NP-Completeness
Introduction

• Most of the algorithms we have studied have been polynomial time
• This means that for an input of size $n$ their running time is $O(n^k)$ where $k$ is a constant
• Can all problems be solved in polynomial time?
• NO
Tractable Vs. Intractable

• Polynomial time – Tractable
• Super-polynomial time - Intractable
NP-completeness

• Problems whose status is unknown
• No polynomial time algorithms have been discovered yet
• But neither has there been a proof that no polynomial time algorithm exists
Some examples

• Shortest Vs. Longest Path
• Euler tour Vs. Hamiltonian circuit

Euler tour – cycle that traverses each edge of G once. O(E) time solution.

Hamiltonian circuit – cycle that traverses each vertex of G once. NP-complete
Informal problem classes

• P – solvable in polynomial time
• NP – ‘verifiable’ in polynomial time
• NPC – if it is in NP and as hard as any problem in NP.

Basic idea is this – if any problem in NPC can be solved in polynomial time then they all can be solved in polynomial time.

We will prove this later.
Decision problems Vs. Optimization problems

- Optimization problems – Feasible solution with best (max or min) value
  Eg. Shortest path – path with min cost
- Decision problems – ‘Yes’ /‘No’ problems
- We are going to stick to decision problems
- Easy to cast an optimization problem as a decision problem.
Reductions

• Common idea to proving NP-completeness
• Supposed we are given a problem A that we want to solve in polynomial time
• Suppose also we have a different problem B that we already can solve in polynomial time
Reductions

Suppose also we have a procedure that transforms an instance $\alpha$ of $A$ to an instance $\beta$ of $B$ such that:

• The transformation takes polynomial time
• The answers are the same. $\alpha$ is yes only if $\beta$ is yes.

Called a reduction algorithm
Figure 34.1 Using a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B. In polynomial time, we transform an instance $\alpha$ of A into an instance $\beta$ of B, we solve B in polynomial time, and we use the answer for $\beta$ as the answer for $\alpha$. 
Reduction

Since NP-completeness is about showing how hard a problem is, we use a reduction to show that no polynomial time algorithm exists for some problem B.

Suppose we know a decision problem A that is NP-complete. If we can reduce B in polynomial time to A we can show B is NP-complete also.
A first NP-complete problem

Satisfiability:
A boolean formula contains variables (0 or 1 values) with \(^\) and \(\lor\) and NOT operations and parenthesis.
A formula is satisfiable if there is some assignment of values to its variables so that it evaluates to 1.
k-CN\(^n\) if it is the \(\lor\) and \(^\) of exactly k variables.
2-CN\(^n\): \((x_1 \lor \neg(x_2)) ^ (x_1 \lor x_3)\) - Polynomial time
3-CN\(^n\): NP-complete
Languages and Encodings

- A problem instance can be represented in a form the computer understands – **encoding**
- We are used to common encodings e.g. Decimal -> Binary, Alphabet -> ASCII
- This encoding represents a **concrete problem**
- A concrete problem is polynomial solvable if for an instance $i$ of length $n$ an $O(n^k)$ algorithm exists to solve it
- Can consider solving a problem as the set of strings that the algorithm accepts.
Polynomial time verification

Hamiltonian cycles:
A simple cycle containing every vertex in V.

A graph that contains the cycle is called Hamiltonian.

Not all graphs are Hamiltonian.
Figure 34.2  (a) A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by shaded edges.  (b) A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.
Verification

• Given a solution, verifying if it is a Hamiltonian can be done in polynomial time easily.

• Formally:

We define a verification algorithm $A$ as a two-argument algorithm that accepts an ordinary input string $x$ and another binary string $y$ called the certificate. $A$ verifies $x$ if there exists a certificate $y$ such that $A(x,y)=1$. 
Verification and Languages

• The language verified by a verification algorithm $A$ is:

$L = \{ x \in \{0,1\}^* : \text{there exists } y \in \{0,1\}^* \text{ st } A(x,y) = 1 \}$
The complexity class NP

• Informally, NP is the class of problems that can be verified in polynomial time
• Formally, a language $L$ belongs to NP iff $L = \{ x \in \{0,1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ s.t. } A(x,y) = 1 \}$

We say that $A$ verifies a language $L$ in polynomial time.
Some observations

• If $L \in P$ then $L \in NP$. Since if we can solve a problem in polynomial time, we can easily convert it to a verification algorithm.
• It is unknown if $P=NP$ but most researchers believe this is not the case
• Many other question also remain unresolved, for example, is NP closed under complement? i.e. if $L \in NP$, then does $L' \in NP$?
Co-NP

• We can define the complexity class Co-NP as the set of languages as the set of languages $L$ such that $L' \in \text{NP}$.

