# CSC 320 - Lecture 16

#time-complexity #running-time #nondeterministic-deciders #p #np #deterministic #HAMPATH #satisfiable #boolean #np-complete #PATH #TM #turing-machines #complexity-class

---

## Complexity Relationships Between Different Types of TMs

**Theorem**. Let $t(n)$ be a function, $t(n) \geq n$. Every $t(n)$-time multitape TM has an equivalent $O(t^2(n))$ time single-tape TM.

**Idea**. On single-tape TM, simulate single step on multitape TM.

**Show**. Uses at most $O(t(n))$ steps.

---

**Theorem**. For every multitape Turing Machine there is an equivalent single-tape Turing Machine.

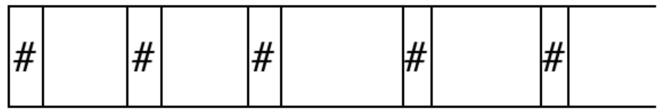
Analyze running time single TM uses in this proof to simulate multitape TM.

---

More formally let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ be multitape TM with $k$ tapes.

- Convert $M$ into single-tape Turing Machine $S = (Q, \Sigma, \Gamma', \delta', q_0, q_{accept}, q_{reject})$ with...
  - $\Gamma' = \Gamma \cup \{\#\} \cup \{\dot{\#}\} \cup \{\dot{a} \mid a \in \Gamma, \dot{a} \notin \Gamma\}, \#, \dot{\#} \notin \Gamma$ .
  - and transition function $\delta'$ for $S$ simulates each move of $M$.
    - for each step of $M : S$ moves from virtual tape to virtual tape. How many steps for one scan?
      - **First Scan**. Search for all current head positions and determine content of current cells on all virtual tapes.
        - Length of active position of tape: $t(n) \in O(t(n))$.
      - **Second Scan**. Update content of current cells and update head positions, according to transition that $M$ executing.
        - $\Rightarrow$ for $t(n)$ many steps, Stakes $O(t^2(n))$

- If a virtual tape head moves right encountering #, make room on virtual tape: all tape content starting at # is shifted by 1 cell to the right and adds a blank for the #.




The first and second scan take $O(t(n))$.

**Note**. $t(n)$-time multitape TM. Assume $t(n) \geq n$.

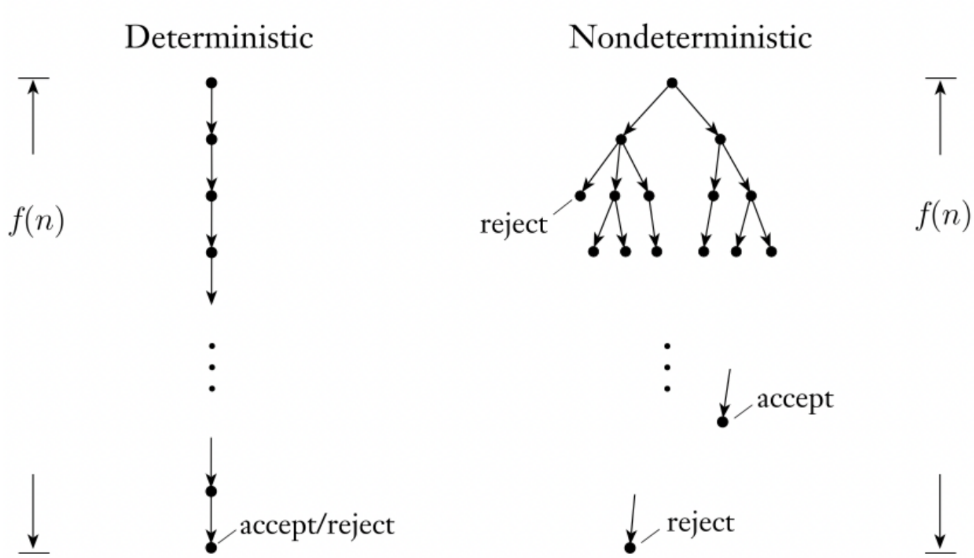
**Question**. How do we talk about time complexity when the TM is a nondeterministic decider?

