Complexity Theory and Reductions

Suhas Arehalli

COMP221 - Spring 2025

So far, we've talked about analyzing the time complexity of *algorithms*: Once I have a solution to a problem, I want to figure out how fast that solution is. However, when *designing* an algorithm, it's almost always useful to know whether algorithms of a particular time complexity is *possible*. A little more flippantly, a designer of algorithms will often ask: **Are there no faster solutions to this problem, or am I just bad at algorithmic design?**

Amongst other things, a theory computational complexity theory will help us answer this question by giving us a way to talk about the (relative) difficulty of *problems*, with difficulty defined in terms of the time complexity of the best solutions to the problem.

In an ideal world, we would be able to develop techniques to prove formally that fast solutions don't exist for a problem we're working on. This would allow us to throw in the towel on finding a fast solution and begin to apply some heuristic search techniques or start looking toward approximate solutions. Unfortunately, the world is not (yet) ideal, and we will instead have to settle for a weaker kind of claim — that as of now no algorithms researcher has been able to find a fast algorithm, and that a $1,000,000 prize lies in wait for the first person to find one.

1 Preliminaries

As the introduction stated, we are building a theory of the difficulty of a *problem*, and we say a problem's difficulty is measured in terms of how fast it can be solved. However, we note that it is really hard to prove that a faster solution *doesn't* exist — there really aren't many generalizable techniques we have to show that!

As a result, we're actually going to develop a theory of *relative* difficulty. We want to show that a problem is just as hard, if not harder than another. This, it turns out, is going to be something that is not too hard to demonstrate in practice!

In order to make our lives easier, we're also going to restrict the *kind* of problem we consider when doing reductions. Many problems we've looked at so far are what we call *search problems* (i.e., find an object that satisfies some conditions) or *optimization problems* (i.e., find the *best* object that satisfies some conditions). These are interesting, but because they all deal with different kinds of objects (paths, trees, arrays, sets, integers, etc.), they're hard to relate. Instead, for the purposes of complexity theory, we'll analyze *decision problems*, problems whose answer is either true or false. As you read the next section, consider how a theory of reductions as we describe it would be overcomplicated if we considered functions with non-boolean outputs. Rest assured, decision problems can be made to capture the richness of corresponding search and optimization problems, as we'll see.

The last preliminary is in regard to how we define difficulty. For reasons that we aren't able to get into in this course, it is conventional to think about broader time complexity classes than we do when analyzing particular algorithms. Rather than distinguishing between algorithms that run in $O(n)$, $O(n \log n)$, $O(n^2)$, and so on, we tend to think of algorithms in terms of those that run in **polynomial time** (i.e., $O(n^k)$ for some k) and those that don't. We typically call polynomial time algorithms **tractable** to indicate that polynomial time solution are those that are practically usable on large problem instances.

2 Reductions

Recall that we'd like to claim that some problem instance, ProbA is as hard, if not harder than ProbB. Since we've decided to say a problem is hard if it's not solvable in polynomial time, what we mean formally is something along the lines of "If ProbB is not solvable in polynomial time, then ProbA is not solvable in polynomial time."

It turns out we can make this claim via a proof technique called a **Reduction**. The specific kind of reduction we consider here is called a Karp Reduction, or a polynomial-time Many-to-One reduction, though the distinction is not critical for this class, where this is the only kind of reduction we'll see.

Def. We say that ProbA **reduces to** ProbB when we can write an algorithm A that solves ProbA that can be written as follows:

function A(X)
    Convert X to a problem instance of ProbB, Y, in polynomial time
    return B(Y)
end function

where B is *any* algorithm that solves problem B.

Now we can make 2 claims:

Statement 1 (Easiness Reduction)**.** If ProbA reduces to ProbB and ProbB can be solved in polynomial time, then ProbA can be solved in polynomial time

Proof. Let B be a polynomial time algorithm that solves ProbB. Construct A as in the definition of the reduction. Since B and the conversion both run in polynomial time, A runs in polynomial time. Since A solves ProbA, ProbA can be solved in polynomial time. □

This result follows from our standard algorithmic analysis procedures: The idea of a reduction lets us construct a solution to ProbA from any solution to ProbB, and so, given a polynomial time solution to ProbB, we apply that template and build a polynomial time solution to ProbA. Here's a second claim:

Statement 2 (Hardness Reduction)**.** If ProbA reduces to ProbB and ProbA cannot be solved in polynomial time, then ProbA cannot be solved in polynomial time

This one is a bit more tricky to understand, but actually follows directly from the first statement!

