Lecture 10

- The notion of a reduction.
- Cook- and Karp-reductions.
- $\mathcal{C}$-hard and $\mathcal{C}$-complete for a complexity class $\mathcal{C}$.

We now consider the issue of comparing two *problems* from the standpoint of computational difficulty. We re-emphasize that we are comparing problems, not algorithms. For example, is LongSimplePath computationally at least as difficult as ShortSimplePath? Is SAT computationally at least as difficult as Circuit-SAT, or SubsetSum?

Of course to be able to do this, we need to specify what we mean by "computationally more difficult." In this context, we now introduce the term "*hard*." "Hard" in our context refers to difficulty, i.e., instead of saying "computationally difficult," we will say "computationally hard." We do this because that is the customary terminology computing folks use in this context.

An approach Suppose, given a problem, it can only be one of "easy" or "hard." Also suppose we want to assert that problem B is at least as hard as problem A. Consider the following statement that we claim is equivalent:

$$B \textit{ is easy} \implies A \textit{ is easy.}$$

We argue that the above characterization of "problem B is at least as hard as problem A" is meaningful. As rationale, following are the possibilities that

the above assertion admits.

(1) A is easy and B is easy.
(2) A is hard and B is hard.
(3) A is easy and B is hard.

The possibility that it excludes is:

(4) A is hard and B is easy.

This is exactly what we want when we say that B is at least as hard as A.

Reduction If indeed it is true that B is easy implies A is easy, we say that there exists a *reduction* from A to B. We can meaningfully adopt the symbol "$\leq$" to represent this. That is, if A reduces to B, we write this as $A \leq B$. We will start to annotate the "$\leq$" with a subscript to further clarify what we mean by "reduces."

Two important things we emphasize:

1. This is a relative notion of "easy" and "hard" for two problems. That is, we are comparing the hardness of A with that of B only. We are saying nothing about whether A is hard, or whether B is hard. For example, B may be easy, and A may be hard. All we are saying is that in the former case, A must be easy as well, and in the latter case, B must be hard as well.

2. The direction of the reduction is important. B is easy implies A is easy is a reduction from A to B. Not the other way around. This is why I find the "$\leq$" notation useful. A reduction from A to B is written as $A \leq B$. It may or may not be the case that $B \leq A$ as well.

The notion of a reduction is fundamental and very important to practice in computing. And it is useful not only in the context of computational hardness. Suppose, for example, you have invented a lock for doors that you want to claim is at least as good as every other lock that exists in the market.

Then, you could say: the security of every other lock reduces to the security of my lock. In other words, if your lock is insecure, then so is every other lock. And if any of the other locks is secure, then so is yours.

One can use the same mindset for other properties that arise in computing as well, such as reliability and robustness. Indeed, the above reduction that corresponds to "at least as secure as" is commonly used, for example, to demonstrate that a new cryptographic scheme is "secure."

Similarly, we can use the notion of a reduction to show the existence, or non-existence, of an algorithm for a problem. Recall the problem which we can call HALT: given as input an encoding of an algorithm A and a string x, does A when run with input x halt? We proved early on in the course that no algorithm exists for HALT.

Now suppose you have some problem p, and you want to prove that no algorithm exists for it. One way is to invent a reduction that corresponds to algorithm-existence, and reduce HALT under that reduction to p. This would establish that if there exists an algorithm for p, then there exists an algorithm for HALT, which in turn proves that no algorithm exists for p.

On the other hand, suppose I want to prove that there does indeed exist an algorithm for my problem p. Then, under the notion of a reduction that corresponds to algorithm-existence, if I reduce p to some problem q for which we know an algorithm exists, then I have shown that there exists an algorithm for p.

The Cook reduction

The Cook reduction, so called after Stephen Cook of U of T, is a particular kind of reduction to compare the computational hardness of two decision problems. In the Cook reduction, we premise that "easy" means "in polynomial-time." As we are constraining ourselves to decision problems only, "in polynomial-time" is the same as "$\in \mathbf{P}$."

Definition 12 (Cook reduction)**.** *Given two decision problems, A, B, we say*

that A Cook-reduces to B, denoted $A \leq_c B$ if and only if:

$$B \in \boldsymbol{P} \implies A \in \boldsymbol{P}$$

For example, consider the following two versions of the Hamiltonian path problem.

