### Algorithmically unsolvable problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulate problem</td>
<td>∞</td>
</tr>
<tr>
<td>Halting problem</td>
<td>∞</td>
</tr>
<tr>
<td>Program correctness</td>
<td>∞</td>
</tr>
<tr>
<td>Program equivalence</td>
<td>∞</td>
</tr>
<tr>
<td>Integral roots of a polynomial</td>
<td>∞</td>
</tr>
<tr>
<td>Goodstein’s theorem</td>
<td>∞</td>
</tr>
<tr>
<td>Generalized $(3n + 1)$ problem</td>
<td>∞</td>
</tr>
<tr>
<td>Post correspondence problem</td>
<td>∞</td>
</tr>
<tr>
<td>Problem</td>
<td>Running time</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>Search in a sorted array</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Search in an unsorted array</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Integer addition</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Generate primes</td>
<td>$O(n \log \log n)$</td>
</tr>
<tr>
<td>Sorting</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Fast Fourier transform</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Integer multiplication</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>Linear programming</td>
<td>$O(n^{3.5})$</td>
</tr>
<tr>
<td>Primality test</td>
<td>$O(n^{10})$</td>
</tr>
<tr>
<td>Satisfiability problem</td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td>Traveling salesperson problem</td>
<td>$O((n - 1)!)$</td>
</tr>
<tr>
<td>Sudoku, chess, checkers, go</td>
<td>expo. class</td>
</tr>
</tbody>
</table>
Polynomial and exponential functions

\[ f(n) = n! \cdot 2^n \cdot n^2 \cdot n\log(n) \cdot \sqrt{n} \cdot \log(n) \]
Exponential functions

- Moore’s law (Doubling of computing power every 18 months)
- Compound interest in banks
- Coronavirus

Source: https://www.worldometers.info/coronavirus/
## Goal

- Our goal is to solve all computational problems efficiently
- An **efficient/fast** algorithm is one that solves a problem in polynomial time
## Polynomial-time algorithm

### Definition

- A **polynomial-time** algorithm is an algorithm whose worst-case time complexity is bounded above by a polynomial function in input size.
- If $n$ is the input size, then there exists a polynomial $p(n)$ such that

$$T(n) \in \mathcal{O}(p(n))$$

### Analysis

- $n \log n$ is not polynomial in $n$ but $n \log n \in \mathcal{O}(n^2)$
  Hence, algorithm with this complexity is a polynomial-time algorithm
### Definition

- For a given algorithm, the input and output sizes are defined as the **number of characters** required to write/encode/specify the input and output, respectively, using a reasonable encoding method.
- Reasonable encodings: base 2, base 16, base 10, base $b \geq 2$
- Unreasonable encoding: base 1 (i.e., unary encoding)

### Example

- **Problem:** $\text{SORT}(a[1..n])$
  - Input: $n$ positive integers
  - Output: $n$ numbers in nondecreasing order. Then
  - Suppose $L = \max(a[1..n])$
  - Input size: $\Theta(n \log L)$
  - Output size: $\Theta(n \log L)$
### Input and output sizes

<table>
<thead>
<tr>
<th><strong>IsPrime</strong>(<em>n</em>)</th>
</tr>
</thead>
</table>
| 1. *answer* ← *true*
| 2. for *i* ← 2 to ⌊√*n*⌋ do
| 3. if *n* is divisible by *i* then
| 4. *answer* ← *false*
| 5. break
| 6. return *answer*

### Problem

- Time complexity of **IsPrime**(*n*) is Ω (√*n*).

  Is **IsPrime**(*n*) a polynomial-time algorithm?

### Solution

- **No!**
- Input size: *s* = log₂ *n* bits (to store value *n*)
- Output size: 1 bit (to store Boolean answer)
- Time complexity: Ω (√*n*) = Ω (2<sup>s/2</sup>) exponential

  But this does not prove that the problem cannot have any fast algorithm.
### Problem
- Time complexity of \texttt{FIBONACCI-DP}(n) is $\Theta(n^2)$. Is \texttt{FIBONACCI-DP}(n) a polynomial-time algorithm?