• Thus closure under complement is the same as asking, $\text{NP}=\text{co-NP}$?

• $P$ is closed under complement.
Figure 34.3  Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. (a) P = NP = co-NP. Most researchers regard this possibility as the most unlikely. (b) If NP is closed under complement, then NP = co-NP, but it need not be the case that P = NP. (c) P = NP \cap co-NP, but NP is not closed under complement. (d) NP \neq co-NP and P \neq NP \cap co-NP. Most researchers regard this possibility as the most likely.
Reducibility

- Intuition: A problem Q can be reduced to another problem Q’ if any instance of Q can be easily rephrased as an instance of Q’, the solution to which provides a solution to the instance of Q.

- \( ax+b=0 \) can be reduced as \( 0.x^2+ax+b=0 \)
Reducibility in terms of languages

A language $L_1$ is **polynomial time reducible** to a language $L_2$ (written as $L_1 \leq^p L_2$) if there exists a polynomial time computable function $f: \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$

$$x \in L_1 \text{ if and only if } f(x) \in L_2$$

$f$: **reduction function** and a polynomial time algorithm $F$ that computes $f$ is called a **reduction algorithm**
Figure 34.4  An illustration of a polynomial-time reduction from a language $L_1$ to a language $L_2$ via a reduction function $f$. For any input $x \in \{0, 1\}^*$, the question of whether $x \in L_1$ has the same answer as the question of whether $f(x) \in L_2$. 
Lemma 34.3
If $L_1, L_2 \in \{0,1\}^*$ are languages such that $L_1 \leq_P L_2$, then $L_2 \in P$ implies $L_1 \in P$

Figure 34.5  The proof of Lemma 34.3. The algorithm $F$ is a reduction algorithm that computes the reduction function $f$ from $L_1$ to $L_2$ in polynomial time, and $A_2$ is a polynomial-time algorithm that decides $L_2$. Illustrated is an algorithm $A_1$ that decides whether $x \in L_1$ by using $F$ to transform any input $x$ into $f(x)$ and then using $A_2$ to decide whether $f(x) \in L_2$. 
NP-Completeness

A language \( L \in \{0,1\}^* \) is NP-complete (class NPC) if:

1. \( L \in \text{NP} \)
2. \( L' \leq_P L \) for every \( L' \in \text{NP} \)

If a language \( L \) satisfies property 2 but not necessarily property 1, we say that \( L \) is **NP-hard**.
Theorem 34.4

If any NP-complete problem is polynomial-time solvable, then P=NP. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.
Figure 34.6  How most theoretical computer scientists view the relationships among P, NP, and NPC. Both P and NPC are wholly contained within NP, and $P \cap NPC = \emptyset$. 
Circuit Satisfiability

• Goal – prove at least one problem to be NP-complete.
• Proving CIRCUIT-SAT is NP-complete is beyond the scope of this course.
• We can loosely argue why this is so.
Figure 34.7  Three basic logic gates, with binary inputs and outputs. Under each gate is the truth table that describes the gate’s operation. (a) The NOT gate. (b) The AND gate. (c) The OR gate.
Figure 34.8 Two instances of the circuit-satisfiability problem. (a) The assignment \(x_1 = 1, x_2 = 1, x_3 = 0\) to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. (b) No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.
Showing it NP-complete has two parts:
- Circuit satisfiability belongs to the class NP
  This involves showing that it can be verified.
- Circuit satisfiability is NP-hard
Proof Process

If $L$ is a language such that $L' \leq_p L$ for some $L'$ in NPC, then $L$ is NP-Hard. Moreover if $L$ is in NP the $L$ is in NPC.
Proof process for showing L is NP-complete

• Prove L is in NP
• Select a known NP-complete language L’
• Describe an algorithm that computes a function f mapping every instance x of L’ to L.
• Prove that the function f satisfies x ∈ L’ iff f(x) ∈ L for all x ∈ \{0,1\}*
• Prove that the algorithm computing x runs in polynomial time
An example: Formula Satisfiability

A boolean formula comprises of the following:

• n boolean variables $x_1, x_2, ..., x_n$
• m boolean connectives ( ^, V, NOT, ->, <->)
• Paranthesis

The satisfiability problem asks if a boolean formula is satisfiable (has a truth assignment)
Figure 34.10 Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.
Figure 34.11  The tree corresponding to the formula $\phi = ((x_1 \rightarrow x_2) \lor \neg((\neg x_1 \leftrightarrow x_3) \lor x_4)) \land \neg x_2$. 
The clique problem

- A clique in a graph G is a subset V’ of V, each pair of which is connected by an edge in E.
- **Size** of the clique is the number of vertices it contains
- **Clique** problem is an optimization problem of finding the clique with the largest size in the graph.