# Running Time / Time Complexity For Nondeterministic Deciders

**Definition**. Let $N$ be a nondeterministic decider. The **running time** or **time complexity** of $N$ is the function $f : \mathbb{N} \longrightarrow \mathbb{N}$ where $f(n)$ is the maximum number of steps that $N$ uses on any branch of its computation on any input of length $n$.

If $f(n)$ is the running time of $N$ then we say: $N$ **runs in time** $f(n)$ **and** $N$ **is an** $f(n)$**-time nondeterministic TM**.

# Time Complexity Deciders VS Nondeterministic Deciders



**Note**. Running time of deciders is a model for the running time when running algorithms on a classical computer.

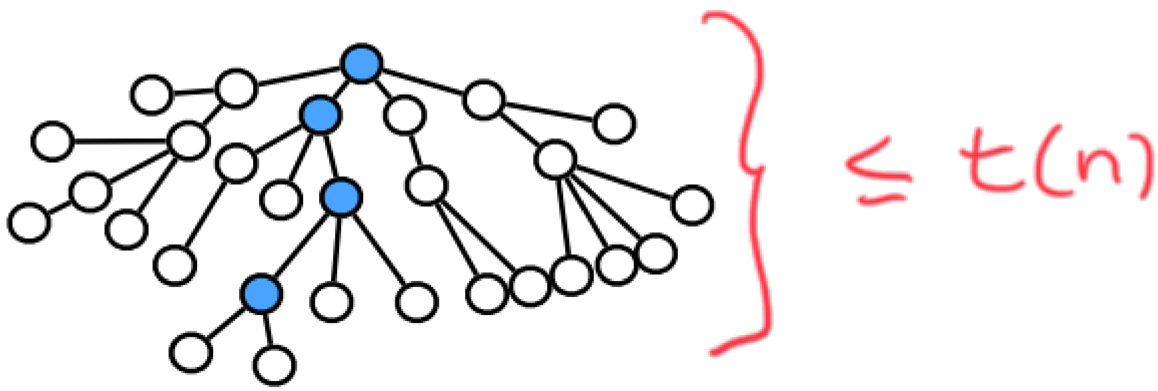
Definition of running time for nondeterministic deciders. Not intended to correspond to real-world computing device. Purely theoretical concepts for characterizing complexity of important classes of computational problems.

# Theorem

Let $t(n)$ be a function, $t(n) \geq n$. Then every $t(n)$-time nondeterministic single-tape TM has an equivalent $2^{O(t(n))}$-time deterministic single-tape TM.

$N : t(n)$-time nondeterministic TM. Assume $t(n) \geq n$. Input length: $n$. Total number of leaves $\leq b^{t(n)}$.

**Note**. $2^{c \cdot t(n)} = 2^{O(t(n))}$.




# Analysis of Simulation

- Simulation in BFS manner...
  - first visit root
  - visit all nodes at depth $d$ before any nodes at depth $d+1$.
- Tree size: number of nodes $< 2$. number of leaves $\Rightarrow$ number of nodes $O(b^{t(n)})$)
- Time to travel from root down to a node: $O(t(n))$
- $\Rightarrow D$'s running time: $O(t(n)) \cdot b^{t(n)} = O(t(n)) \cdot 2^{O(t(n))}$.

$$b^{t(n)} = 2^{log_2(b^{t(n)})} = 2^{(log_2 b)t(n)}$$

$$\begin{aligned} O(t(n)) \cdot b^{t(n)} &= O(t(n)) \cdot 2^{O(t(n))} \\ &= 0(2^{log_2 t(n)} \cdot 2^{0(t(n))}) \\ &= 2^{O(t(n))} \end{aligned}$$

- TM $D$ has three tapes; conversion into single-tape TM.

$$\begin{aligned} O(2^{O(t(n))})^2 &= O(2^{O(2\cdot t(n))}) \\ &= O(2^{O(t(n))}) \end{aligned}$$

# Complexity Class $P$

$$P = \bigcup_k TIME(n^k)$$

i.e., $P$ is the class of languages that are decidable in polynomial time on a deterministic single-tape TM.