Proof. Assume for contradiction that ProbB could be solved in polynomial time. By the prior statement, ProbA can be solved in polynomial time. This contradicts what we were given, thus ProbB cannot be solved in polynomial time. □

And now we finally have the tools necessary to prove relative difficulty! We can say that ProbB is *as hard, if not harder* than ProbA if ProbA reduces to ProbB. Equivalently, we can say that ProbA is *as easy or easier* that ProbB if ProbA reduces to ProbB.

Be careful about the direction here. We will really only consider hardness reductions in this course, and it is easy to mistakenly reduced the wrong direction. My claim is that you should, at least for the first handful of problems you solve, walk through the full logic of the reduction rather than trying to remember the direction you should reduce the two problems. My claim: If you think through the full logic, the direction should follow from the time complexity analysis reasoning we've developed!

3 Sample Reductions:

3.1 Traveling Salesperson Problem (TSP) and Hamiltonian Cycle

Suppose we believe that Hamiltonian Cycle is a hard problem (and it is!). We'd like to show TSP is just as hard, if not harder. That is, if we can't solve Hamiltonian Cycle in polynomial time, we can't solve TSP in polynomial time. Before we show this, let's define the relevant problems:

Problem Statement (TSP)**.**
Input: A complete[a] graph $G = (V, E)$, a cost function $c : E \to \mathbb{R}$, and $k \in \mathbb{Z}$
Output: TRUE iff there exists a cycle that contains every vertex in V with total cost $\leq k$.

[a] TSP is sometimes formulated as requiring a complete graph, though we can see that this is equivalent to any other formulation by just setting the cost of edges we don't want to matter arbitrarily high.

Problem Statement (Hamiltonian Cycle)**.**
Input: A graph $G = (V, E)$
Output: TRUE iff there exists a simple cycle that contains every vertex in V.

Statement 3. TSP is as hard, if not harder, than Hamiltonian Cycle.

Proof. We reduce Hamiltonian Cycle to TSP. Consider, for a TSP solver TSP, the following algorithm:

function HamCycle($(G = (V, E))$
  Let $V' = V$
  Let $E' = \{(v, w) \mid v, w \in V, v \neq w\}$
  Let $c(v, w) = \begin{cases} 1, & (v, w) \in E \\ 2, & otherwise \end{cases}$
  return TSP($G' = (V', E')$, $|V|$)
end function

First, we show that the conversion runs in polynomial time. This is simple: conversion is dominated by constructing c and E', both of which require $|V|^2$ time.

Next, we need to show that HamCycle actually solves the Hamiltonian Cycle problem. To do this formally, we proceed by cases:

Case 1: Suppose G does contain a Hamiltonian Cycle. We must show that HamCycle returns TRUE. That is, we must show G' contains a TSP Cycle of cost $\leq k$. To see that, we consider the edges of the Hamiltonian Cycle in G once translated to G'. Since G' is complete, the edges are present, and since V is unchanged, it is still a cycle that visits every vertex. Now we consider it's cost under c: Since the cycle was in G, each of it's edges has cost 1, and since it's a cycle containing each vertex in V, its cost is $|V|$! Thus TSP(G', $|V|$) must return true, as desired!

Case 2: Suppose HamCycle returns TRUE. Then TSP(G', $|V|$) must have returned true. Now consider the cycle the TSP solver found. Since it visits every vertex, it must have $\geq |V|$ edges and since edges have cost either 1 or 2, it must contain exactly $|V|$ edges of exactly cost 1. Therefore, every edge must be in G, and thus we have a a simple cycle that visits every vertex (a Hamiltonian Cycle!) in G, as desired.

Thus, Hamiltonian Cycle reduces to TSP! □

Now, in practice, I will rarely ask for this level of detail and formality in demonstrating the correctness of a reduction, but I *will* ask for at least a few sentences justifying why your reduction is both correct and polynomial time.

3.2 Vertex Cover and Independent Set

Consider the following problems:

Problem Statement (Vertex Cover)**.**
Input: A graph $G = (V, E)$ and $k \in \mathbb{Z}$
Output: TRUE iff there exists a subset $C \subseteq V$ with $|V| \leq k$ such that $\forall (v, w) \in E$, $v \in C$ or $w \in C$.

