NP-complete problems = "computationally difficult"
⇒ Don't expect a poly-time algorithm

Undecidable problems = impossible to construct an algorithm that's correct or every input, regardless of running time.

1936 - Alan Turing showed that the Halting Problem is undecidable.

Related to Gödel's incompleteness theorem (1931):
   Roughly, in every mathematical system there is a true but unprovable statement.

**Halting Problem**:
- **Input**: a program \( P \) (written in any language) with an input \( I \)
- **Output**: TRUE if \( P(I) \) ever terminates
  FALSE if \( P(I) \) never terminates (i.e., has an infinite loop)
Theorem: the Halting Problem is undecidable.

Proof: by contradiction.

Suppose we had a program that solves the Halting Problem on every input.

Call this program \textsc{Terminates}(P).

Consider the following evil program:

\textsc{Paradox}(Z):

\begin{enumerate}
  \item If \textsc{Terminates}(Z, Z) \\
    \text{Then goto (1)} \\
    \text{Else return}
\end{enumerate}
What does PARADOX do?

a) if PROGRAM Z on input Z terminates
   then PARADOX(Z) goes into an infinite loop
   & it never terminates

b) if Program Z or Z never terminates
   then PARADOX(Z) terminates.

Let Z = PARADOX.

Then does PARADOX(PARADOX) terminate?

Case a) if PARADOX on PARADOX terminates
   then PARADOX(PARADOX) never terminates

b) if PARADOX on PARADOX never terminates
   then PARADOX(PARADOX) terminates

Either way we get a contradiction.

So our assumption that TERMINATES() exists must be impossible.
Circuit-SAT:

**input:** DAG (directed acyclic graph) representing a circuit where vertices are gates: AND, NOT, OR, known input, unknown input

AND, OR have indegree 2
NOT has indegree 1
known input have no indegree and are labeled TRUE or FALSE
unknown are labeled ?

One sink is labeled output

**output:**

an assignment to the unknown gates so that the circuit evaluates to T if such an assignment exists.

No otherwise.
Example:

\[ \begin{array}{c}
\text{OUTPUT} \\
\text{OR} \\
\text{AND} \\
\text{OR} \\
\text{NOT} \\
\text{AND} \\
\text{OR} \\
\text{AND} \\
\end{array} \]

\[ \begin{array}{c}
? \\
? \\
T \\
? \\
? \\
? \\
\end{array} \]

Setting \( x_1 = T, x_2 = F, x_4 = F, x_5 = F \) it outputs T.

SAT is a special case of CIRCUIT-SAT. It's circuit has a bunch of OR's at the bottom for the clauses, and then AND's at the top for the between clause constraints.
Theorem: CIRCUIT-SAT is NP-complete

Proof:

a) CIRCUIT-SAT ∈ NP:
   Given a circuit & an assignment, easy to evaluate the circuit & check if outputs T.

b) Need to show for every A ∈ NP, A → CIRCUIT-SAT.
   Know A is a search problem (because it's in NP).
   So there is an algorithm C ("checker") that takes input I & solution S and checks that S is a solution to I.

Any algorithm can be represented by a circuit - the (unknown) input gates are used to take I & S as input.
Need to solve A using Circuit-SAT.
Take input I for A.
Want to know if there exists a solution S for I on A.

Create a circuit C_I that mimics the algorithm C on input I with unknown S.
If we run an alg. to solve Circuit-SAT on C_I then it will tell us if there is an assignment to the inputs corresponding to S so that I has a solution.
Thus, if C_I has a solution that gives us a solution S to I.
& if there is NO solution then I has NO solution.
**Theorem:** SAT is NP-complete

**Proof:**

a) SAT \in NP:

Given input \( f \) and assignment to \( x_1, \ldots, x_n \).

in \( O(nm) \) time can check that every clause is satisfied.

b) CIRCUIT-SAT \rightarrow SAT.

Given a circuit \( C \),

the variables in our formula \( f \) are the input variables & for every gate add a new variable \( g \).

If \( g \) is known input and set to \( T \), add clause \( \neg g \)

If \( g \) is an OR of \( h_1 \) & \( h_2 \)

\[ g \leftarrow \text{OR} \]
\[ h_1 \quad h_2 \]

\[ \text{add}: \ (g \lor h_2) \land (g \lor h_1) \land (\neg g \lor h_1, \lor h_2) \]

\[ \text{to satisfy}: \ g = T \iff h_1 = T \lor h_2 = T \]
\[ h_1 = T \Rightarrow g = T \]
\[ h_2 = T \Rightarrow g = T \]
\[ h_1 = T \land h_2 = T \Rightarrow g = F \]
\[(\overline{gvh}_1) \land (\overline{gvh}_2) \land (gvh, \overline{vh}_2)\]

to satisfy: \(g = T\) if \(h_1 = T, h_2 = T\)
\(\Rightarrow\) if \(h_1 = F \Rightarrow g = F\)
\(\Rightarrow\) if \(h_2 = F \Rightarrow g = F\)
\(\Rightarrow\) if \(h_1 = h_2 = T \Rightarrow g = T\)

\[(\overline{gvh}) \land (\overline{gvh})\]

to satisfy: \(g = T\)
\(\Rightarrow\) if \(h = T \Rightarrow g = F\)
\(\Rightarrow\) if \(h = F \Rightarrow g = T\)