- $P$: often considered that class of problems solvable - on a classical computer - in practice.
- Not entirely accurate but many problems in $P$ indeed solvable in practice.
- If a problem $A$ is in $P$ then there exists an $n^c$-time algorithm for $A$, for some constant $c$.

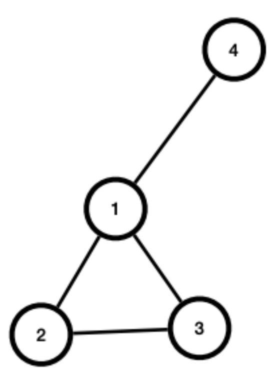
# Problems in $P$ - $(O(n^3))$

**Note**. We are not claiming here right now any efficient algorithms. We just want to show membership in $P$.

## Example 1

$A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$

$\langle G \rangle = (1,2,3,4)((1,2),(2,3),(1,3),(1,4))$




- $M =$ "On input $\langle G \rangle$:
    1. Select first node of $G$; mark it **(1 step)**
    2. Repeat following step until no new nodes are marked ($\leq n$ **times**)

    - For each node $v$ in $G$: mark $v$ if incident to an edge where the other endpoint is already marked. ($\leq n^2$ **edges**)
3. Scan all nodes of $G$: determine whether or not they all are marked ($n$ **steps**)
    - If they are: accept
    - Otherwise: reject."

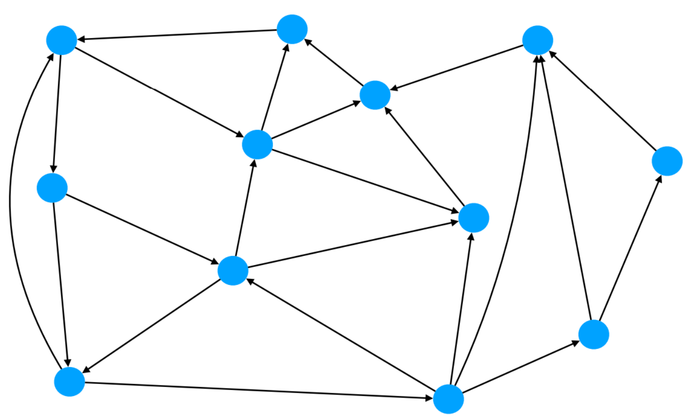
**Note**. We do not have to worry whether this is a single tape Turing Machine or a Multi-Tape Turing Machine.

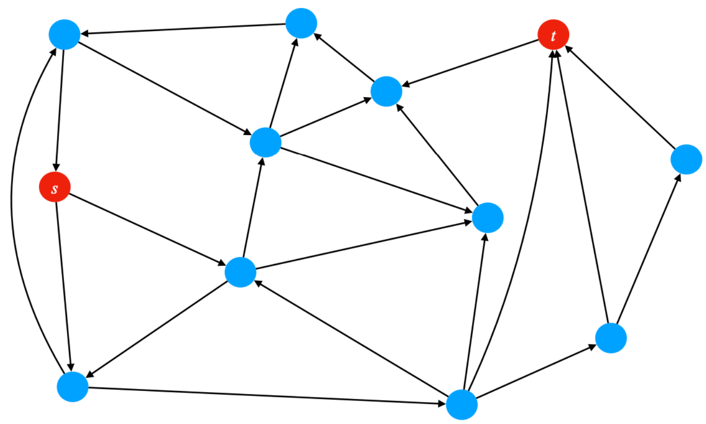
## Example 2

$PATH$

- **Input**. A directed graph $G = (V, A)$ and vertices $s, t \in V$.
- **Question**. Does there exist a directed path from $s$ to $t$ in $G$?
- We assume that $|V| = n$.

