CMSC 451: Lecture 15
NP-Completeness: Basic Definitions

Efficiency and Polynomial Time: Up to this point of the semester we have been building up your "toolkit" for solving algorithmic problems efficiently. Hopefully when presented with a computational problem, you now have a clearer idea the sorts of techniques that could be used to solve the problem efficiently (such as divide-and-conquer, DFS, greedy, dynamic programming, network flow).

What do we mean when we say "efficient"? If n is small, a running time of 2^n may be just fine, but when n is huge, even n^2 may be unacceptably slow. A general point of agreement is that algorithmic solutions that rely on *brute-force search* run in *exponential time* are not efficient. The alternative is a structured approach. Such solutions run in *polynomial time*, that is, the worst-case running time is $O(n^c)$, where c is a nonnegative constant. (Note that running times like $O(n \log n)$ are polynomial time, since $n \log n = O(n^2)$.) By *exponential time* we mean any function that is at least $\Omega(c^n)$ for a constant $c > 1$. (This includes $n!$, since $n! \approx (n/e)^n > 2^n = \Omega(2^n)$.)

Henceforth, we will use the terms "efficient" and "easy" to mean solvable by an algorithm whose asymptotic worst-case running time is *polynomial in the input size* (even if that polynomial is n^{1000}). Note the emphasis on the term "worst-case". There are problems for which simple heuristics can efficiently solve almost all "typical" input instances but no efficient algorithmic solution is known that works for all inputs. (A good example of this is the Graph Isomorphism problem.)

While the distinction between worst-case polynomial time and worst-case exponential time is quite crude, it has a number of advantages. For example, the composition of any two polynomials is a polynomial. (That is, if $f(n)$ and $g(n)$ are both polynomials, then so is $f(g(n))$. For example, $(n^a)^b = n^{ab}$.) This means that, if a program makes a polynomial number of calls to a function that runs in polynomial time, then the overall running time is a polynomial.

The Emergence of Hard Problems: Near the end of the 1960's, although there was great success in finding efficient solutions to many combinatorial problems, there was also a growing list of problems which were "hard" in the sense that no known efficient algorithmic solutions existed for these problems.

A remarkable discovery was made about this time. Many of these believed hard problems turned out to be equivalent, in the sense that if you could solve *any one* of them in polynomial time, then you could solve *all* of them in polynomial time. Often these hard problems involved slight generalizations to problems that are solvable in polynomial time. A list of some of these problems is shown in Table 1.

The mathematical theory, which was developed by Richard Karp and Stephen Cook, gave rise to the notions of P, NP, and NP-completeness. Since then, thousands of problems were identified as being in this equivalence class. It is widely believed that none of them can be solved in polynomial time, but there is no proof of this fact. This has given rise to arguably the biggest open problems in computer science:

$$\text{P} = \text{NP?}$$

Table 1: Computationally hard problems and related (easy) counterparts.

Hard Problems (NP-complete)	**Easy Problems (in P)**
3SAT	2SAT
Traveling Salesman Problem (TSP)	Minimum Spanning Tree (MST)
Longest (Simple) Path	Shortest Path
Hypergraph Matching	Graph Matching
Knapsack	Unary Knapsack
Independent Set in Graphs	Independent Set in Trees
Integer Linear Programming	Linear Programming (weak poly time)
Hamiltonian Cycle	Eulerian Cycle
Balanced Cut	Minimum Cut

While we will not be able to provide an answer to this question, we will investigate this concept in the next few lectures.

Note that represents a radical departure from what we have been doing so far this semester. The goal is no longer to prove that a problem *can* be solved efficiently by presenting an algorithm for it. Instead we will be trying to show that a problem *cannot* be solved efficiently. The question is how to do this?

Decision Problem: Many of the problems that we have discussed involve *optimization* of one form or another: find the shortest path, find the minimum cost spanning tree, find the maximum flow. For rather technical reasons, most NP-complete problems that we will discuss will be phrased as decision problems.