### Solution
- No!
- Input size: $s = \log_2 n$ bits (to store value $n$)
- Output size: $\Theta(n)$ bits (as $F_n$ requires $\Theta(n)$ bits)
- Time complexity: $\Theta(n^2) = \Omega(4^s)$ exponential

There cannot be any polynomial-time algorithm for computing the $n$th Fibonacci number. Why?
- Output size itself is exponential in the size of input.
## Problem having exponential-sized output

### Problem
- **Problem:** Print all simple paths
- **Input:** Graph $G$, source vertex $x$, destination vertex $y$
- **Output:** Print all simple paths from $x$ to $y$

### Analysis
- **Output size:** Worst-case exponential function of the input size
  
  Hence, polynomial-time algorithms don’t exist
- **We will not consider problems having exponential-sized output**
  because no polynomial-time algorithms exist for such problems
### Intractable problems

<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A problem is <strong>intractable</strong> if an exponential amount of time is needed to discover its solution, given that the output size a polynomial function of the input size.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| Problem: Equivalence of two regular expressions  
Input: Two regular expressions $R_1$ and $R_2$  
Output: Yes/no if $R_1$ is equivalent $R_2$ |

<table>
<thead>
<tr>
<th>Analysis</th>
</tr>
</thead>
</table>
| Output size: Polynomial function of input size  
There does not exist polynomial-time algorithms |
# Types of problems

## Definitions

- A **decision problem** asks for a yes/no answer.
- A **search problem** asks for arbitrary string(s) as output.
- A **counting problem** asks for the number of solutions to a search problem.
- An **optimization problem** asks for the best possible solution to a search problem.
- A **function problem** asks for a unique output for every input.

## Examples

- Decision problem:  \texttt{ISPRIME}(n)
- Search problem:  \texttt{FINDFACTORS}(n)
- Counting problem:  \texttt{COUNTFACTORS}(n)
- Optimization problem:  \texttt{TSP}(G, w)
- Function problem:  \texttt{TSP}(G, w)
# Hardness of problems

## Types

- **Hard (or intractable):** Problems that can never be solved in polynomial time.
- **Easy:** Problems that can be solved in polynomial time.
- **Possibly hard (or possibly intractable):** Problems that have no known polynomial time algorithms.

## Examples

- **Hard:** Given two regular expressions $R_1$ and $R_2$, is $R_1$ equivalent to $R_2$?
- **Easy:** Is there a path from $x$ to $y$ with weight $\leq M$?
- **Possibly hard:** Is there a path from $x$ to $y$ with weight $\geq M$?
The complexity class P denotes the set of all decision problems that can be solved by deterministic algorithms in polynomial time.

### Examples

- Is a given array sorted?
- Is a given graph cyclic?
- Is a given graph connected?
- Does a given set contain a specific element?
- Most problems we have seen have a corresponding decision version.
# Complexity class NP

<table>
<thead>
<tr>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>- The complexity class <strong>NP</strong> denotes the set of all decision problems that can be solved by nondeterministic algorithms in polynomial time.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>- All problems in P, i.e., P ⊆ NP</td>
</tr>
<tr>
<td>- Problem: Decision version of TSP((G, w, b))</td>
</tr>
<tr>
<td>Input: Graph (G), weight function (w), length (b)</td>
</tr>
<tr>
<td>Output: Yes if there exists a sequence of vertices (starting from a vertex and visiting each vertex exactly once) with length at most (b).</td>
</tr>
</tbody>
</table>
Analysis

- A nondeterministic algorithm has two stages:
  1. (Nondeterministic) Guessing stage:
     Make all guesses simultaneously.
     (Analogous to parallel algorithm or parallel universe model)
  2. (Deterministic) Verification stage:
     Verify/check if the guess is a correct solution or not.
- Guessing stage takes $O(1)$ time
- Verification stage takes polynomial time
Complexity class NP

Definition

- A **polynomial-time nondeterministic algorithm** is a nondeterministic algorithm whose verification stage is a polynomial-time algorithm.
- The complexity class **NP** denotes the set of all decision problems that can be solved by polynomial-time nondeterministic algorithms.