HamPath: given as input a non-empty undirected graph G, is there is a simple path in G of all its vertices?

HamPathStartEnd: given as input a non-empty undirected graph G and two vertices in it a, b with $a \neq b$, is there a simple path in G of all its vertices that starts at a and ends at b?

Claim 50. HamPath $\leq_c$ HamPathStartEnd

Proof. By construction. We assume that there exists a polynomial-time algorithm, call it H_{SE}, for HamPathStartEnd. We propose the following algorithm, H, for HamPath. The idea behind the algorithm is that G has a Hamiltonian path if and only if it has a Hamiltonian path that starts at one of its vertices and ends at another.

```
H(G = ⟨V, E⟩)
1 if |V| = 1 then return true
2 foreach u ∈ V do
3     foreach v ∈ V \ {u} do
4         if H_SE(G, u, v) = true then return true
5 return false
```

H is correct and terminates if H_{SE} is correct and terminates. If H_{SE} runs in time $O(n^c)$ for input size n and some constant c, then H runs in time $O(n^2 \times n^c) = O(n^{c+2})$, which is polynomial in the input size n. □

Claim 51. HamPathStartEnd $\leq_c$ HamPath

Proof. By construction. We assume that there exists a polynomial-time algorithm, call it H, for HamPath. We propose the following algorithm, H_{SE}, for HamPathStartEnd. We do this by forcing any Hamiltonian path that exists to have a particular starting and ending vertex.

$H_{SE}(G = \langle V, E\rangle, a, b)$
1 **if** $a = b$ *or* $|V| = 1$ **then return** false
2 Let $V' \leftarrow V \cup \{x, y\}$, where $x, y \notin V$
3 Let $E' \leftarrow E \cup \{\langle x, a\rangle, \langle b, y\rangle\}$
4 **return** $H(\langle V', E'\rangle)$ □

As another example, we revisit LongSimplePath.

LongSimplePath: given as input (i) an undirected graph $G = \langle V, E\rangle$, (ii) two vertices $a, b \in V$ with $a \neq b$, (iii) an integer $k \in [1, |V| - 1]$, does there exist a simple path $a \rightsquigarrow b$ of $\geq k$ edges?

Claim 52. HamPath $\leq_c$ LongSimplePath

Proof. We show that HamPathStartEnd $\leq_c$ LongSimplePath and the use the transitivity of $\leq_c$. That is, $A \leq_c B$ and $B \leq_c C$ implies that $A \leq_c C$. So, given that HamPath $\leq_c$ HamPathStartEnd, if HamPathStartEnd $\leq_c$ LongSimplePath, then HamPath $\leq_c$ LongSimplePath.

To show that HamPathStartEnd $\leq_c$ LongSimplePath, given input $\langle G, a, b\rangle$ to HamPathStartEnd, we invoke the algorithm for LongSimplePath with input $\langle G, a, b, |V| - 1\rangle$ and return whatever it returns. □

The above proof relies on the following claim.

Claim 53. *$A \leq_c B$ and $B \leq_c C \implies A \leq_c C$*

It turns out that LongSimplePath $\leq_c$ HamPath as well. However, the reduction in that direction is not necessarily straightforward. We discuss this issue of problems that have the same computational hardness, but seemingly different "expressive power" once we discuss more on the aspect of computational hardness.

We now ask: is there an example of where $A \leq_c B$ but $B \not\leq_c A$? The answer is 'yes.' Consider, for example, ShortSimplePath and Halt. It is easy to show that ShortSimplePath $\leq_c$ Halt because ShortSimplePath $\in \mathbf{P}$ immaterial of whether Halt $\in \mathbf{P}$ or not. However, Halt $\not\leq_c$ ShortSimplePath because if indeed such a reduction exists, then this would imply that there exists an algorithm for Halt which we know there is not.

We observe also that ShortSimplePath $\leq_c$ LongSimplePath because notwithstanding whether LongSimplePath $\in \mathbf{P}$ or $\notin \mathbf{P}$, we know that ShortSimplePath $\in \mathbf{P}$. An interesting question is whether a reduction holds in the other direction, i.e., whether LongSimplePath $\overset{?}{\leq}_c$ ShortSimplePath. We cannot assertively say 'no.' However, as we shall see soon, if indeed that is true, then $\mathbf{P} = \mathbf{NP}$, which would be shocking, and widely conjectured to be false. In other words, we will establish that $\mathbf{NP} \neq \mathbf{P}$ implies that LongSimplePath $\not\leq_c$ ShortSimplePath.