A problem is called a *decision problem* if its output is a simple "yes" or "no" (or you may think of this as True/False, 0/1, accept/reject). For example, the minimum spanning tree decision problem might be: "Given a weighted graph G and an integer z, does G have a spanning tree whose weight is at most z?" Observe that a solution to the decision problem can often be converted into a solution to the corresponding optimization problem, for example, by combining a decision algorithm with a binary search over possible values z.

This may seem like a less interesting formulation of the problem. It does not ask for the weight of the minimum spanning tree, and it does not even ask for the edges of the spanning tree that achieves this weight. However, our job will be to show that certain problems *cannot* be solved efficiently. If we show that the simple decision problem cannot be solved efficiently, then certainly the more general optimization problem certainly cannot be solved efficiently either.

Language Recognition: For historical reasons, decision problems are often expressed as *language recognition* problems. A *language* is defined to be a set (finite or infinite) of strings. For example, the set of strings that are palindromes (reading the same backwards and forwards) is an example of a language. We'll see that we can use the concept of languages to describe any computational problem.

To express a computational problem as a language-recognition problem, we first should be able to express its input as a string. Given any mathematical object I, define *serialize*(I) to be a function that maps I to a string. (For example, serializing a graph would involve outputting a string that encodes all its vertices, all its edges, and any additional information such as edge weights.)

Using this, we could define a language MST encoding the Minimum Spanning Tree problem as a language (that is, a collection of strings):

$$\text{MST} = \{\text{serialize}(G, z) \mid G \text{ has a minimum spanning tree of weight at most } z\}.$$

Since it will be a hassle to continuously refer to the serialization function, we will just write this more succinctly as

$$\text{MST} = \{(G, z) \mid G \text{ has a minimum spanning tree of weight at most } z\}.$$

Now, consider any string x that is a valid serialization (that is, encoding) of a graph G and integer z. The language recognition question "Is x a member of the language MST?" could amount to carrying out the following procedure:

- Decode x as a graph G and integer z. (If the encoding is invalid, report "no" right away and terminate.)
- Run any MST algorithm (e.g., Kruskal) on G to determine the weight w of MST(G).
- If $w \leq z$, report "yes" and otherwise report "no".

In the terminology of language recognition, reporting "yes" is called *accepting* the input x and reporting "no" is called *rejecting* the input.

The Class P: We now present an important definition:

Definition: P is the set of all languages (i.e., decision problems) for which membership can be determined in (worst case) polynomial time.

Intuitively, P corresponds to the set of all decisions problems that can be solved efficiently, that is, in polynomial time. Note P is not a language, rather, it is a set of languages. A set of languages that is defined in terms of how hard it is to determine membership is called a *complexity class*. (Therefore, P is a complexity class.)

Since Kruskal's algorithm runs in polynomial time, it follows that MST $\in$ P. We could define equivalent languages for all of the other optimization problems we have seen this year (e.g., shortest paths, max flow, min cut).

A Harder Example: To show that not all languages are (obviously) in P, consider the following:

$$\text{HC} = \{G \mid G \text{ has a simple cycle that visits every vertex of } G\}.$$

Such a cycle is called a *Hamiltonian cycle* and the decision problem is the *Hamiltonian Cycle Problem*. (It was named after the great 19th century mathematician and physicist, William Rowan Hamilton, famed for the discovery of quaternions.)