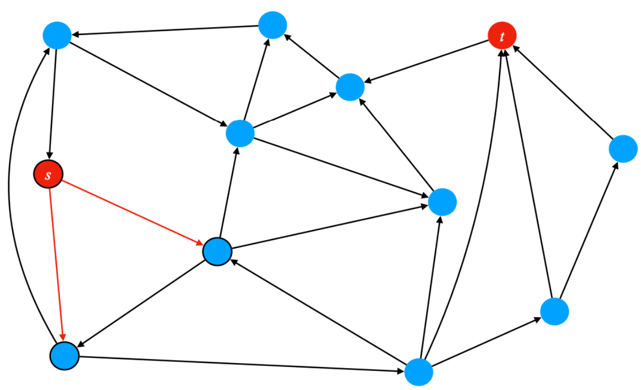
$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from} s \text{ to } t\}.$

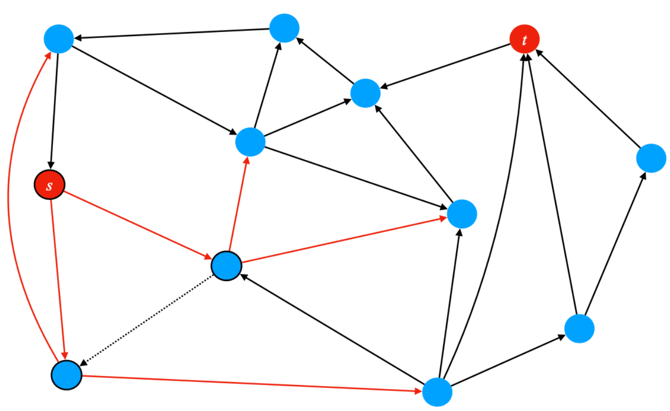


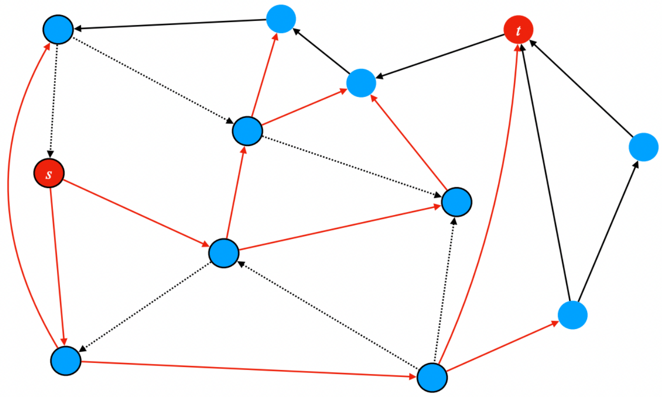



$$PATH \in P$$

- Observe that a brute-force algorithm for this problem is too slow:
  - Examine all potential paths in $G$, i.e., sequences of nodes from $V$ of length at most $n$.
    - **Note**. If any directed path exists from $s$ to $t$, then there is one of length at most $n$.
  - Check wether any potential path is a directed path from $s$ to $t$.
  - But the number of such potential paths is roughly $n^2$ (exponential in number of nodes in $G$).
  - Brute-force algorithm uses exponential times.
- **Idea**. Polynomial time algorithm for $PATH$.
  - Use breadth-first search
  - Mark all nodes in $G$ that are reachable from $s$ by directed paths of length 1
  - Mark all nodes in $G$ that are reachable from $s$ by directed paths of length 2
  - Mark all nodes in $G$ that are reachable from $s$ by directed paths of length 3
  - Mark all nodes in $G$ that are reachable from $s$ by directed paths of length $n$
  - Check whether or not $t$ was discovered.