Source: Jeff Erickson’s Algorithms textbook
The complexity class **NP** denotes the set of all decision problems with the following property: If the answer is yes, then there is a proof of this fact that can be checked in polynomial time.

Intuitively, the complexity class **NP** denotes the set of all decision problems where we can verify a yes answer quickly if we have the solution in front of us.

The complexity class **co-NP** denotes the set of all decision problems with the following property: If the answer is no, then there is a proof of this fact that can be checked in polynomial time.
<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is $P = NP$?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is the greatest question in theoretical computer science.</td>
</tr>
<tr>
<td>That is:</td>
</tr>
<tr>
<td>Nobody knows if (deterministic) polynomial-time algorithms exist for solving all of NP problems.</td>
</tr>
<tr>
<td>Nobody knows if there is an NP problem that is not in P.</td>
</tr>
<tr>
<td>Nobody knows if NP is the same set as coNP.</td>
</tr>
<tr>
<td>Most scientists believe that $P \neq NP$.</td>
</tr>
<tr>
<td>It is most likely that Turing Award (i.e., the Nobel prize of computer science) will be given to the person who resolves the $P \neq NP$ problem.</td>
</tr>
</tbody>
</table>
Complexity classes NP-hard and NP-Complete

**Definition**

- **NP** = Problems solvable in poly time using **nondeterminism**
  = Problems with solutions that can be verified/checked in polynomial time.
- **NP-Hard** = Problems at least as hard as NP problems.
  Formally, a problem $X$ is NP-Hard if every NP problem $Y$ is polynomial-time reducible to $X$.
- **NP-Complete** = Hardest problems in NP.
  Formally, a problem $X$ is NP-Complete if (i) $X$ is in NP, and (ii) $X$ is NP-Hard.

Source: Jeff Erickson’s Algorithms textbook
Complexity classes NP-hard and NP-Complete

Less time

NP

NP-Complete

P

NP-Hard

EXP

EXP-Hard

EXP-Complete

More time
### Easy problems and possibly hard problems

<table>
<thead>
<tr>
<th>Easy problems</th>
<th>Possibly hard problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest path</td>
<td>Longest path</td>
</tr>
<tr>
<td>Linear programming</td>
<td>Integer linear programming</td>
</tr>
<tr>
<td>Minimum spanning tree</td>
<td>Traveling salesperson</td>
</tr>
<tr>
<td>2-Satisfiability</td>
<td>3-Satisfiability</td>
</tr>
<tr>
<td>Min cut</td>
<td>Max cut</td>
</tr>
<tr>
<td>Planar 4-colorability</td>
<td>Planar 3-colorability</td>
</tr>
<tr>
<td>Independent set on trees</td>
<td>Independent set</td>
</tr>
</tbody>
</table>

- The problems on the right have escaped efficient algorithms for decades to centuries. Why?
- The problems on the right seem hard for the same reason – they are all related.
- Each pair of those problems can be reduced to each other.
What is polynomial-time reduction?

Definition

- Reduction is a fantastic idea to solve one problem using the solution to another.
- **Problem** $P_{\text{old}}$ poly.-time reduces to problem $P_{\text{new}}$, denoted by $P_{\text{old}} \leq_p P_{\text{new}}$, if the following transformation happens in polynomial time.
  - transform any input instance of $P_{\text{old}}$ to an instance of $P_{\text{new}}$
  - solve $P_{\text{new}}$
  - transform output of $P_{\text{new}}$ to output of $P_{\text{old}}$
  - return output of $P_{\text{old}}$
What is polynomial-time reduction?

### Definition

- Reduction is a fantastic idea to solve one problem using the solution to another.
- **Problem** $P_{old}$ poly.-time reduces to problem $P_{new}$, denoted by $P_{old} \leq_p P_{new}$, if any instance of problem $P_{old}$ can be solved using the following:
  1. **poly. number of standard computational steps.**
  2. **poly. number of calls to function that solves problem $P_{new}$.**
- $P_{old} \leq_p P_{new}$ means $P_{new}$ is at least as hard as $P_{old}$. 
What is polynomial-time reduction?