A limitation of $\leq_c$ Recall the notion of the complement of a decision problem, and denote the complement of a decision problem p as $\overline{p}$. Then:

Claim 54. *For every decision problem p, $p \leq_c \overline{p}$*

The proof for the above claim is quite straightforward. Suppose we have a polynomial-time algorithm for $\overline{p}$. Then for an algorithm for p, simply invoke the algorithm for $\overline{p}$, and return the complement of what it returns. Thus, we have a polynomial-time algorithm for p.

Now, we recall the notion of *closure* for a set under an operation.

Definition 13. *We say that a set S is closed under a binary operation $\boxplus$ if, given any $e_1, e_2 \in S$, it is the case that $e_1 \boxplus e_2 \in S$.*

As an example, the set of real numbers, $\mathbb{R}$, is closed under the addition operation, $+$. So is the set of integers, and the set of natural numbers. However, while the set of integers is closed under subtraction, the set of natural numbers is not. We now specifically define closure for a complexity class under a reduction.

Definition 14. *We say that a complexity class $\mathcal{C}$ is closed under a reduction $\leq$ if, given any two decision problems p_1, p_2, the following is true:*

$$p_1 \leq p_2 \text{ and } p_2 \in \mathcal{C} \implies p_1 \in \mathcal{C}$$

The mindset behind the above notion of closure for a reduction is the following. A complexity class, $\mathcal{C}$, characterizes an upper-bound on the inefficiency

of an algorithm that is needed for a problem. The notion of a reduction is also an upper-bound, but between two problems. The above notion of closure merely naturally relates the two: we can perceive $p_1 \leq p_2$ as stating that p_1 is upper-bounded by p_2, and $p_2 \in \mathcal{C}$ as stating that p_2 is upper-bounded by $\mathcal{C}$. The question is whether those imply that p_1 is upper-bounded by $\mathcal{C}$.

Claim 55. *If **NP** is closed under $\leq_c$, then so is **co-NP**.*

Proof. We need to prove: $p_1 \leq_c p_2 \wedge p_2 \in \textbf{co-NP} \implies p_1 \in \textbf{co-NP}$. Using $\bar{\cdot}$ to represent the complement of a decision problem, we know:

$$\begin{aligned} p_2 \in \textbf{co-NP} &\iff \overline{p_2} \in \textbf{NP} && \because \text{ definition of } \textbf{co-NP} \\ \text{and, } p_1 \leq_c p_2 &\iff \overline{p_1} \leq_c \overline{p_2} && \because \overline{p_1} \leq_c p_1 \leq_c p_2 \leq_c \overline{p_2} \end{aligned}$$

Thus, we have:

$$\begin{aligned} p_1 \leq_c p_2 \wedge p_2 \in \textbf{co-NP} &\iff \overline{p_1} \leq_c \overline{p_2} \wedge \overline{p_2} \in \textbf{NP} \\ &\implies \overline{p_1} \in \textbf{NP} && \because \textbf{ NP} \text{ closed under } \leq_c \\ &\iff p_1 \in \textbf{co-NP} \end{aligned}$$

□

We are now able to make the following claim, which expresses an undesirable property of the Cook-reduction, $\leq_c$.

Claim 56. *If **NP** is closed under $\leq_c$, then **NP** = **co-NP**.*

Before we get to the proof, we discuss why it is undesirable that **NP** = **co-NP**. The reason is that problems from the two classes do not seem similar to one another. Specifically, **NP** comprises those problems for which there is an efficiently sized evidence (a "certificate" or "witness") which we can verify efficiently. Problems in **co-NP**, on the other hand, appear to have no such feature.

Proof. (to Claim 56) We prove, under the premise that **NP** is closed under $\leq_c$, that (i) **co-NP** $\subseteq$ **NP**, and, (ii) **NP** $\subseteq$ **co-NP**.

To prove (i), we first recall from Claim 54 that for every decision problem p, it is true that $\overline{p} \leq_c p$. Therfore, for every $q \in$ **co-NP**, we know that $q \leq_c \overline{q}$ and $\overline{q} \in$ **NP**, and therefore if **NP** is closed under $\leq_c$, $q \in$ **NP**. Thus, **co-NP** $\subseteq$ **NP**.