## Running Time

- Polynomial time algorithm / decider $M$ for PATH:
  - $M =$ "On input $\langle G, s, t\rangle$, where $G$ is a directed graph with nodes $s$ and $t$:
    - Place a mark on node $s$ ($O(1)$)
    - Repeat until no additional nodes are marked: ($O(n)$)
      - Scan arcs of $G$: If arc $(a, b)$ is found where $a$ is marked and $b$ is unmarked then mark $b$ ($O(m) = O(n^2)$)
    - If $t$ is marked accept, else, reject" ($O(1)$)

Running time is $O(n^3)$.

# Class $P$

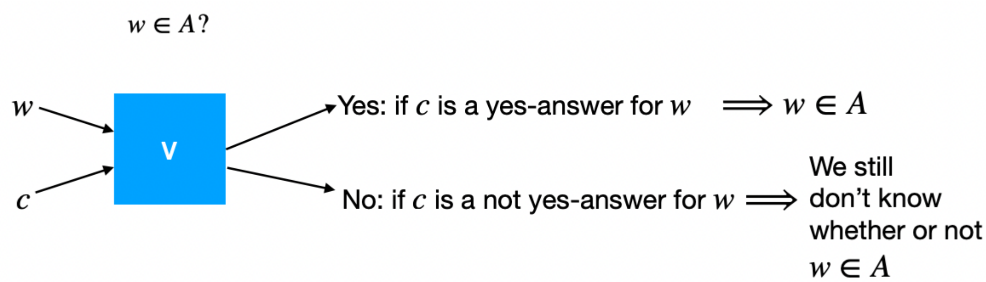
- $M$ be a (deterministic) decider.
  - **Running time** or **time complexity** of $M$: function $f : \mathbb{N} \longrightarrow \mathbb{N}$, where $f(n)$ is maximum number of steps that $M$ uses on any input of length $n$.
  - If $f(n)$ is the running time of $M$, we say that $M$ **runs in time** $f(n)$ and that $M$ is an $f(n)$**-time TM**.
- $t : \mathbb{N} \longrightarrow \mathbb{R}^+$ be a function.
  - **Time complexity class** $TIME(t(n))$: collection of all languages decidable by an $O(t(n))$-time TM.
- $t(n)$ be a function, $t(n) \geq n$.
  - Every $t(n)$-time multitape TM has equivalent $O(t^2(n))$-time single-tape TM.
- $P$: **class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine**.

# Class $NP$

- $N$ nondeterministic decider.

- **Running time** of $N$: function $f : \mathbb{N} \longrightarrow \mathbb{N}$, where $f(n)$ is maximum number of steps that $N$ uses on any branch of its computation on any input of length $n$.
- $t : \mathbb{N} \longrightarrow \mathbb{R}^+$, $t(n) \geq n$.
  - Every $t(n)$-time nondeterministic single-tape TM has an equivalent $2^{O(t(n))}$-time deterministic single-tape TM.
- Let $A$ be a language. A **verifier** for $A$ is an algorithm $V$ with...
  - $A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$.
- Verifier $V$ uses **certificates** $c$ to verify $w \in A$.

$NP$, class of problems solvable in **nondeterministic polynomial time**, is the class of languages that have polynomial time verifiers.




$NTIME(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n))$
$\text{-time nondeterministic Turing machine}\}$