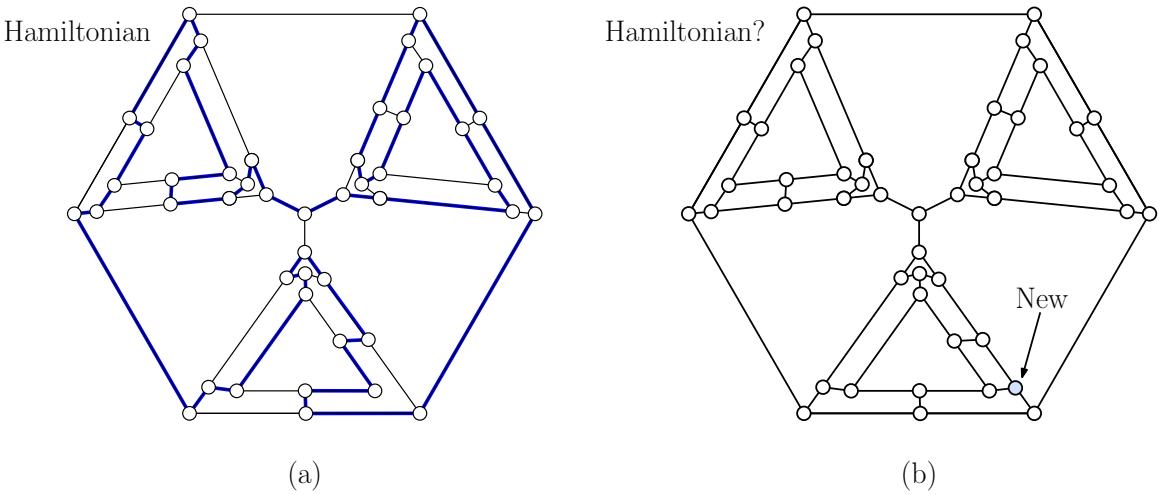

Fig. 1: The Hamiltonian cycle (HC) problem.

In Fig. 1(a) we show an example of a Hamiltonian cycle in a graph. If you think that the problem is easy to solve, try to solve the problem on the graph shown in Fig. 1(b), which has one additional vertex and one additional edge. Either find a Hamiltonian cycle in this graph or prove than none exists. To make this even harder, imagine a million-vertex graph with many slight variations of this pattern.

Is HC $\in$ P? No one knows the answer for sure, but it is conjectured that it is not. (In fact, we will show that later that HC is NP-complete.)

In what follows, we will be introducing a number of classes. We will jump back and forth between the terms "language" and "decision problems", but for our purposes they mean the same things. Before giving all the technical definitions, let us say a bit about what the general classes look like at an intuitive level.

Polynomial-Time Verification and Certificates: In order to define NP-completeness, we need to first define "NP". Unfortunately, providing a rigorous definition of NP will involve a presentation of the notion of *nondeterministic* models of computation, and will take us away from our main focus. (Formally, NP stands for "*nondeterministic polynomial time*".) Instead, we will present a very simple, hand-wavy definition, which will suffice for our purposes.

To do so, it is important to first introduce the notion of a verification algorithm. Many language recognition problems that may be *hard to solve*, but they have the property that they are *easy to verify* that a string is in the language. Recall the Hamiltonian cycle problem defined above. As we saw, there is no obviously efficient way to find a Hamiltonian cycle in a graph. However, suppose that a graph *does* have a Hamiltonian cycle, and someone wanted to convince us of its existence. This person would simply tell us the vertices in the order that they appear along the cycle. It would be a very easy matter for us to inspect the graph and check that this is indeed a legal cycle that it visits all the vertices exactly once. Thus, even though we know of no efficient way to *solve* the Hamiltonian cycle problem, there is a very efficient way to *verify* that a given graph has one. (You might ask, but what if the graph did

not have one? Don't worry. A verification process is not required to do anything if the input is not in the language.)

The given cycle in the above example is called a certificate. A *certificate* is any piece of information which allows us to verify that a given string is in a language in polynomial time.

More formally, given a language L, a *verification algorithm* is any algorithm that is given an input string x. If $x \in L$, then there exists a string y, called the *certificate*, such that it is possible to prove (with the aid of y) that $x \in L$. If $x \notin L$, then we do not care what the algorithm does. If there exists a verification algorithm that runs in polynomial time, we say that L can be *verified in polynomial time* (or *polynomially verifiable*).

What do we mean by "prove that $x \in L$ with the aid of y"? This is where we will be a bit hand-wavy. In the case of Hamiltonian Cycle, HC is the language of (encoded) graphs that have Hamiltonian cycles. If $x \in \text{HC}$, that is, if this graph has a Hamiltonian cycle, then we can take y to be the sequence of vertices that form the cycle. The "proof" consists of verifying that y indeed defines a cycle in the graph that visits every vertex exactly once. Clearly, if x is represented in any reasonable manner (e.g., an adjacency matrix), we can do this in time that is linear in the number of vertices.

