \log n$) ... in $O(n \log n)$

TIME($n^3$) ... in $O(n^3)$

TIME($2^n$) ... in exponential time.

etc.

**NOTE:**

TIME(n) ≤ TIME(n log n) ≤ TIME($n^2$) ≤ TIME($n^3$) ≤ TIME($n^k$) ≤ TIME($2^n$)
The Time Complexity of an Example Algorithm
Algorithm to decide $\{0^k1^k \mid k \geq 0 \}$

- Scan input to make sure it is in the form $0^*1^*$.
  - $n$ steps to scan
  - $n$ steps to reposition to left end
  - $2n$ steps $O(n)$

- Repeat while tape contains at least one $0$ and at least one $1$...
  - Scan across tape and change a $0$ to $x$ and a $1$ to $x$.
  - $2n$ steps $O(n)$

- End loop
  - $n/2$ repetitions
  - Whole loop takes $\frac{n}{2} \cdot O(n) = O(n^2)$

- If tape contains all $x$s, then accept else reject.
  - $n$ steps $O(n)$

$O(n) + O(n^2) + O(n) \rightarrow O(n^2)$
Cross off every other 0
Cross off every other 1
Repeat until nothing remains.
At each stage we should have the same number of 0's as 1's.
So: \( \exists 0^k1^k/k \geq 0^3 \in \text{TIME}(n^2) \)

But there is a better algorithm!

- Scan input to make sure it is in the form \( 0^*1^* \) \( O(n) \)

- Repeat while the tape contains at least one 0 and at least one 1...
  - Scan tape to see if number of 0's plus number of 1's is ODD or EVEN \( O(n) \)
  - If ODD then REJECT. \( O(1) \)
  - If no 0's and no 1's remain then ACCEPT, else REJECT \( O(n) \)

So: \( \exists 0^k1^k/k \geq 0^3 \in \text{TIME}(n \log n) \)
Different Models of Computation
What about a different model of computation?

Assume we have multiple tapes.

**Algorithm Using 2 Tapes.**

1. Copy all 0's to tape 2.
   
   \[ \begin{array}{c}
   \text{00000}
   \text{11111}
   \end{array} \]

   Time: \(O(n)\)

2. Reposition tape 2 to beginning.
   
   \[ \begin{array}{c}
   \text{00000}
   \text{11111}
   \end{array} \]

   Time: \(O(n)\)

3. Scan both tapes simultaneously.

4. Make sure both heads hit at the same time.
   
   \[ \begin{array}{c}
   \text{100000}
   \text{000000}
   \text{111111}
   \text{000000}
   \end{array} \]

   Time: \(O(n)\)
**Theorem**

For every multitape Turing machine algorithm that takes time $t(n)$,
There is an equivalent single tape Turing machine that takes time $O(t^2(n))$.

**Proof**

In time $t(n)$, the longest the tapes can be is $t(n)$.
You can simulate the multitape algorithm on a machine with one tape.
Each step of the simulation can be done in $O(t(n))$ time.
To simulate the entire algorithm:

$t(n) \cdot O(t(n)) = O(t^2(n))$
Bottom Line

The model of computation matters!

However, the differences are "relatively small."

A polynomial-time algorithm will remain polynomial-time, regardless of the details of the model of computation!

As long as the machines are deterministic!

The class of Polynomial-time Problems seems quite robust.

(Details of the computer don't matter.)
Non-deterministic T.M.S.

Running Time:
The number of steps the TM uses on the longest branch of computation.

Deterministic Computation History

Non-deterministic Computation History

Running Time

Accept/Reject

Accept

Reject

Reject
Every nondeterministic TM can be simulated on a deterministic TM, using exponentially many more steps.

NonDet TM takes $4^{19}$ steps on input $w$

Det simulation can be done in $2^{4^{19}}$ steps

NonDet TM takes $O(n^2)$ time

Det simulation can be done in $2^{n^2}$ time
The Complexity Classes P and NP
All reasonable deterministic models of computation are **POLYNOMICALLY EQUIVALENT**.

The class of languages that can be decided... [i.e., the set of problems that can be solved... ] in **POLYNOMIAL TIME on a DETERMINISTIC TURING MACHINE**.

\[ P = \bigcup_{k} \text{TIME}(n^k) \]
THE "PATH" PROBLEM

INPUT: A DIRECTED GRAPH $G$ AND TWO NODES, $s$ AND $t$...
IS THERE A PATH FROM $s$ TO $t$?

Is there a path from 7 to 4?

PATH $\exists P$

Proof

- PROVIDE AN ALGORITHM
- SHOW ITS RUNNING TIME.