Problem Statement (Independent Set)**.**
Input: A graph $G = (V, E)$ and $k \in \mathbb{Z}$
Output: TRUE iff there exists a subset $V' \in V$ with $|V'| \geq k$ such that $\forall v, w \in V'$, $(v, w) \notin E$.

Make sure you read the notation carefully: A vertex cover is a set of vertices that *cover* (i.e., is incident to) each edge in the graph. An independent set is set of vertices such that no pair of vertices within the set are adjacent. These problems want to determine whether there are sufficiently small or sufficiently large sets of this sort respectively. One should be able to imagine the optimization variants of these problem: Find the smallest vertex cover or largest independent set within a graph.

Now to the claim:

Statement 4. Vertex Cover and Independent Set are equally hard.

Note that to show things are equally hard, we'll have to reduce both directions! Now, luckily, this isn't two hard once we realize how to translate between instances of both problems.

By definition, for C to be a set cover of a graph $G = (V, E)$, every edge $(v, w) \in E$ must have either v or w in C. Now consider without loss of generality some arbitrary vertices $x, y \in V \setminus C$. Since neither x or y is in C and C is a set cover, we know that $(x, y) \notin E$. Thus $V \setminus C$ is an independent set! This is a brief proof of the following statement:

Statement 5. For a graph $G = (V, E)$, $C \subseteq V$ is a set cover iff $V \setminus C$ is an independent set.

Which, if we consider those set's sizes, gets us to a corollary:

Corollary 1. For a graph $G = (V, E)$, a set of cover of size k exists iff an independent set of size $|V| - k$ exists.

And now to our proof of Statement **??**:

Proof. Consider the following reductions, where VS, as solution to the Vertex Cover problem, assumes IS is some solution to the Independent Set problem and vice versa:

function VC($G = (V, E)$, k)
  return IS(G, $|V| - k$)
end function

and

function IS($G = (V, E)$, k)
  return VC(G, $|V| - k$)
end function

Correctness of both algorithms follows for Corollary **??**, and subtraction can certainly be done in polynomial time. □

3.3 Independent Set and Clique

Consider the Clique problem:

Problem Statement (Clique)**.**
Input: A graph G and $k \in \mathbb{Z}$
Output: TRUE iff there exists a subset $C \subseteq V$ with $|V| = k$ such that $\forall v, w \in C$, $(v, w) \in E$

To show this is equivalent to Independent Set, we make another observation about the relationship between these problems. However, now we have to consider a small transformation of the graph G:

Consider the *complement* of G, $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(v, w) \mid v, w \in V, v \neq w, (v, w) \notin E\}$. This graph has an edge between $v, w \in V$ iff no such edge exists in G. Thus, if we have an independent set of size k in G, it becomes a clique in $\overline{G}$! This works in reverse as well: If we have a clique of size k in G, those vertices form an independent set in $\overline{G}$. This is, again, a brief proof of the following claim:

Statement 6. There is a clique of size k in G iff there is an independent set of size k in $\overline{G}$.

Now, one thing left to settle is that the Clique problem asks for a clique of size *exactly* k, while Independent Set asks just for an independent set of size $\geq k$. This is solved by simply observing that every subset of a clique is also a clique, so if we have a clique of size $> k$, we can simply take a subset of size k that is guaranteed to also be a clique. Thus we can extend this statement to say

Statement 7. There is a clique of size $\geq k$ in G iff there is an independent set of size $\geq k$ in $\overline{G}$.

Now we get to our big statement:

Statement 8. Clique and Independent Set are equally hard

Proof. We do this by reducing in both directions. Since we've seen this a few times, I'll be brief. Consider the following reductions:

function CLIQUE($G = (V, E)$, k)
  $\overline{E} = \{(v, w) \mid v, w \in V, v \neq w, (v, w) \notin E\}$
  return IS($\overline{G} = (V, \overline{E})$, k)
end function

and

function IS($G = (V, E)$, k)
  $\overline{E} = \{(v, w) \mid v, w \in V, v \neq w, (v, w) \notin E\}$
  return CLIQUE($\overline{G} = (V, \overline{E})$, k)
end function

The conversion involves simply iterating through pairs of edges in V^2, and testing set-membership ($O(1)$), so constructing $\overline{E}$ will take polynomial time.