CLIQUE = \{ <G,k>: G is a graph with a clique of size k \}
Naïve approach

• Naïve algorithm to determine if a graph G has a clique of size $k$ is to list all $k$-subsets of V and check each one to see if forms a clique.

• Clique problem is NP-complete.
Proof

To show CLIQUE $\in$ NP, it's easy to see that checking if a given subset can form a clique can be done in polynomial time.

Next step is Reduction. We show that 3-CNF-SAT $\leq_p$ CLIQUE.

This is somewhat surprising. What does formula satisfiability have to do with graphs?
Reduction

• Let $\Phi = C_1 \land C_2 \land \ldots \land C_k$ be a boolean formula in 3-CNF with $k$ clauses.
• For $r=1, 2, \ldots, k$ each clause $C_r$ has exactly three distinct literals $l_1^r, l_2^r, l_3^r$.
• We shall construct a graph $G$ such that $\Phi$ is satisfiable iff $G$ has a clique of size $k$. 

Graph construction

The graph is constructed as follows. For each clause $C_r = (l_1^r, l_2^r, l_3^r)$ in $\Phi$ we place a triple of vertices $v_1^r, v_2^r, v_3^r$ into $V$.

We put an edge between two vertices $v_i^r, v_j^s$ if both of the following hold:

- $v_i^r$ and $v_j^s$ are in different triples, i.e. $r \neq s$, and
- their corresponding literals are consistent, i.e. $l_i^r$ is not the negation of $l_j^s$
Figure 34.14 The graph $G$ derived from the 3-CNF formula $\phi = C_1 \land C_2 \land C_3$, where $C_1 = (x_1 \lor \neg x_2 \lor \neg x_3)$, $C_2 = (\neg x_1 \lor x_2 \lor x_3)$, and $C_3 = (x_1 \lor x_2 \lor x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and $x_1$ may be either 0 or 1. This assignment satisfies $C_1$ with $\neg x_2$, and it satisfies $C_2$ and $C_3$ with $x_3$, corresponding to the clique with lightly shaded vertices.
• We have to show that this transformation is a reduction.
• First suppose that $\Phi$ has a satisfying assignment.
• Then, each clause $C_r$ contains at least one literal $l_i^r$ that is assigned 1 and each such literal corresponds to $v_i^r$.
• Picking one such true vertex from each clause yields a set $V'$ of $k$ vertices.
• We claim $V'$ is a clique
• For any $v_i^r$ and $v_j^s$, $r \neq s$, both corresponding literals $l_i^r$ and $l_j^s$ are mapped to 1 by the given satisfying assignment and thus cannot be complements.

• Conversely suppose $G$ has a clique of size $k$. No edges in $G$ connect vertices in the same triple, so $G$ contains exactly one vertex per triple.

• Assigning a 1 to each such literal can be done without fear of assigning it to both a literal and its complement.

• Each clause is satisfied and so $\Phi$ is satisfied.
The vertex-cover problem

- A vertex cover of an undirected graph $G=(V,E)$ is a subset $V'$ of $V$ such that if $(u,v) \in E$, then $u \in V$ or $v \in V'$ (or both).
- Each vertex “covers” incident edges and a vertex cover for $G$ “covers” all edges in $E$.
- Size of a vertex cover – the number of vertices in it.
- **Vertex cover Problem** is to find a vertex cover of minimum size.
The vertex-cover problem

As a decision problem:

• VERTEX-COVER=\{<G,k>: graph G has a vertex cover of size k\}

• VERTEX-COVER is NP-complete
PROOF

VERTEX-COVER $\in$ NP:

A verification algorithm can affirm that some certificate $V'$ has $|V'|=k$ and check each edge $(u,v)$ of the graph $G$ to see if $u \in V'$ or $v \in V'$. Straightforward, polynomial time.
Proof

We prove vertex-cover is NP-hard by showing $\text{CLIQUE} \leq_p \text{VERTEX-COVER}$

Reduction based on the idea of the complement of a graph.