Problem-old (input-old) \( \leq_p \) P_new

1. input-new \( \leftarrow f \)(input-old) \( \triangleright \) poly. time transformation
2. output-new \( \leftarrow \text{PROBLEM-NEW}(\text{input-new}) \)
3. output-old \( \leftarrow g \)(output-new) \( \triangleright \) poly. time transformation
4. return output-old

Problem-old poly. time reduces to Problem-new
What is polynomial-time reduction?

Suppose $P_{\text{old}} \leq_p P_{\text{new}}$

<table>
<thead>
<tr>
<th>Easy problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>• If $P_{\text{new}}$ can be solved in polynomial time,</td>
</tr>
<tr>
<td>then $P_{\text{old}}$ can be solved in polynomial time.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hard problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>• If $P_{\text{old}}$ cannot be solved in polynomial time,</td>
</tr>
<tr>
<td>then $P_{\text{new}}$ cannot be solved in polynomial time.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Same complexity class</th>
</tr>
</thead>
<tbody>
<tr>
<td>• If $P_{\text{new}} \leq_p P_{\text{old}}$, then</td>
</tr>
<tr>
<td>$P_{\text{old}}$ can be solved in polynomial time</td>
</tr>
<tr>
<td>if and only if $P_{\text{new}}$ can be solved in polynomial time.</td>
</tr>
</tbody>
</table>
### Problem

- **Problem: Least common multiple (LCM)**
  - **Input:** Two integers $a$ and $b$.
  - **Output:** Return the smallest integer $m$ such that $m$ is a multiple of $a$ and $m$ is also a multiple of $b$.

- **Problem: Greatest common divisor (GCD)**
  - **Input:** Two integers $a$ and $b$.
  - **Output:** Return the largest integer $d$ such that $d$ divides $a$ and $d$ divides $b$. 
Reduction: \( \text{LCM} \rightarrow \text{GCD} \)

\[
\text{LCM}(a, b) = \frac{a \times b}{\text{GCD}(a, b)}
\]

GCD is poly. time \(\Rightarrow\) LCM is poly. time
Problem

- Problem: Arithmetic operations on decimal numbers
  Input: Two decimal numbers $a$ and $b$.
  Output: Return the result of an arithmetic operation on $a$ and $b$ in the decimal system.

Problem

- Problem: Arithmetic operations on binary numbers
  Input: Two binary numbers $a$ and $b$.
  Output: Return the result of an arithmetic operation on $a$ and $b$ in the binary system.
Reduction: DecimalCalculator $\rightarrow$ BinaryCalculator

**DecimalCalculator**($a, b$)

1. $a_{\text{binary}} \leftarrow \text{DecimalToBinary}(a)$
2. $b_{\text{binary}} \leftarrow \text{DecimalToBinary}(b)$
3. $c_{\text{binary}} \leftarrow \text{BinaryCalculator}(a_{\text{binary}}, b_{\text{binary}})$
4. $c \leftarrow \text{BinaryToDecimal}(c_{\text{binary}})$
5. return $c$

**BinaryCalculator** is poly. time  
⇒ **DecimalCalculator** is poly. time
Problem

Problem: Closest pair

Input: A set $S$ of $n$ numbers, and threshold $t$.

Output: Is there a pair $s_i, s_j \in S$ such that $|s_i - s_j| \leq t$?

`ClosestPair(S, t)`

1. `Sort(S)`
2. `return (\min_{i \in [1, n-1]} |s_i - s_j|) \leq t`

**Sort** is poly. time $\Rightarrow$ **ClosestPair** is poly. time
Problem

- Problem: Longest increasing subsequence
  Input: An integer or character sequence $S$.
  Output: What is the longest sequence of integer positions $\{p_1, \ldots, p_m\}$ such that $p_i < p_{i+1}$ and $S_{p_i} < S_{p_{i+1}}$?