Use a marking algorithm $O(m^2)$ where $m =$ number of nodes
THEOREM

Every context-free language is in P.

PROOF

Provide an $O(n^3)$ algorithm.
A "dynamic programming" algorithm.

- Use a table to store partial results.
- Avoid having to recompute things over and over.
- Build bigger results out of smaller results.

For $i = 1$ to $n$.
- Compute all results of size $i$.
- Store each result.
- Make use of results of size $< i$.

End
HAMPATH: The "HAMILTONIAN" Path Problem

Given a directed graph, is there a path that goes through every node exactly once?

\[ \text{HAMPATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph and there is a "HAMILTONIAN" Path from } s \text{ to } t \} \]

We are given the starting and ending nodes.

Example: G, 1, 8

1 3 5 4 2 6 7 8
**Exponential Algorithm**

*Generate all possible paths.*

\[ 1 2 3 4 5 6 7 8 \\
1 4 3 2 8 7 5 6 \]

*Test each path to see if it is legal.*

**Note:** This "test" can be done quickly! **IN POLYNOMIAL TIME**

This problem is in class NP. It seems to require exponential time. But given the answer, we can verify it in polynomial time.
Definition of NP

Polynomial Verifiability
POLYNOMIAL VERIFIABILITY

Given a language $A$, a "VERIFIER" is an algorithm that is given some extra information, $c$, which it can use to check (in polynomial time) to verify that $w$ is in $A$.

EXAMPLE: HAMPATH

Given a problem, such as $w = <G, s, t>$ is there a Hamiltonian Path? EXPONENTIALLY HARD [Probably]

But the verifier algorithm is passed some info: $c = "13542678"$ and can then confirm that $w \in \text{HAMPATH}$ IN POLYNOMIAL TIME.
**DEFINITION**

A "VERIFIER" for a language $A$ is an algorithm $V$ where

$$A = \{ w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c \}$$

A "POLYNOMIAL-TIME VERIFIER" runs in polynomial time in the length of $w$.

A language is "POLYNOMIALLY VERIFIABLE" if it has a polynomial-time verifier.

The string $c$ is called the "CERTIFICATE" (or "PROOF").

We don't care about the length of $c$; but note that a polynomial-time verifier does not have time to read a certificate that is longer than polynomial in the length of $w$. 
"NP" is the class of languages that have polynomial-time verifiers.

Theorem

A language is in NP iff it is decided by some nondeterministic polynomial-time Turing Machine.

Sometimes this is given as the definition of "NP".
PROOF

* Convert a polynomial-time Verifier into an equivalent polynomial-time non-deterministic Turing Machine.

The TM:

**Input:** $w$ (of length $n$)

**Algorithm:**
- Non-deterministically guess string $c$ (length at most $n^k$)
- Run $V$ on $\langle w, c \rangle$
- If $V$ accepts, accept; Else Reject.

* Assume you have a polynomial-time non-deterministic TM. Construct a polynomial-time Verifier.

The Verifier:

**Input:** $\langle w, c \rangle$

**Algorithm:**
- Simulate the Non-deterministic TM.
- Use $c$ as a guide about which choice to make at each step.
- If this branch accepts, then ACCEPT ELSE REJECT.
$P = \text{The class of languages for which membership can be DECIDED quickly.}^*$

$NP = \text{The class of languages for which membership can be VERIFIED quickly.}$

That is, given some information [the "certificate/proof"], you can quickly confirm that $w$ is in the language.

* "quickly" means "in Polynomial time"
**DEFINITION**

\[ \text{NTIME}(t(n)) = \{ L \mid \text{L is a language decided by an } \ O(t(n)) \text{ time nondeterministic T.M.} \} \]

**TIME}(n^2) = \text{The set of languages that can be decided by a DETERMINISTIC T.M. in } O(n^2) \text{ time.} \]

\[ \text{NTIME}(n^2) = \text{The set of languages that can be decided by a NONDETERMINISTIC T.M. in } O(n^2) \text{ time.} \]

\[ \text{NP} = \bigcup_{k} \text{NTIME}(n^k) \]
The "CLIQUE" Problem

Given an undirected graph...
A "clique" is a set of nodes such that every node in the clique is connected to every other node in the clique.
A $k$-clique is a clique with $k$ members.
Does this graph contain a 5-clique?

\[ \text{CLIQUE} = \exists \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \]

**Theorem**

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

**Proof**

- Provide a polynomial-time verifier
- or -
- Provide a polynomial-time nondeterministic Turing machine.
The class \"P\"

The class of languages that can be decided...
[The set of problems that can be solved...

...in polynomial time on a deterministic Turing machine.