Are All Languages Polynomially Verifable? You might wonder whether given the immense power of being able to pick a certificate out of thin air, isn't every language polynomially verifiable. Let's see some examples of problems that do not seem to be polynomially verifiable.

$$\begin{aligned} \text{UHC} &= \{G \mid G \text{ has a } \textit{unique} \text{ Hamiltonian cycle}\} \\ \overline{\text{HC}} &= \{G \mid G \text{ has } \textit{no} \text{ Hamiltonian cycle}\}. \end{aligned}$$

There is no known polynomial time verification algorithm for either of these. For example, suppose that a graph G is in the language UHC. What information would someone give us that would allow us to verify that G is indeed in the language? They could certainly show us one Hamiltonian cycle, but it is unclear that they could provide us with any easily verifiable piece of information that would demonstrate that this is the only one. For $\overline{\text{HC}}$, it is unclear what certificate could be provided to convince us that the graph has no Hamiltonian cycle.

The class NP: We can now define the complexity class NP.

Definition: NP is the set of all languages that can be verified in polynomial time.

Observe that if we can solve a problem efficiently without a certificate, we can certainly solve given the additional help of a certificate. Therefore, $\text{P} \subseteq \text{NP}$. However, it is not known whether $\text{P} = \text{NP}$. It seems unreasonable to think that this should be so. In other words, just being able to verify that you have a correct solution does not help you in finding the actual solution very much. Most experts believe that $\text{P} \neq \text{NP}$, but no one has a proof of this.

Reductions: One way to attack the question of whether $\text{P} \neq \text{NP}$ would be show that there exists a language L such that $L \in \text{NP}$ but $L \notin \text{P}$. A good candidate for such a problem would be the "hardest" problem in NP. To define what it means for one problem to be hard relative to another, we introduce the notion of reduction, that is, using an algorithm for one problem to solve another.

Before discussing reductions, let us just consider the following question. Suppose that there are two problems, H and U. We know (or you strongly believe at least) that H is *hard*, that is it cannot be solved in polynomial time. On the other hand, the complexity of U is *unknown*. We want to prove that U is also hard. How would we do this? Effectively, we want to show that

$$(H \notin \mathrm{P}) \;\Rightarrow\; (U \notin \mathrm{P}).$$

To show that U is not solvable in polynomial time, we will suppose (towards a contradiction) that a polynomial time algorithm for U did existed, and then we will use this algorithm to solve H in polynomial time, thus yielding a contradiction. In other words, we could prove the contrapositive,

$$(U \in \mathrm{P}) \;\Rightarrow\; (H \in \mathrm{P}).$$

To make this more concrete, suppose that we had a subroutine[1] that can solve any instance of problem U in polynomial time. Given an input x for the problem H, we could faithfully translate it into an input x' for U. By "faithful" we mean that $x \in H$ if and only if $x' \in U$ (see Fig. 2). Then we run our U subroutine on x' and output whatever it outputs.

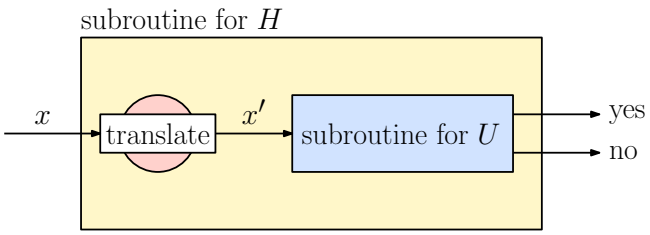

Fig. 2: Reducing H to U.

It is easy to see that if U is solvable in polynomial time, then so is H. We assume that the translation module runs in polynomial time. If so, we call this a *polynomial reduction*[2] of problem H to problem U, which is denoted $H \leq_P U$.

3-Colorability and Clique Cover: Let us consider an example to make this clearer. The following problem is well-known to be NP-complete, and hence it is strongly believed that the problem cannot be solved in polynomial time.