Problem

- Problem: Longest common subsequence
  Input: Integer or character sequences $S$ and $T$.
  Output: What is the longest subsequence that is common to both $S$ to $T$?
Reduction: LIS $\rightarrow$ LCS

<table>
<thead>
<tr>
<th>LIS($S$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $T \leftarrow \text{Sort}(S)$</td>
</tr>
<tr>
<td>2. $lis \leftarrow \text{LCS}(S, T)$</td>
</tr>
<tr>
<td>3. return $lis$</td>
</tr>
</tbody>
</table>

LCS is poly. time
$\Rightarrow$ LIS is poly. time
Is $P = NP$?

Source: https://en.wikipedia.org/wiki/P_versus_NP_problem
NP-Completeness

If any NP-Hard problem is solvable in poly-time, then every NP problem (1000s of them) is solvable in poly-time.
If any NP-Complete problem cannot be solved in poly-time, then every NP-hard problem (1000s of them) cannot be solved in poly-time.
Problem: Satisfiability (SAT)

- Given a Boolean formula (or logical expression) in conjunctive normal form (CNF), find either a satisfying truth assignment or report that none exists.

- Examples.
  1. \((x \lor y \lor z) \land (x \lor \overline{y}) \land (y \lor \overline{z}) \land (z \lor \overline{x}) \land (\overline{y} \lor \overline{y} \lor \overline{z})\)
    No solution exists.
  2. \((\overline{x_1} \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_4)\)
    Solution is \((x_1, x_2, x_3, x_4) = (T, T, F, F)\)

- Applications.
  Circuit design, image analysis, software engineering, artificial intelligence, and automatic theorem proving
### Problem: $k$-Satisfiability ($k$-SAT)

- **The $k$-SAT problem** is a restricted version of the SAT problem, in which each clause has at most $k$ literals.

- **Examples of 3-SAT.**
  1. $(i) \ (x \lor y \lor z) \land (x \lor \overline{y}) \land (y \lor \overline{z}) \land (z \lor \overline{x}) \land (\overline{y} \lor \overline{y} \lor \overline{z})$

     No solution exists.

  2. $(ii) \ (x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_4)$

     Solution is $(x_1, x_2, x_3, x_4) = (T, T, F, F)$
Proving the first problem $\text{SAT}$ is NP-Complete

1. Show that $\text{SAT}$ is in NP.
2. Reduce (in poly. time) every problem in NP to $\text{SAT}$.

Proving a new problem $X$ is NP-Complete

1. Show that problem $X$ is in NP.
2. Reduce (in poly. time) an existing NP-Complete problem to $X$. 
3-SAT is in NP-Complete

Problem
• Prove that 3-SAT is NP-Complete.

Solution
1. Show that 3-SAT is in NP:
   Suppose we are given a solution to the 3-CNF Boolean expression.
   In polynomial time we can verify whether the given truth assignment is correct.
2. Show that SAT → 3-SAT:
   ?
Reduction: SAT → 3-SAT

Let $C = (a_1 \lor a_2 \lor A)$ where $A = (a_3 \lor \cdots \lor a_k)$

Let $C' = (a_1 \lor a_2 \lor y) \land (\overline{y} \lor A)$

Proof

- [If $C$ is satisfiable, then $C'$ is satisfiable.]
  - [Case $a_1 = 1$ or $a_2 = 1$.] $C = 1$.
    Assign $y = 0$ to get $C' = (a_1 \lor a_2 \lor 0) \land (1 \lor A) = 1$.
  - [Case $A = 1$.] $C = 1$.
    Assign $y = 1$ to get $C' = (a_1 \lor a_2 \lor 1) \land (0 \lor A) = 1$.

- [If $C'$ is satisfiable, then $C$ is satisfiable.]
  $C' = 1 \Rightarrow (a_1 \lor a_2 \lor y = 1) \land (\overline{y} \lor A) = 1$
  - [Case $y = 0$.] Then $(a_1 \lor a_2) = 1 \Rightarrow C = 1$.
  - [Case $y = 1$.] Then $A = 1 \Rightarrow C = 1$. 
Reduction: SAT $\rightarrow$ 3-SAT

SAT($F$)

1. for each clause $C$ in $F$ with $k$ literals do
2. create $k - 3$ tiny clauses of size 3; using a total of $k - 3$ new variables; call this collection of tiny clauses $C'$
3. let the obtained formula be called $F'$
4. return 3-SAT($F'$)

SAT poly. time reduces to 3-SAT
An **independent set** of a graph $G$ is a subset of the vertices such that no two vertices in the subset represent an edge of $G$.