To prove (ii), pick any $p \in$ **co-NP**. We know that $\overline{p} \leq_c p$, and therefore, using the same argument as above, if **co-NP** is closed under $\leq_c$, then **NP** $\subseteq$ **co-NP**. But we know, from Claim 55 that if **NP** is closed under $\leq_c$, then so is **co-NP**. Therefore, if **NP** is closed under $\leq_c$, then **NP** $\subseteq$ **co-NP**. $\square$

Another way to express Claim 56 is to state its contrapositive.

> *If* ***NP*** $\neq$ ***co-NP****, then* ***NP*** *is not closed under* $\leq_c$*.*

We have already argued for why the consequence of Claim 56 is undesirable, based on the seeming separation between **NP** and **co-NP**. This contrapositive suggests a different kind of undesirability in its consequence. To say that $p \in$ **NP** is to upper-bound the inefficiency of an algorithm for p. To say also that $p \leq_c q$ also upper-bounds that inefficiency. Now, to say that a class, **NP** in this case, is not closed for a notion of $\leq$, $\leq_c$ in this case, seems undesirable, because it seems that both those notions should coincide.

The Karp reduction

We now discuss another kind of reduction which we call the Karp reduction, named after Richard Karp. We denote it as $\leq_k$.

Definition 15 (Karp reduction)**.** *Given two decision problems A, B, suppose the set of all instances of problem A is denoted I_A and the set of all instances of problem B is denoted I_B. We say that A Karp-reduces to B, written $A \leq_k B$ if and only if:*

> *there exists a polynomial-time computable function $m : I_A \to I_B$ such that $i \in I_A$ is a* true *instance of problem A if and only if $m(i) \in I_B$ is a* true *instance of problem B.*

For examples, we return to HamPathStartEnd and HamPath. In particular, we point out that the Cook reduction we propose in the proof for Claim 51 is a Karp reduction. But the Cook reduction in the proof for Claim 50 is not a Karp reduction. To be more precise, we state and prove Karp reductions between the two problems in the following claim.

Claim 57. HamPathStartEnd $\leq_k$ HamPath

Proof. Recall that an instance, whether true or false, of HamPathStartEnd is a tuple $\langle G, a, b\rangle$. And an instance of HamPath is $\langle G\rangle$. We prove the claim by construction. That is, we propose a function m from instances of HamPathStartEnd to instances of HamPath and establish that it possesses the required properties.

Such an m is as follows. Given $\langle G = \langle V, E\rangle, a, b\rangle$, we introduce two new vertices, $x, y \notin V$. Let the resultant set of vertices be V'. We introduce two new edges $\langle x, a\rangle$ and $\langle y, b\rangle$. Let the resultant set of edges be E'. Let $G' = \langle V', E'\rangle$. We now claim: (i) this m is polynomial-time computable, and, (ii) $\langle G, a, b\rangle$ is a true instance of HamPathStartEnd if and only if $\langle G'\rangle$ as produced from $\langle G, a, b\rangle$ as specified above is a true instance of HamPath.

Towards (i), we observe that we add a constant number of verices and edges only. So to compute m takes time $O(n)$, where n is the size of the input, $\langle G, a, b\rangle$. Towards (ii), for the "only if" direction, suppose $\langle G, a, b\rangle$ is a true instance of HamPathStartEnd. This means that there is a Hamiltonian path $a \rightsquigarrow b$ in G. Therefore, $x \rightsquigarrow y$ is a Hamiltonian path in G'. That is, $\langle G'\rangle$ is a true instance of HamPath as desired.

For the "if" direction of (ii), we prove the contrapositive. Suppose no Hamiltonian path $a \rightsquigarrow b$ exists in G, and for the purpose of contradiction, assume a Hamiltonian path exists in G'. Then, as the degree of each of x, y in V' is 1, any Hamiltonian path in G' must have x, y as the start and end vertices. This means that the Hamiltonian path must be of the form $x \to a \rightsquigarrow b \to y$. This implies that there is a Hamiltonian path $a \rightsquigarrow b$ in G, a contradiction. □

Claim 58. HamPath $\leq_k$ HamPathStartEnd

Proof. By construction. We propose the following mapping, m. Given an instance $\langle G = \langle V, E\rangle\rangle$ of HamPath, introduce two new vertices $x, y \notin V$.