3-coloring (3Col): Given a graph G, can each of its vertices be labeled with one of three different "colors", such that no two adjacent vertices have the same label (see Fig. 3(a) and (b)).

Coloring arises in various partitioning problems. (E.g., You are arranging your relatives to sit at three big tables during a wedding reception. You have pairs that don't get along,

[1] It is important to note here that this supposed subroutine for U is a *fantasy*. We know (or strongly believe) that H cannot be solved in polynomial time, thus we are essentially proving that such a subroutine cannot exist, implying that U cannot be solved in polynomial time.

[2] Richard Karp used this style of reduction in his influential paper on NP-completeness, and for this reason, it is called a *Karp reduction*. More generally, we might consider calling the subroutine multiple times like a black-box. This is called a *Cook reduction*, after Stephen Cook. While Cook reductions are theoretically more powerful, Karp reductions are simpler and work for virtually all NP-completeness proofs.

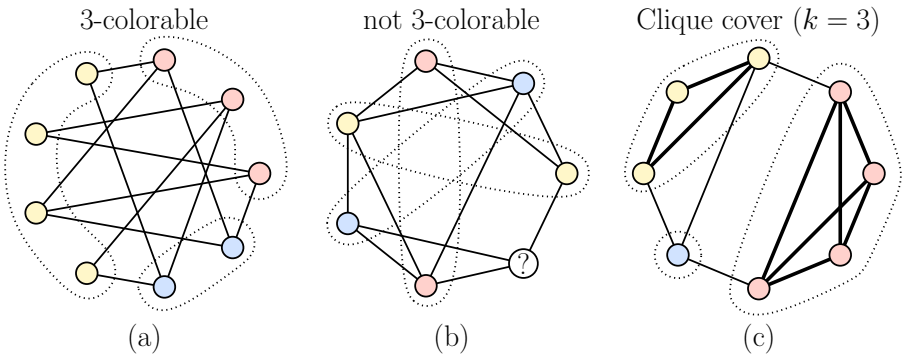

Fig. 3: 3-coloring and Clique Cover.

each represented by an edge in your graph. Can you assign them to three tables avoiding warring pairs at the same table?) It is well known that planar graphs can be colored with four colors, and there exists a polynomial time algorithm for doing this. But determining whether three colors are possible (even for planar graphs) seems to be hard, and there is no known polynomial time algorithm.

The 3Col problem will play the role of the known hard problem H. To play the role of U, consider the following problem. Given a graph $G = (V, E)$, we say that a subset of vertices $V' \subseteq V$ forms a *clique* if for every pair of distinct vertices $u, v \in V'$ $(u, v) \in E$. That is, the subgraph induced by V' is a complete graph.

Clique Cover (CCov): Given a graph $G = (V, E)$ and an integer k, can we partition the vertex set into k subsets of vertices $V_1, \ldots, V_k$ such that each V_i is a clique of G (see Fig. 3(c)).

The clique cover problem arises in clustering. We put an edge between two nodes if they are similar enough to be clustered in the same group. We want to know whether it is possible to cluster all the vertices into at most k groups.

We want to prove that CCov is hard, under the assumption that 3Col is hard, that is,

$$(\text{3Col} \notin \text{P}) \implies (\text{CCov} \notin \text{P}).$$

Again, we'll prove the contrapositive:

$$(\text{CCov} \in \text{P}) \implies (\text{3Col} \in \text{P}).$$

Let us assume that we have access to a polynomial time subroutine $\text{CCov}(G, k)$. Given a graph G and an integer k, this subroutine returns true (or "yes") if G has a clique cover of size k and false otherwise. How can we use this *alleged* subroutine to solve the well-known hard 3Col problem? We need to find a translation, that maps an instance G for 3-coloring into an instance (G', k) for clique cover (see Fig. 4).