Correctness for CLIQUE follows from Statement **??**. To see correctness for IS, observe that $\overline{\overline{G}} = G$ and apply Statement **??**. □

4 Satisfiability and Cook-Levin

Up to this point we've built a theory of *relative* difficulty, so we can only talk about problems being *as difficult* as others. However, we want to say that problems, in isolation, are hard. Luckily, our theory of reductions will get us there if we're able to find at least one problem we can confidently call hard — that problem is **Satisfiability**, or **SAT**.

As I mentioned in the intro, we won't quite get to where we want to go, but we'll get pretty close. a lot of this has to do with re-evaluating what we mean when we say a problem is hard. So far, we've said that easy problems are those that are solvable in polynomial time, but we've said little about what problems we think of as hard. That'll be formalized in the next section!

4.1 (Boolean) Satisfiability

To understand this problem, we'll introduce a few terms about boolean formulas:

Def. A **Boolean variable** x is a variable that can be *assigned* a value of TRUE/T or FALSE/F. A **literal** l is either a boolean variable x , or the negation of a variable $\overline{x}$ (if x is T, then $\overline{x}$ is F and vice versa). A **clause** C of size k is a disjunction (logical OR) of literals, $C = (l_1 \vee l_2 \vee \ldots l_k)$.

Def. A **Boolean formula** ϕ is a combination of ANDs, ORs, and NOTs applied to a set of Boolean variables. A boolean formula is in **conjunctive normal form** (CNF) iff it is written as a conjunction (logical AND) of clauses.

For example, a Boolean formula ϕ over variables x_1, x_2 can be written like $\phi = (x_1 \wedge \overline{x_2}) \vee \overline{(x_2 \vee x_1)}$. A CNF boolean formula would instead look something like $\phi = (x_1 \vee x_2) \wedge (\overline{x_2})$, with clauses $(x_1 \vee x_2)$ and $(\overline{x_2})$, and using literals x_1, x_2, and $\overline{x_2}$.

For our purposes, it's also useful to keep in mind a result from formal logic about CNF:

Statement 9. Every Boolean formula is logically equivalent to some Boolean formula in CNF.

Actually converting between CNF and non-CNF formulas is not of interest to us here, but knowing we can convert will be useful in understanding how the satisfiability problem is defined. Now we're finally ready to define what the problem is about:

Def. A boolean formula ϕ over a set of variables V is **satisfiable** iff there exists an assignment of T and F to the variables in V such that ϕ evaluates to true.

For example, if we have $\phi = (x_1 \vee x_2) \wedge (\overline{x_2})$, then the assignment $x_1 = T$ and $x_2 = F$ will have $\phi = T$. Thus, $x_1 = T$ and $x_2 = F$ is a **satisfying assignment**, and ϕ is satisfiable. However, not every boolean formula is satisfiable: Consider $\phi = x_1 \wedge \overline{x_1}$ — no assignment of T or F to x_1 will have $\phi = T$!

Now to the satisfiability problem:

Problem Statement (Boolean Satisfiability/SAT)**.**
Input: A set of variables V and a set C of clauses over V
Output: TRUE iff there is an assignment of T or F to variables in V such that at least one literal is true in each clause in C.

This is a strange formulation until you realize that this is just a reflection of the way in which we encode a Boolean formula. Since every Boolean formula is equivalent to one in CNF, we can represent our formula as a set of clauses. Since a formula in CNF is a logical AND over clauses, we need every clause to evaluate to true. In this problem, we represent clauses as sets of literals, and since a clause is a logical OR, at least one of literals needs to be true for the clause to be true. Thus, we arrive at "at least one literal in each clause in C is true" meaning that the corresponding Boolean formula in CNF evaluates to T.

We can simplify what's happening conceptually by formulating SAT as follows:

Problem Statement (Boolean Satisfiability/SAT — Alternative)**.**
Input: A set of variables V and a CNF Boolean formula ϕ
Output: TRUE iff there is an assignment of T or F to variables in V such that $\phi = T$.

but by convention, the first is the more common framing (and easier to work with when reducing to other problems) and so we'll stick to it.

Here is where learning about CNF helps: Since every boolean formula can be written in CNF, we're essentially able to test the satisfiability of boolean formulas generally!

We introduce SAT partially because it's interesting in it's own right, but mostly because it is, for historical reasons, **THE** hard problem in complexity theory. Now, we haven't yet formalized our notion of hardness, but for now, trust that SAT is hard.

4.2 3-SAT

Since SAT itself is kind-of hard to work with, we introduce a simpler variant that is just as hard: 3-SAT.