The class \"NP\"

The class of languages that can be decided...
[The set of problems that can be solved...

...in polynomial time on a nondeterministic Turing machine.
**Unsolved Question:**

\[
P = \text{NP} \quad \text{? Which is it?} \quad P \subseteq \text{NP}
\]

There are lots of problems known to be in \text{NP}.

• *NONE* of these problems can be solved in poly. time on a deterministic T.M.

These problems seem to require exponential time to solve.
EXPONENTIAL-TIME PROBLEMS

\[ \text{EXPTIME} = \bigcup_{k} \text{TIME}(2^{n^k}) \]

RESULTS

\[ P \subseteq \text{NP} \subseteq \text{EXPTIME} \]

APPARENTLY:

\[ P \subset \text{NP} = \text{EXPTIME} \]

BUT THIS IS ALSO POSSIBLE:

\[ P = \text{NPC} \subset \text{EXPTIME} \]
NP-Complete Problems
NP-COMPLETENESS

- An interesting subset of NP problems. The "NP-Complete Problems."

- If a polynomial time algorithm is ever found (on a deterministic machine) for any "NP-Complete" problem, then \( P = NP \) follows.

... And polynomial time algorithms exist for all problems in NP!

Many interesting problems are NP-complete. They seem to require exponential time.
The Satisfiability Problem “SAT”

Boolean variables: \( x_1, x_2, x_3, \ldots \)

TRUE, FALSE

Boolean operations: \( \land, \lor, \neg \)

Boolean formulas, e.g.,

\[
\emptyset = (\overline{x} \land y) \lor (\overline{y} \land z)
\]

"Satisfiable": If there is an assignment to the variables to make the formula true.

\( x = \text{FALSE}, y = \text{TRUE}, z = \text{FALSE} \)

\[ SAT = \{ \emptyset \mid \emptyset \text{ is a satisfiable Boolean formula} \} \]
SAT ∈ NP

Nondeterministically guess the solution (e.g. X = FALSE, Y = TRUE, ...)
Check that it satisfies the Boolean formula in polynomial time.

THEOREM

SAT ∈ P iff P = NP.
Or equivalently...
SAT is "NP-complete".

Finding a polynomial time algorithm to solve a Boolean formula on a deterministic machine, would:

• Prove that ALL problems in NP have polynomial time algorithms.
• Rock the world.
Proof that SAT is NP-complete
RECALL...

- The Turing Machine Acceptance Problem, $A_{TM} = \{<M,w> | M \text{ accepts } w\}$
- $A_{TM}$ is undecidable.
- We "REDUCED" $A_{TM}$ to an instance of the POST CORRESPONDENCE PROBLEM.
- This proved that the PCP was undecidable.
- We showed how to simulate the execution of a TM with the tiles of a PCP instance.
- The "computation history" was a sequence of "configurations."
- Finding a solution to the PCP was equivalent to finding an accepting computation history.
Proof That SAT \text{\small{E}} \text{\small{P}} \iff P=NP

• A problem is in NP if there is a NONDETERMINISTIC TURING MACHINE that will solve it in Polynomial time.

• Got a Problem? \( \langle N, w \rangle \)

• Convert it into an instance of the SAT problem.
  (A huge Boolean formula)

• Do this conversion in Polynomial time.

• If you can solve this SAT problem in Polynomial time, i.e. if SAT \text{\small{E}} \text{\small{P}},

Then, you can solve any problem in NP in Polynomial time.

Such that, there is a branch in the Nondeterministic computation that ACCEPTS IFF the Boolean formula is SATISFIABLE.
THEOREM ("COOK-LEVIN")
SAT is NP-complete

• Any NP problem can be reduced into a SAT problem.
• This reduction can be done in poly-time.

• So if you can solve \( \text{SAT} \) in poly-time on a det. Machine, you can solve any problem in NP on a det. Machine in poly-time; i.e., \( P=NP \).
Given a problem in NP...

Given: \( N \) = A nondeterministic TM.
\( w \) = An input to that TM.

Convert it into a boolean formula, \( \phi \).
Such that \( \phi \) is satisfiable
iff \( N \) accepts \( w \).
(iff \( N \) has an accepting computation history for \( w \)).
The accepting computation history on \( N \) can take at most \( n^k \) steps, for some \( k \). Therefore, it can use at most \( n^k \) tape cells.

Step 1 in the reduction:

CREATE AN \( n^k \times n^k \) "TABLEAU"

TO MODEL THE ACCEPTING COMPUTATION HISTORY.

(It's big, but polynomial in size)

Step 2:

Create lots of boolean variables to model what could be in each cell of the TABLEAU.