**Problem:** Independent set (decision version)

*Input:* $G = (V, E)$ and an integer $k$.

*Output:* Is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in $S$?

Problem

- Prove that **IndependentSet** is NP-Complete.

Solution

1. Show that **IndependentSet** is in NP:
   Suppose we are given a subset $S$ of the vertices in a graph. In polynomial time we can verify that, for each pair of vertices in the set $S$, there is no edge between them.

2. Show that $3$-SAT $\rightarrow$ **IndependentSet**:
   ?
Reduction: 3-SAT $\rightarrow$ IndependentSet

**Analysis**

- A clause (of size at most 3) can be transformed to a clause cluster (of size at most 3)
- Add edges between $x_i$ and all its complement vertices $\overline{x}_i$
Reduction: 3-SAT → IndependentSet

\[(x_1 \lor \bar{x}_2 \lor \bar{x}_3) \land (\bar{x}_1 \lor x_2 \lor x_3) \land (\bar{x}_1 \lor x_2 \lor \bar{x}_3) \land (x_1 \lor \bar{x}_2 \lor x_3)\]

The reduction \(k = 4\)

Source: David Mount’s notes

Analysis

- Given a \(k\), one needs to select at \(k\) vertices that satisfy the independent set property
- Select a vertex from each clause without violating the independent set property

Correctness

\(x_1 = x_2 = 1, x_3 = 0\)
Reduction: 3-SAT → IndependentSet

3-SAT($F$)

[Transform the input: Boolean expression to graph]
1. $k \leftarrow$ number of clauses in $F$
2. for each clause $(x_1 \lor x_2 \lor x_3)$ in $F$ do
3. create a clause cluster consisting of three vertices labeled $x_1, x_2, x_3$
4. create edges $(x_1, x_2), (x_2, x_3), (x_3, x_1)$ between all pairs of vertices in the cluster
5. for each vertex $x_i$ do
6. create edges between $x_i$ and all its complement vertices $\overline{x_i}$ (conflict links)

[Transform the output: Vertex set to truth assignment]
7. return $\text{IndependentSet}(G, k)$

3-SAT poly. time reduces to $\text{IndependentSet}$
A vertex cover of a graph $G$ is a subset of the vertices that touch/cover all edges of $G$.

Problem:

Problem: Minimum vertex cover (decision version)
Input: $G = (V, E)$ and a natural number $k$.
Output: Check if there exists a set of $k$ vertices that cover all edges.

Source: Mathworld Wolfram.
### Problem

- Prove that \textsc{VertexCover} is NP-Complete.

### Solution

1. **Show that \textsc{VertexCover} is in NP:**
   
   Suppose we are given a subset $S$ of the vertices in a graph. In polynomial time we can verify that, for each vertex in the set $S$, the edges the vertex covers/touches.

2. **Show that \textsc{IndependentSet} $\rightarrow$ \textsc{VertexCover}:**
   
   ?
Reduction: IndependentSet → VertexCover


**Reduction: IndependentSet → VertexCover**

In a graph $G$, $S$ is an independent set $\iff (V - S)$ is a vertex cover

**Proof**

- **[If $S$ is an independent set, then $(V - S)$ is a vertex cover.]**
  
  If $S$ is an independent set, there is no edge $e = (u, v)$ in $G$, such that both $u, v \in S$. Hence for any edge $e = (u, v)$, at least one of $u, v$ must lie in $(V - S)$.
  
  $\implies (V - S)$ is a vertex cover in $G$.

- **[If $(V - S)$ is a vertex cover, then $S$ is an independent set.]**
  
  If $(V - S)$ is a vertex cover, between any pair of vertices $(u, v) \in S$ if there exists an edge $e$, none of the endpoints of $e$ would exist in $(V - S)$ violating the definition of vertex color. Hence, no pair of vertices in $S$ can be connected by an edge.
  