$G=(V,E)$, the complement of $G$ is $G^\wedge=(V, E^\wedge)$

where $E^\wedge=\{(u,v): u, v \in V, u \neq v \text{ and } (u, v) \text{ not in } E\}$

$G'$ contains those edges not in $G$. 
Figure 34.15  Reducing CLIQUE to VERTEX-COVER. (a) An undirected graph $G = (V, E)$ with clique $V' = \{u, v, x, y\}$. (b) The graph $\overline{G}$ produced by the reduction algorithm that has vertex cover $V - V' = \{w, z\}$. 
The reduction takes as input an instance \((G, k)\) of the CLIQUE problem. Computes \(G^\) in polynomial time.

Output of reduction is \((G^, |V|-k)\) of the vertex-cover problem.

To show this is a reduction we need to show that 
\(G\) has a clique of size \(k\) iff \(G^\) has a vertex cover of size \(|V|-k\)
Suppose $G$ has a clique $V'$ with $|V'| = k$. We claim that $V - V'$ is a vertex cover of $G^\wedge$.

Let $(u,v)$ be any edge in $E^\wedge$. Then $(u,v)$ is not in $E$, which implies that at least one of $u$ or $v$ does not belong to $V'$, since every pair of vertices in $V'$ is connected by an edge of $E$.

Equivalently at least one of $u$ or $v$ is $V - V'$ which means that the edge $(u,v)$ is covered by $V - V'$.

Since $(u,v)$ was chosen arbitrarily from $E^\wedge$, every edge of $E^\wedge$ is covered by a vertex in $V - V'$. 
Conversely, suppose $G^\vDash$ has a vertex cover $V'$ where $|V'| = |V| - k$. Then, for all $(u,v) \in V$, if $(u,v) \in E^\vDash$ then $u \in V'$ or $v \in V'$ or both.

The contrapositive of this implication is that for all $u,v \in V$, if $u$ is not in $V'$ and $v$ is not in $V'$ then $(u,v) \in E$. In other words, $V-V'$ is a clique and has a size $|V| - |V'| = k$.
Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.
Subset-Sum problem

• Given a finite set $S$ and a target $t$.
• We ask if there is a subset $S'$ of $S$ whose element sum to $t$.
• Eg: $S\{1,2,7,14,49,98,343,686,2409,2793,16808,17206,117705,117993\}$, and $t=138457$ then $S'=\{1,2,7,98,343,686,2409,17206,117705\}$
Subset-sum problem

SUBSET-SUM = \{ <S, t> : there exists a subset \( S' \) of \( S \) such that \( t = \sum_{s \in S'} s \) \}

Assumption is that our input is encoded in binary.

SUBSET-SUM is NP-complete.
SUBSET-SUM is NP-Complete

• SUBSET-SUM is in NP

Let $S'$ a subset of $S$ be the certificate. Checking if the sum of the elements of $S'$ is $t$ takes polynomial time.

Hence SUBSET-SUM is verifiable in polynomial time.
• SUBSET-SUM is NP-Hard

We show this by showing that 3-CNF-SAT \leq_p SUBSET-SUM.

Given a 3-CNF formula \( \Phi \) over variables \( x_1, x_2, \ldots, x_n \) with clauses \( C_1, C_2, \ldots, C_k \) each containing exactly 3 distinct literals, the reduction constructs an instance \( <S,t> \) of the subset-sum problem such that \( \Phi \) is satisfiable iff there is a subset of \( S \) whose sum is exactly \( t \).
Without loss of generality, we can make two simplifying assumptions on $\Phi$

• First, no clause contains both a variable and its negation

• Each variable appears in at least one clause

The reduction creates two numbers in set $S$ for each variable $x_i$ and two numbers in $S$ for each clause $C_j$. We create all numbers in base 10, with each number having $n+k$ digits and each digit corresponding to one variable or clause.
We construct $S$ and $t$ as follows. To do this, we label each digit by a variable or a clause. Least significant $k$ digits by clauses and most significant $n$ digits by variables.
<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v'_1$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$v_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v'_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$v_3$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$v'_3$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$s_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$s'_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$s_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$s'_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$s_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$s'_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>$s_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$s'_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$t = 1 \ 1 \ 1 \ 4 \ 4 \ 4 \ 4 \ 4$