Step 3:

Create a formula to express all the constraints on the TABLEAU to guarantee it models a legal, accepting computation history.
Each cell contains a single symbol.

- `#` ← to mark end-of-tape
- `State` ← Each row should have a state, $q$
- `Tape Symbol` ←

$$Q \times \Gamma \times \Sigma$$
What is in each cell?

What is in cell 5,8?

It could be 0, or 1 or 4 or # or

Create a boolean variable for each possibility.

\[ X_{5,8,0} = \text{TRUE} \quad \text{iff the cell contains "0"} \]
\[ X_{5,8,1} = \text{TRUE} \quad \text{iff the cell contains "1"} \]
\[ X_{5,8,4} = \text{TRUE} \quad \text{iff the cell contains "4"} \]
\[ X_{5,8,\#} = \text{TRUE} \quad \text{iff the cell contains "}\#"\]
\[ X_{5,8,44} = \text{TRUE} \quad \text{iff the cell contains "44"} \]

\[ X_{i,j,s} \quad \text{for all } 1 \leq i,j \leq N^k \]
\[ \text{and } s \in \{0,1,4,#,\#\} \]
Now build the formula.
Goal: Add all constraints to assure the TABLEAU is a legal computational history that accepts.

**CONSTRAINT #1**

Every cell contains exactly one symbol.

**CONSTRAINT #2**

Every The First Row is the starting configuration.

**CONSTRAINT #3**

Some cell contains the symbol $\Phi_{accept}$

**CONSTRAINT #4**

Each Row (i.e. each "configuration") can legally follow the previous configuration, according to the transitions in the Nondeterministic TM we are modelling.
Construct the entire Boolean formula:

\[ \varnothing = \varnothing_{\text{cell}} \land \varnothing_{\text{start}} \land \varnothing_{\text{accept}} \land \varnothing_{\text{move}} \]

This formula, while really huge, is polynomial in size (of \( w \)).

If \( \varnothing \) has a solution, then there is an accepting computation history.

If there is an accepting computation history, then this formula has a solution.

If you can determine whether this formula \( \varnothing \) has a solution in poly-time, then you can determine in poly-time whether a Non-deterministic TM \( M \) will accept \( w \).

\[ \text{SAT} \in \text{P} \implies \text{P} = \text{NP} \]
\[ \phi = \phi_{\text{cell}} \land \phi_{\text{start}} \land \phi_{\text{accept}} \land \phi_{\text{move}} \]

**Constraint #1**

Every cell contains exactly one symbol.

\[ x_{5,8}, \# = \text{TRUE} \iff \text{cell contains } \# \]

\[ x_{5,8}, q_4 = \text{TRUE} \iff \text{cell contains } q_4 \]

\[ \bigvee_{s \in \{q_0, q_1, q_2, q_3, \#\}} x_{i,j,s} \]

\[ \bigwedge_{s \neq t} \left( x_{i,j,s} \lor x_{i,j,t} \right) \]

At least one of the variables for this cell is TRUE

For every pair of variables for this cell, at least one is FALSE

Combining (\( \land \)): Exactly one variable is true. And this is true of all cells, \((i,j)\):

\[ \phi_{\text{cell}} = \bigwedge_{1 \leq i, k \leq N} \left[ \left( \bigvee x_{i,j,s} \right) \land \left( \bigwedge_{s \neq t} \left( x_{i,j,s} \lor x_{i,j,t} \right) \right) \right] \]
\[ \emptyset = \emptyset_{\text{cell}} \land \emptyset_{\text{start}} \land \emptyset_{\text{accept}} \land \emptyset_{\text{move}} \]

**Constraint #2:**
The first row describes the initial configuration.

\[ \emptyset_{\text{start}} = \ldots \land \ldots \land \ldots \land \ldots \]

\[ = x_{1,1,\#} \land x_{1,2,q_0} \land x_{2^n,2^n,\#} \land \ldots \]

\[
\begin{array}{ccccccc}
\# & q_0 & w_1 & w_2 & \ldots & w_n & \# \\
\end{array}
\]

- **Input string:** \( w = w_1, w_2, w_3, \ldots, w_n \)
- **All blanks:**

\[ \ldots \land x_{1,3,w_1} \land x_{1,4,w_2} \land x_{1,5,w_3} \ldots \land x_{1,n+2,w_n} \]

\[ \triangledown \quad w \text{ is in the first } N \text{ cells of the tape.} \]

\[ \ldots \land x_{1,n+3,-} \land x_{1,n+4,-} \land \ldots \land x_{1,N+k+2,-} \]

\[ \triangledown \quad \text{The remaining cells of the tape contain the blank symbol, } - . \]
\[ \phi = \phi_{\text{cell}} \land \phi_{\text{start}} \land \phi_{\text{accept}} \land \phi_{\text{move}} \]

**Constraint #3:**

The ACCEPT state is reached in the computation history.