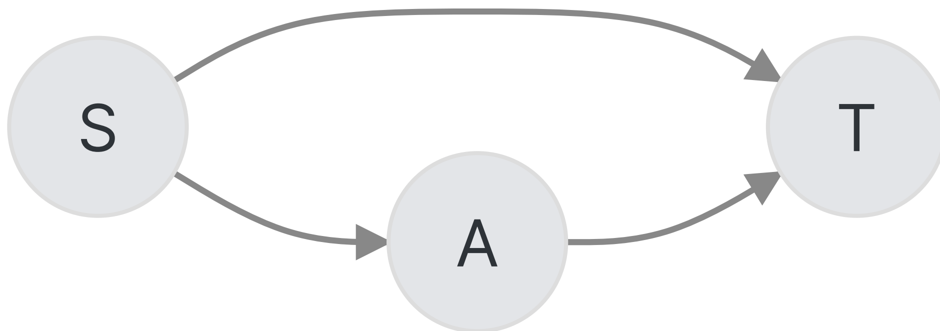
$$NP = \bigcup_k NTIME(n^k)$$

# Terminology

- Let $V$ be a verifier...
  - Running time of $V$ measured in terms of length of $w$ ($\langle w, c \rangle$) only
  - a **polynomial-time verifier** runs in polynomial time in length of $w$.
- Language $A$ is **polynomially verifiable** if it has a polynomial-time verifier $V$.
- **Note**. If $V$ polynomial verifier $V$ then certificate $c$ has polynomial length in terms of length of $w$.

# Example

Let $G$ be a directed graph... A **Hamiltonian path** in $G$ is a directed path that visits each node exactly once.




Where (S, A) -> (A, T) is a Hamiltonian Path.

## $HAMPATH$:

- **Input**. Directed graph $G = (V, A)$, vertices $s, t \in V$.
- **Question**. Does there exist a Hamiltonian path from $s$ to $t$ in $G$?

Language $HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$.

Does $HAMPATH$ have a polynomial verifier $V$?

- Certificate for $HAMPATH$?
  - To verify $\langle G, s, t\rangle \in HAMPATH$: certificate $c$ must be Hamiltonian path from $s$ to $t$.
- $V$ takes as input $\langle G, s, t\rangle$ and $c$ and tests whether $c$ is a Hamiltonian path from $s$ to $t$ in $G$.

## $HAMPATH \in NP$

Nondeterministic TM that decides $HAMPATH$ in nondeterministic polynomial time:

- $N_1 =$ "On input $\langle G, s, t\rangle$, where $G$ is a directed graph with nodes $s$ and $t$
  - Write a list of $n$ numbers $p_1, \ldots, p_n$, where $n$ is the number of nodes in $G$
    - Each number is nondeterministically selected to be between 1 and $n$ (Certificate)
  - Check for repetitions in list. If any are found, rejects.
  - Check whether $s = p_1$ and $t = p_n$. If either fail, reject.
  - For each $i$ between 1 and $n - 1$, check whether $(p_i, p_{i+1})$ is an arc of $G$. If not, reject.
  - If all tests have been passed, accept."

The verifier is the last 4 points listed above. The verifier runs in polynomial time.

# Theorem

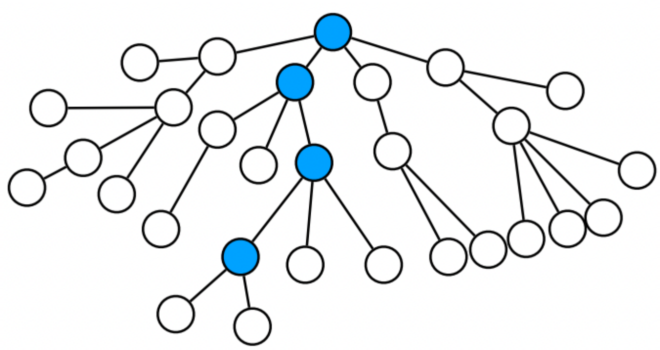
A language $A$ is in $NP$ if and only if $A$ is decided by some nondeterministic polynomial-time Turing machine.

**Idea**. We convert a polynomial time verifier $V$ to an equivalent nondeterministic polynomial-time Turing machine $N$ and vice versa.

- $N$ simulates $V$ by guessing the certificate.
- $V$ simulates $N$ by using accepting branch as certificate.

**Proof**. Converting a polynomial time verifier $V$ to an equivalent nondeterministic polynomial-time Turing machine $N$:

- Let $A \in NP$ and show that $A$ is decided by a nondeterministic polynomial-time TM $N$.
  - Let $V$ be a polynomial time verifier for $A$ that exists by definition of $NP$.
  - Assume that $V$ is a TM that runs in time $n^k$ and construct $N$ as follows...
    - $N =$ "On input $w$ of length $n$: Nondeterministically select string $c$ of length at most $n^k$.
    - Run $V$ on input $\langle w, c \rangle$
    - If $V$ accepts, accept; otherwise, reject".