Observe that both problems involve partitioning the vertices up into groups. There are two differences. First, in the 3-coloring problem, the number of groups is fixed at three. In the

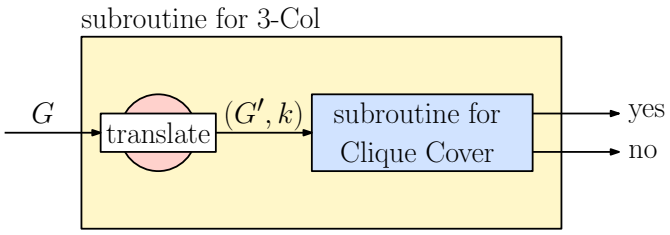

Fig. 4: Reducing 3Col to CCov.

Clique Cover problem, the number is given as an input. Second, in the 3-coloring problem, in order for two vertices to be in the same group they should *not* have an edge between them. In the Clique Cover problem, for two vertices to be in the same group, they *must* have an edge between them. Our translation therefore, should convert edges into non-edges and vice versa.

This suggests the following idea for reducing the 3-coloring problem to the Clique Cover problem. Given a graph G, let $\overline{G}$ denote the *complement graph*, where two distinct nodes are connected by an edge if and only if they are not adjacent in G. Let G be the graph for which we want to determine its 3-colorability. The translator outputs the pair $(\overline{G}, 3)$. We then feed the pair $(G', k) = (\overline{G}, 3)$ into a subroutine for clique cover (see Fig. 5).

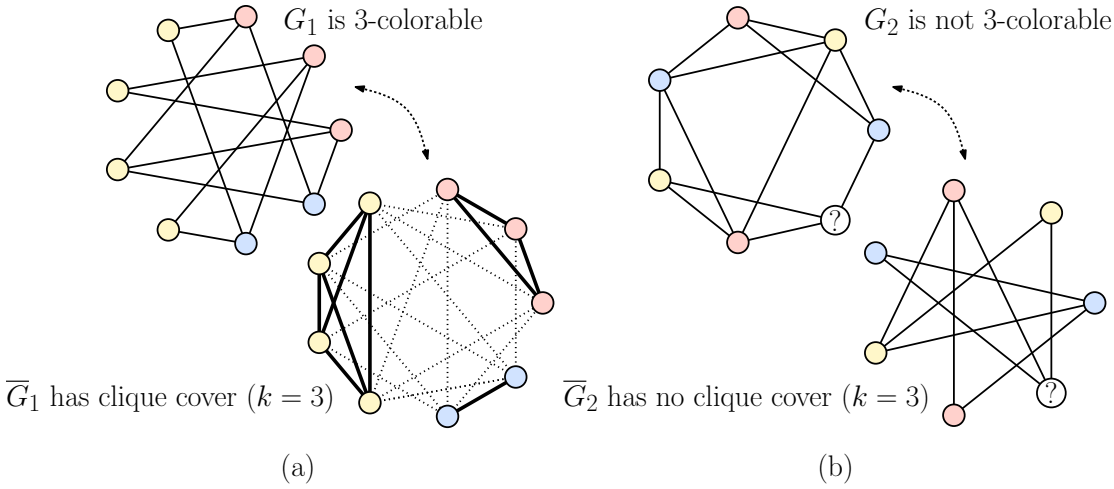

Fig. 5: Clique covers in the complement.

The following formally establishes the correctness of this reduction by showing that we have *faithfully* translated an instance of 3Col to an equivalent instance of CCov.

Claim: A graph $G = (V, E)$ is 3-colorable if and only if its complement $\overline{G} = (V, \overline{E})$ has a clique-cover of size 3. In other words,

$$G \in \text{3Col} \quad \Longleftrightarrow \quad (\overline{G}, 3) \in \text{CCov}.$$

Proof: ($\Rightarrow$) If G 3-colorable, then let V_1, V_2, V_3 be the three color classes. We claim that this is a clique cover of size 3 for $\overline{G}$, since if u and v are distinct vertices in V_i, then $\{u, v\} \notin E$

(since adjacent vertices cannot have the same color) which implies that $\{u, v\} \in \overline{E}$. Thus every pair of distinct vertices in V_i are adjacent in $\overline{G}$.