**Figure 34.19** The reduction of 3-CNF-SAT to SUBSET-SUM. The formula in 3-CNF is $\phi = C_1 \land C_2 \land C_3 \land C_4$, where $C_1 = (x_1 \lor \neg x_2 \lor \neg x_3)$, $C_2 = (\neg x_1 \lor \neg x_2 \lor \neg x_3)$, $C_3 = (\neg x_1 \lor \neg x_2 \land x_3)$, and $C_4 = (x_1 \lor x_2 \lor x_3)$. A satisfying assignment of $\phi$ is $(x_1 = 0, x_2 = 0, x_3 = 1)$. The set $S$ produced by the reduction consists of the base-10 numbers shown; reading from top to bottom, $S = \{1001001, 1000110, 100001, 101110, 10011, 111100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$. The target $t$ is 1114444. The subset $S' \subseteq S$ is lightly shaded, and it contains $v'_1, v'_2, v_3$, corresponding to the satisfying assignment. It also contains slack variables $s_1, s'_1, s'_2, s_3, s_4, s'_4$ to achieve the target value of 4 in the digits labeled by $C_1$ through $C_4$. 
The target \( t \) has a 1 in each digit labeled by a variable and a 4 in each digit labeled by a clause.

For each variable \( x_i \), there are two integers, \( v_i \) and \( v'_i \), in \( S \). Each has a 1 in the digit labeled by \( x_i \) and 0's in the other variable digits. If literal \( x_i \) appears in clause \( C_j \), then the digit labeled by \( C_j \) in \( v_i \) contains a 1. If literal \( \neg x_i \) appears in clause \( C_j \), then the digit labeled by \( C_j \) in \( v'_i \) contains a 1. All other digits labeled by clauses in \( v_i \) and \( v'_i \) are 0.

All \( v_i \) and \( v'_i \) values in set \( S \) are unique. Why? For \( l \neq i \), no \( v_l \) or \( v'_l \) values can equal \( v_i \) and \( v'_i \) in the most significant \( n \) digits. Furthermore, by our simplifying assumptions above, no \( v_i \) and \( v'_i \) can be equal in all \( k \) least significant digits. If \( v_i \) and \( v'_i \) were equal, then \( x_i \) and \( \neg x_i \) would have to appear in exactly the same set of clauses. But we assume that no clause contains both \( x_i \) and \( \neg x_i \), and that either \( x_i \) or \( \neg x_i \) appears in some clause, and so there must be some clause \( C_j \) for which \( v_i \) and \( v'_i \) differ.

For each clause \( C_j \), there are two integers, \( s_j \) and \( s'_j \) in \( S \). Each has 0's in all digits other than the one labeled by \( C_j \). For \( s_j \), there is a 1 in the \( C_j \) digit, and \( s'_j \) has a 2 in this digit. These integers are "slack variables," which we use to get each clause-labeled digit position to add to the target value of 4.

Simple inspection of Figure 34.19 demonstrates that all \( s_j \) and \( s'_j \) values in \( S \) are unique in set \( S \).
Suppose that $\Phi$ has a satisfying assignment. For $i=1,2,..n$, if $x_i=1$ in this assignment, then include $v_i$ in $S'$. Otherwise include $v_i'$. Hence, for each variable-labeled digit, the sum of values in $S'$ must be 1 which matches $t$. Because each clause is satisfied there is some literal in the clause with value 1. Therefore each digit labeled by a clause has at least 1 contribute to its value by a $v_i$ or $v_i'$ value in $S'$. 
The Traveling-Salesman problem

- A salesman must visit $n$ cities.
- Make a tour or hamiltonian cycle, visiting each city exactly once and finishing at the city where he started.
- Integer cost $c(i,j)$ to travel between city $i$ and city $j$
- **Objective:** Minimize the total cost of the tour
Figure 34.18  An instance of the traveling-salesman problem. Shaded edges represent a minimum-cost tour, with cost 7.
TSP

• Stated as a decision problem:

\[ TSP = \{<G,c,k>: \quad G=(V,E) \text{ is a complete graph} \]
\[ \quad c \text{ is a function from } V \times V \rightarrow \mathbb{Z} \]
\[ \quad k \in \mathbb{Z}, \text{ and} \]
\[ \quad G \text{ has a traveling salesman tour} \]
\[ \quad \text{with cost at most } k \} \]
TSP is NP-complete.

- TSP $\in$ NP.

Given an instance, a sequence of $n$ vertices in the tour is the certificate.

Verification algorithm checks that this sequence contains each vertex exactly once, sums the edge cost and checks whether the sum is at most $k$.

Polynomial.
• TSP is NP-hard

Show that HAM-CYCLE ≤_p TSP