Let the resultant set be $V' = V \cup \{x, y\}$. For every $u \in V$, introduce two new edges, $\langle x, u\rangle, \langle y, u\rangle$. Let the resultant edge-set be E'. Let $G' = \langle V', E'\rangle$. The function m maps $\langle G\rangle$ to $\langle G', x, y\rangle$.

We first observe that m is computable in time polynomial (linear) in the size of G. We claim also that G has a Hamiltonian path if and only if G' has a Hamiltonian path $x \rightsquigarrow y$ in G'.

For the "only if" direction, suppose G has a Hamiltonian path $p \rightsquigarrow q$. Then $x \to p \rightsquigarrow q \to y$ is a Hamiltonian path in G'. For the "if," suppose G has no Hamiltonian path, and for the purpose of contradiction, G' has a Hamiltonian path $x \rightsquigarrow y$. Then, such a path in G' is of the form $x \to p \rightsquigarrow q \to y$ for some $p, q \in V$, and the edges in $p \rightsquigarrow q$ are in E. Thus, G has a Hamiltonian path $p \rightsquigarrow q$, which is a contradiction. □

We have several more examples of reductions in the next lecture. For now, we present two more relatively straightforward examples. To prove that HamPathStartEnd $\leq_k$ LongSimplePath, we can simply adopt a function that maps $\langle\langle V, E\rangle, a, b\rangle$ to $\langle\langle V, E\rangle, a, b, |V| - 1\rangle$.

To prove that ShortSimplePath $\leq_k$ LongSimplePath, we can exploit the fact that ShortSimplePath $\in \mathbf{P}$. So we, for example, simply "pre-prepare" two instances of LongSimplePath, one of which is true, which we call i_t, and the other false, which we call i_f. E.g., the former could simply be $G = \langle V, E\rangle$ where $V = \{a, b\}$ and $E = \{\langle a, b\rangle\}$, the vertices a, b, and $k = 1$. The latter could simply be $V = \{a, b\}$, $E = \emptyset$, the vertices a, b and $k = 1$.

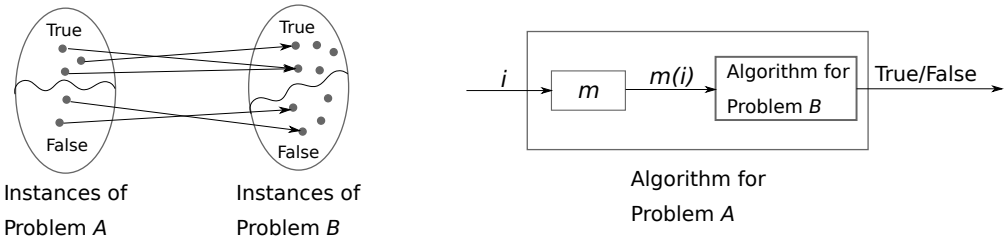
Then, every $\langle G, a, b, k\rangle$ that is a true instance of ShortSimplePath, we map to i_t and every such instance that is false we map to i_f. This mapping is polynomial-time computable because given $\langle G, a, b, k\rangle$, we can determine in polynomial-time whether it is a true or false instance of ShortSimplePath. And then simply output i_t or i_f, respectively.

The above example calls to mind another name by which the Karp reduction is known: *polynomial-time, many-to-one* reduction. The "many-to-one" part of that name emphasizes the fact that the function m is required to be many-to-one only. In our example of reducing ShortSimplePath to LongSimplePath above, all true instances of ShortSimplePath (of course there are infinitely many) are mapped to the same instance of LongSimplePath, and

all false instances are mapped to the same instance as well. This is certainly not onto: there are infinitely many true and false instances of LongSimplePath that are not in the range of this function.

This many-to-one aspect impacts our proofs for the "if and only if" property of the mapping function m. When we reduce $A \leq_k B$, and we propose some $m\colon I_A \to I_B$ as the mapping, we need to consider the range of m only in our proof that $i \in I_A$ is a true instance if and only if $m(i) \in I_B$ is a true instance. We do not have to consider the entire set I_B.