($\Leftarrow$) Suppose $\overline{G}$ has a clique cover of size 3, denoted V_1, V_2, V_3. For $i \in \{1, 2, 3\}$ give the vertices of V_i color i. We assert that this is a legal coloring for G, since if distinct vertices u and v are both in V_i, then $\{u, v\} \in \overline{E}$ (since they are in a common clique), implying that $\{u, v\} \notin E$. Hence, two vertices with the same color are not adjacent.

It is useful to observe that the reduction was from 3Col to CCov, which means that we are at liberty to use any value of k we like, and $k = 3$ was convenient. You might wonder why we didn't need to consider other values of k. The reason is that we didn't need to. Of course, if we were trying to do the reduction in the opposite direction from CCov to 3Col, we would need to worry about this.

Polynomial-time reduction: We now take this intuition of reducing one problem to another through the use of a subroutine call, and place it on more formal footing. Notice that in the example above, we converted an instance of the 3-coloring problem (G) into an equivalent instance of the Clique Cover problem ($\overline{G}, 3$).

Definition: We say that a language (i.e. decision problem) L_1 is *polynomial-time reducible* to language L_2 (written $L_1 \leq_P L_2$) if there is a polynomial time computable function f, such that for all x, $x \in L_1$ if and only if $f(x) \in L_2$.

In the previous example we showed that 3Col $\leq_P$ CCov, and in particular, $f(G) = (\overline{G}, 3)$. Note that it is easy to complement a graph in $O(n^2)$ (i.e. polynomial) time (e.g. flip 0's and 1's in the adjacency matrix). Thus f is computable in polynomial time.

Intuitively, saying that $L_1 \leq_P L_2$ means that "if L_2 is solvable in polynomial time, then so is L_1." This is because a polynomial time subroutine for L_2 could be applied to $f(x)$ to determine whether $f(x) \in L_2$, or equivalently whether $x \in L_1$. Thus, in sense of polynomial time computability, L_1 is "no harder" than L_2.

You shouldn't read too much into the notation or make incorrect inferences. For example, this does not imply that L_1 is necessarily easier in the sense that its running time is smaller. It may very well be that $L_1 \leq_P L_2$ and L_1 takes $O(n^{10})$ time to compute while L_2 takes only $O(n)$ time. It could be that L_1 is solvable in polynomial time, but L_2 takes exponential time to compute. It could be that both take exponential time to compute. What we can infer, however, is that, if L_2 is solvable in polynomial time, then L_1 *cannot* take exponential time.

The way in which inequality is applied in NP-completeness is exactly the converse. We usually have strong evidence that L_1 is not solvable in polynomial time, and hence the reduction is effectively equivalent to saying "since L_1 is not likely to be solvable in polynomial time, then L_2 is also not likely to be solvable in polynomial time." Thus, this is how polynomial time reductions can be used to show that problems are as hard to solve as known difficult problems.

Summarizing the above, and recalling that the composition of poly-time functions is poly-time, we have the following.

Lemma: Given languages L_1, L_2, and L_3,

(i) If $L_1 \leq_P L_2$ and $L_2 \in P$, then $L_1 \in P$.
(ii) If $L_1 \leq_P L_2$ and $L_1 \notin P$, then $L_2 \notin P$.
(iii) If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$ then $L_1 \leq_P L_3$. (Transitivity of "$\leq_P$")

NP-completeness: We now have the necessary tools to define NP-completeness. This is subset of NP that are "hardest" in the sense that if it is known that if any one is solvable in polynomial time, then they all are. This is made mathematically rigorous using the notion of polynomial time reductions.

Definition: A language L is *NP-hard* if $L' \leq_P L$, for all $L' \in$ NP. (Note that L does not need to be in NP.)

Definition: A language L is *NP-complete* if:

(1) $L \in$ NP (that is, it can be verified in polymomial time), and
(2) L is NP-hard (that is, every problem in NP is polynomially reducible to it).

Unfortunately, showing that a problem is NP-hard seems nearly impossible, since it involves a property of *all* languages in NP, an infinite set. In the next lecture, we will present Cook's Theorem, which demonstrates that there does indeed exist an NP-complete problem.