Some cell somewhere contains \( \phi_{\text{accept}} \).

\[ \phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq 2^n} x_{ij, \phi_{\text{accept}}} \]
\[ \phi = \phi_{\text{cell}} \land \phi_{\text{start}} \land \phi_{\text{accept}} \land \phi_{\text{move}} \]

**Constraint #4:**
Every configuration can legally follow the previous configuration, according to the details of the nondet. TM's transition function.

THE "WINDOW" CENTERED ON CELL i,j.

The transition function tells us what the legal windows are.

\[ \begin{array}{c|c|c}
97 & b \rightarrow a, R & 98 \\
\hline
97 & b \rightarrow a, L & 98 \\
\end{array} \]

\[ \begin{array}{c|c|c|c}
\mathbf{c} & 97 & b & \mathbf{a} \\
\hline
\mathbf{c} & 97 & \mathbf{b} & \mathbf{a} \\
\hline
\mathbf{c} & 98 & \mathbf{a} & \mathbf{c} \\
\end{array} \]
This is a transition:

So these are legal "windows":

\[ T = \{ a, b, c, d, \bar{a}, \bar{b}, \bar{c}, \bar{d} \} \]

For each window, make a formula to describe it.

\[ X_{i,j-1,b} \wedge X_{i,j,b} \wedge X_{i,j+1,b} \]

\[ X_{i+1,j-1,b} \wedge X_{i+1,j,b} \wedge X_{i+1,j+1,b} = W_{37} \]

Given an \( i,j \) position, it must contain (or match) one of the legal windows.

\[ W_i \lor W_{i+1} \lor W_{i+2} \lor \cdots \lor W_{37} \lor \cdots \lor W_{592} \]

Now make sure every window in the table is legal.

\[ \emptyset_{\text{move}} = \bigwedge_{1 \leq i,j \leq N^x} \left( \bigvee \left( \left( \bigwedge \text{all legal windows} \right) \right) \right) \]
SPACE
COMPLEXITY
SPACE COMPLEXITY

How to measure?
The number of cells on the tape that we visit.

[Diagram: Tape with cells labeled with numbers and arrows indicating usage]

THE CLASS P-SPACE

QUESTION: What is the relationship between P and PSPACE?
• An algorithm that uses 30 tape cells must use at least 30 time steps.
• An algorithm that uses 30 tape cells may use many more steps.

\[ P \subseteq \text{PSPACE} \]
Most problems are in NP

BUT...

There are problems in PSPACE for which there is no known NP algorithm!

Game

- 2 Players; they alternate.
- Each says the name of a geographic place.
- Kids play this in the car.

<table>
<thead>
<tr>
<th>Player 1: Portland</th>
<th>Player 2: Denver</th>
<th>Player 1: Rio</th>
</tr>
</thead>
</table>

- There is a list/dictionary of valid words. Each word can only be used once.

The Problem: Given the dictionary, can the 1st player win if he chooses carefully?
And-Or Tree (Min-Max Search)

\[ \exists x_1 \bigwedge x_2 \bigwedge x_3 \bigwedge x_4 \bigwedge x_5 \ldots \]

Non-determinism doesn't seem to help trim the search time.

- Guess a good move for me.
- Check all his possible moves.
- Guess another good move.
- Check all his possible moves.

\[ \vdots \]

A P-space Algorithm

- This is a search of a tree.
- The tree is exponential in size.
  \[ \Rightarrow \text{We cannot store the tree.} \]
- Do a depth-first search of this tree.
- Time taken to search the tree: EXPONENTIAL.
**Graph Isomorphism**

Given two graphs, can you match them up?

![Graphs G1 and G2](image)

**Problem:**
Are 2 graphs isomorphic?

This problem is in NP. Given an answer/correspondence, it can be checked in Polynomial time.

Are 2 graphs **NOT** isomorphic?

This problem is **NOT** in NP. There are $N!$ different possible correspondences. You have to check each of them.

BTW: This problem is **NOT** NP-complete.
\( P \\subseteq NP \\subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE} \)

\( P \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE} \)

\( \text{Det. TM - in Poly time} \)
\( \text{NonDet. TM in Poly time} \)
\( \text{Det/Non Det TM - Poly space} \)
\( \text{Det. TM in exponential time} \)
\( \text{Det. TM - exponential space} \)

**We Know:**