  $\implies S$ is an independent set in $G$. 
**Reduction: IndependentSet → VertexCover**

<table>
<thead>
<tr>
<th><strong>INDEPENDENTSET(G, k)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. return **VERTEXCOVER(G,</td>
</tr>
</tbody>
</table>

**INDEPENDENTSET** poly. time reduces to **VERTEXCOVER**
Problem

- Prove that TSP is in NP-Complete.

Solution

1. Show that TSP is in NP:
   Suppose we are given a tour in a graph and a natural number $k$.
   In polynomial time we can verify if the given solution is really a tour (covers each vertex exactly once, except the last vertex) and if the total weight of the tour is less than or equal to $k$.

2. Show that HamiltonianCycle $\rightarrow$ TSP:
### Reduction: HamiltonianCycle → TSP

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem: Hamiltonian cycle</td>
</tr>
<tr>
<td>Input: $G = (V, E)$.</td>
</tr>
<tr>
<td>Output: Check if the graph contains a Hamiltonian cycle, i.e., a cycle that passes through all the vertices of the graph exactly once.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem: TSP</td>
</tr>
<tr>
<td>Input: Weighted graph $G = (V, E)$ with nonnegative weights and a natural number $k$.</td>
</tr>
<tr>
<td>Output: Check if the graph contains a simple cycle of length $\leq k$ (i.e., total weight cost) that passes through all the vertices of the graph exactly once.</td>
</tr>
</tbody>
</table>
Reduction: HamiltonianCycle $\rightarrow$ TSP

Hamiltonian cycle in first graph $\iff$ finding TSP of cost 0 is second graph.
Reduction: HamiltonianCycle $\rightarrow$ TSP

<table>
<thead>
<tr>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>For a given graph $G = (V, E)$, create the graph $G' = (V', E')$ as follows:</td>
</tr>
<tr>
<td>• [vertices.] $V' = V$</td>
</tr>
<tr>
<td>• [edges.] $E' = {(u, v)}$ for unique vertices $u, v$ in $V'$</td>
</tr>
<tr>
<td>• [weights.] for each edge $e$ in $E'$:</td>
</tr>
<tr>
<td>$w(e) = 0$ if $e$ is in $E$,</td>
</tr>
<tr>
<td>$w(e) = 1$ if $e$ is not in $E$</td>
</tr>
</tbody>
</table>
**Reduction: HamiltonianCycle → TSP**

**HamiltonianCycle(G) ⇔ TSP(G′, 0)**

<table>
<thead>
<tr>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image.png" alt="Image" /></td>
</tr>
</tbody>
</table>

1. **[HamiltonianCycle(G) ⇒ TSP(G′, 0).]**
   - If \( G \) contains a Hamiltonian cycle, it forms a cycle in \( G' \) with total cost 0 because the weights of all the edges is 0. Hence, there exists a TSP solution in \( G' \) with total cost \( \leq 0 \).

2. **[TSP(G′, 0) ⇒ HamiltonianCycle(G).]**
   - If \( G' \) contains a cycle that passes through all vertices exactly once, and has length \( \leq 0 \), then the cycle contains only the edges that were originally present in graph \( G \). Hence, there exists a Hamiltonian cycle in \( G \).
Reduction: HamiltonianCycle $\rightarrow$ TSP

<table>
<thead>
<tr>
<th><strong>HamiltonianCycle</strong>$(G = (V, E))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. construct complete graph $G' = (V', E')$ such that $V' = V$</td>
</tr>
<tr>
<td>2. for each edge $e$ in $E'$ do</td>
</tr>
<tr>
<td>3. if $e$ is in $E$ then</td>
</tr>
<tr>
<td>4. $w(e) \leftarrow 0$</td>
</tr>
<tr>
<td>5. else</td>
</tr>
<tr>
<td>6. $w(e) \leftarrow 1$</td>
</tr>
<tr>
<td>7. return TSP$(G', 0)$</td>
</tr>
</tbody>
</table>

**HamiltonianCycle** poly. time reduces to TSP