**Proof**. Assume $A$ is decided by polynomial-time nondeterministic TM $N$ (Other Direction):

- Construct polynomial-time verifier $V$ as follows...
  - $V =$ "On input $\langle w, c \rangle$, where $w$ and $c$ are strings:
    - Simulate $N$ on input $w$; each symbol of $c$ describes nondeterministic choice made at each step
    - If this branch of $N$'s computation accepts, accept; otherwise, reject".

# Examples of Problems / Languages in $NP$

- PATH - note that we already know that PATH is also in P

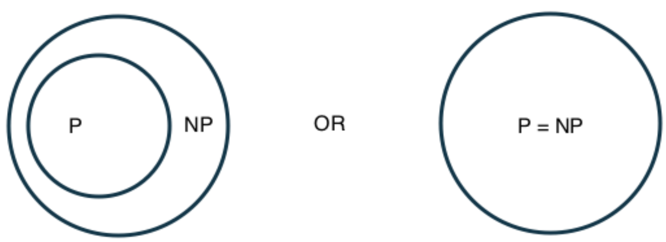
- HAMPATH
- Clique: $CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with } k\text{-clique}\}$
- Independent Set: $IS = \{\langle G, k\rangle \mid G$
  $\text{is an undirected graph with an independent set of size at least } k\}$
- Vertex Cover: $VC = \{\langle G, k\rangle \mid G$
  $\text{is an undirected graph with a vertex cover set of size at least } k\}$
- Subset Sum: $SUBSET - SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}$
  $\text{, and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\} \text{ , we have } \Sigma_{y_i} = t\}$

# The P VS NP Question

- **We Know**.
  - $P \subseteq NP$

$$NP \subseteq \bigcup_k TIME(2^{n^k})$$

- What about $NP \subseteq P$?




# NP-Completeness

1970s: Stephen Cook (UofT) and Leonid Levin (MIT) discovered problems in NP where individual complexity is related to that of entire class.

If polynomial-time algorithm exists for any such problem: all problems in NP would be polynomial time solvable.

Problems called **NP-Complete**. NP-Completeness important for both theory and practice.

# Satisfiability

First problem to be shown NP-complete. $SAT = \{\langle \Phi \rangle \mid \Phi \text{ is a satisfiable Boolean formula}\}$.

# Terminology

**Boolean Variables**. Variable that can take on the values TRUE and FALSE. Usually, represent TRUE by 1 and FALSE by 0.

**Boolean Operations**. AND, OR, NOT. Represented by symbols $\wedge, \vee$, and $\neg$.

**Boolean Formulas**. Expression involving Boolean variables and operators.
$\phi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$.

**Note**. The over bar is shorthand for $\neg$ : $\bar{x}$ means $\neg x$.

# Boolean Operations and Formulas

- AND
  - $0 \wedge 0 = 0$
  - $0 \wedge 1 = 0$
  - $1 \wedge 0 = 0$
  - $1 \wedge 1 = 1$
- OR
  - $0 \vee 0 = 0$
  - $0 \vee 1 = 1$
  - $1 \vee 0 = 1$
  - $1 \vee 1 = 1$
- NOT
  - $\bar{0} = 1$
  - $\bar{1} = 0$

A Boolean formula is **satisfiable** if some assignment of 0's and 1's to the variables makes the formula evaluate to 1.

Formula $\phi$ is satisfiable because assignment $x = 0$, $y = 1$, and $z = 0$ makes $\phi$ evaluate to 1.

We say the assignment **satisfies** $\phi$.

The **satisfiability problem (SAT)**: test whether or not a Boolean formula is satisfiable.

---

# Previous Lecture

Lecture15

# Next Lecture

Lecture17