The following picture shows pictorially how the Karp reduction works to the left, and a consequence of it to the right. The picture to the left shows that true instances are mapped to true instances, and false to false instances, and that the mapping is required to be many-to-one only. The picture to the right shows, if we have a reduction-mapping m for $A \leq_k B$, the manner in which we can use an algorithm for B to decide an instance, i, of A. We first compute $m(i)$, which is an instance of B, and then invoke the algorithm for B. We can immediately return whatever that algorithm for B returns for $m(i)$, because of the "if and only if" property of m.



The figure to the right above underlies the proof for the following claim.

Claim 59. $p \leq_k q \implies p \leq_c q$

We now state and prove the following claims, which suggest that the Karp reduction does not have the same weakness the Cook reduction.

Claim 60. ***NP*** *is closed under* $\leq_k$*. That is, if* $B \in$ ***NP*** *and* $A \leq_k B$*, then* $A \in$ ***NP****.*

Proof. Suppose L_B is a non-deterministic polynomial-time algorithm for B that runs in time $O(n^{c_1})$ on an input of size n. Following is a non-deterministic

polynomial-time algorithm for A, call it L_A. On some input i, which is an instance of A, L_A first computes $m(i)$, where m is the mapped with the "if and only if" property that underlies the reduction $A \leq_k B$. Recall that $m(i)$ can be computed in time n^{c_2} for input of size n and some constant c_2, and $m(i)$ is an instance of problem B. L_A then invokes L_B with input $m(i)$ and outputs whatever it outputs.

We claim that L_A is correct and runs in polynomial-time. To establish the former, suppose i is a true instance of A. Then $m(i)$ is a true instance of B and we know that there exists a sequence of non-deterministic choices L_B makes that causes it to output true. Suppose i is a false instance of A. Then $m(i)$ is a false instance of B and L_B is guaranteed to output false on input $m(i)$. Thus, L_A is correct.

To establish that L_A runs in polynomial-time, suppose $|i| = n$. Then $m(i)$ takes time $O(n^{c_1})$ to compute, for some constant c_1. Furthermore, $|m(i)| = O(n^{c_1})$ because the output of an algorithm cannot have size larger than its running-time. L_A then runs L_B with input $m(i)$, whose running time is thus $O((n^{c_1})^{c_2}) = O(n^{c_1 \cdot c_2})$, for some constant c_2. Thus, the total running-time of L_A is $O(n^{c_1} + n^{c_1 \cdot c_2})$, which is polynomial in n. □

Claim 61. ***co-NP*** *is closed under* $\leq_k$*. That is, if* $B \in$ ***co-NP*** *and* $A \leq_k B$*, then* $A \in$ ***co-NP****.*

Thus, the Karp reduction does not have any seemingly undesirable consequences, like the Cook reduction does. We observe also that the converse of Claim 59 is unlikely to be true. That is, we think that the Karp reduction is a tighter notion than a Cook reduction.

$\mathcal{C}$-hard and $\mathcal{C}$-complete

When we say that a decision problem belongs to a complexity class, that is an expression of an upper-bound for the hardness of the problem. For example, if a decision problem $p \in \mathbf{NP}$, that expresses that p is no harder than **NP**. It says nothing about the lower-bound hardness of p. For example, there may exist a polynomial-, or even constant-time algorithm for p.

We now provide a notion of a lower-bound hardness relative to a complexity class for a problem, by leveraging the notion of a reduction.

Definition 16 ($\mathcal{C}$-hard under $\leq$)**.** *We say that a decision problem p is $\mathcal{C}$-hard for a complexity class $\mathcal{C}$ under a reduction $\leq$ if:*

$$\textit{for all } q \in \mathcal{C},\ q \leq p$$

The point of the above definition is that p is at least as computationally hard as every problem in $\mathcal{C}$. Where "at least as computationally hard" is as specified by a particular reduction that we choose. For example, we could choose $\leq_c$ or $\leq_k$ as our choice of reduction. This is why we add the qualification "under a reduction."

We could now replace the place-holder complexity class $\mathcal{C}$ with, for example, **NP**, and we have a notion of **NP**-hard.

We can certainly say "**NP**-hard under $\leq_c$." However, when folks simply say "**NP**-hard" without specifying under what reduction, they mean under $\leq_k$. Henceforth, unless we specify otherwise, we adopt $\leq_k$ as our notion of reduction.

Note: CLRS uses $\leq_p$ for $\leq_k$.

We now repeat Definition 16 for the specific case that $\mathcal{C} = \mathbf{NP}$, and keep the kind of reduction implicit.

Definition 17 (**NP**-hard)**.** *We say that a problem p is **NP**-hard if for all $q \in \boldsymbol{NP}$, $q \leq_k p$.*

Thus, if p is **NP**-hard, this is an expression of a *lower-bound* hardness for the problem p, with the basis a complexity class. It is saying that p is at least as hard as every problem in **NP**, or, equivalently, no easier than any problem in **NP**.

We now introduce the notion of $\mathcal{C}$-complete, for a complexity class $\mathcal{C}$, which combines an upper- and a lower-bound.

Definition 18 ($\mathcal{C}$-complete)**.** *We say that a problem p is $\mathcal{C}$-complete for a complexity class $\mathcal{C}$ under a reduction $\leq$ if:*

1. $p \in \mathcal{C}$, *and,*
2. p *is* $\mathcal{C}$*-hard under* $\leq$.

The above notion of completeness for a class reminds us that a reduction is "less than or equals," and not "strictly less than." That is, it is possible that p is both in $\mathcal{C}$, and $\mathcal{C}$-hard. Following is our specialization of the above definition to the class **NP**. We again leave the reduction implicit.

Definition 19 (**NP**-complete)**.** *We say that a problem p is **NP**-complete if:*

1. $p \in$ ***NP***, *and,*
2. p *is* ***NP****-hard.*

In the next lecture, we will look at examples of problems that are **NP**-complete. We now, we carry on with our discussions on the above notions.

Do there exist problems that are **NP**-hard and not **NP**-complete? The answer is 'yes.' For example, HALT is **NP**-hard. But it is not in **NP**, and therefore it is not **NP**-complete. We now make more observations that are useful to understand the nature of problems that are **NP**-complete.

Claim 62. *If $p \leq_k q$ and $q \leq_k r$ then $p \leq_k r$.*

Claim 63. *If problem p is **NP**-hard and $p \leq_k q$, then q is **NP**-hard.*

Claim 64. *If there exists $q \in$ **NP**-complete such that $q \in$ **P**, then **P** = **NP**.*

Proof. We already know that $\mathbf{P} \subseteq \mathbf{NP}$. So all we need to prove is that $\mathbf{NP} \subseteq \mathbf{P}$ under the given conditions. Suppose such a q exists. Then, given any $p \in \mathbf{NP}$, we know that $p \leq_k q$, because q is **NP**-hard. But $q \in \mathbf{P}$. So, a polynomial-time algorithm for p would simply compute $m(i)$ on input i which is an instance of the problem p, where m is the function that underlies the reduction $p \leq_k q$, and then invoke the polynomial-time algorithm for q with input $m(i)$. □

The idea with **NP**-complete is that a problem that is **NP**-complete is one of the hardest problems in **NP**. That is, it is hard, i.e., **NP**-hard, but not too

hard, i.e., still remains in **NP**. More generally, the qualifier "-complete" to a complexity class refers to the hardest problems in that complexity class, where "hard" refers to a particular reduction that we adopt. For example, the following claim suggests that $\leq_k$ is not necessarily meaningful as a notion of "hard" to compare problems that are in **P**.

Claim 65. *Suppose q is a decision problem that is: (a) in **P**, and, (b) has at least one* true *instance and at least one* false *instance. Then q is **P**-hard under $\leq_k$.*

The above claims suggests that under $\leq_k$, *almost* every problem that is in **P** is **P**-hard. What are the problems in **P** that are not **P**-hard under $\leq_k$? The answer is: the following two problems only. (i) Given as input any string, including the empty string, map it to true, and, (ii) given as input any string, including the empty string, map it to false.

Note that the problems (i) and (ii) above do not meet the pre-condition (b) in the statement of Claim 65. They do meet pre-condition (a), i.e., each of them is indeed in **P**. An algorithm for problem (i) would simply return true on any input, and an algorithm for (ii) would simply return false on any input. Thus, each runs in constant-time and is therefore $\in$ **P**.

A more meaningful way to compare problems in **P** may be to adopt $\leq$ to be a log-space reduction; that is, the mapping that underlies the reduction is required to be computable with an algorithm that is allowed auxiliary space only logarithmic in the size of the input.