Problem Statement (3-SAT)**.**
Input: A set of variables V and a set C of clauses of size 3 over V
Output: TRUE iff there is an assignment of T or F to variables in V such that at least one

literal is true in each clause in C.

Note that 3-SAT seems, on it's face, easier. Certainly, restricting the input to clauses of exactly size 3 can't make the problem hard. And, additionally, reducing 3-SAT to SAT is trivial, since every instance of 3-SAT is just an instance of SAT. However, we will claim that they are actually equally difficult: If we can solve 3-SAT fast, we can solve SAT fast.

Statement 10. SAT and 3-SAT are equally hard.

Proof. We will, for brevity, only reduce SAT to 3-SAT here. Additionally, I'll present a common shorthand for proofs with SAT: I'll show you how to convert each clause of SAT into a (bounded) set of clauses in 3-SAT that are satisfiable iff the original clause is satisfiable.

Consider an instance of SAT with variables V and clauses C. We will construct a set of size-3 clauses C' over variables V' that are equivalent in terms of satisfiability. Let $C_i = \{z_1, \ldots, z_k\} \in C$ be an arbitrary clause of size k. We can construct an equivalently satisfiable set of clauses of size 3 as follows: First, add $k-3$ variables $x_1, \ldots, x_{k-3} \in V'$. The consider the following $k-2$ size-3 clauses

$$\begin{aligned}
C'_i = \{c'_{i,1} = \{ \quad & z_1, & z_2, & \overline{x_1}\}, \\
c'_{i,2} = \{ \quad & x_1, & z_3, & \overline{x_2}\}, \\
& & \ldots & \\
c'_{i,j+1} = \{ \quad & x_j, & z_{j+2}, & \overline{x_{j+1}}\}, \\
& & \ldots & \\
c'_{i,k-2} = \{ \quad & x_{k-3}, & z_{k-1}, & z_i\}\}
\end{aligned}$$

We claim that this is equivalent in satisfiability to C_i. That is, if C_i is satisfied, then C'_i is satisfied, and if C_i is not satisfied, then neither is C'_i. We'll show both.

Case 1: Suppose we we have an assignment that doesn't satisfy C_i. That means each literal $z_i = F$. We must show that no assignment to our new variables x_i can satisfy C'_i. Suppose for contradiction that there were some assignment to each x_i that made each clause in C'_i true. Let's see what that assignment is via induction:

Lemma 1.
Assume $z_i = F$ for all $z_i \in V$. For $1 \leq j \leq k-3$, $c'_{i,j} = T$ iff $x_j = F$.

Base Case: In the first clause, $z_1 = z_2 = F$, so $x_1 = F$ to make $\overline{x_1} = T$.

Inductive Step: Assume via our inductive hypothesis that if $c'_{i,j} = T$ then $x_j = F$ for $1 \leq j \leq k$. We will show to make $c'_{i,k+1} = T$ we need $x_{k+1} = T$.

Consider $c'_{i,k+1} = \{x_{k+1}, z_{k+2}, \overline{x_{k+2}}\}$. Since $z_{k+1} = F$ via our assumption that C_i isn't satisfied and $x_i = F$ via our IH, this clause is satisfied iff $x_{k+2} = F$, as desired.

Case 2: For this side, we assume that at least one $z_i = T$ (i.e., C_i is satisfied) and must show that there is a satisfying assignment w.r.t. x_is that satisfies C'_i. I'll provide a sketch of where things are different, construct the assignment, and leave it as an exercise for you to show that it does, in fact, constitute a satisfying assignment.

Let $z_{j'}$ be the smallest j' such that $z_{j'} = T$. Note that our inductive argument from the other case applies up until we reach $c_{i,j'-1}$. At that point, we can set $x_{j'-1} = T$ while maintaining the

truth of the clause. From there, we see that by setting $x_j = T$ for all $j \geq j'$, the rest of the clauses will be true via the first literal. Thus we have a satisfying assignment

$$x_j = \begin{cases} F, & j < j' - 1 \\ T, & j \geq j' - 1 \end{cases}$$

Thus, the set of size-3 clauses C'_i and the clause of size-k C_i are equivalent in terms of satisfiability — one is satisfiable iff the other is.

Our reduction goes as follows:

function SAT(V, C)
  $C' = \emptyset$
  for $C_i \in C$ **do**
    Let k be the number of literals in C_i
    Construct the $k-2$ clauses of C'_i as above.
    $V' \leftarrow V \cup \{x_i\}_{1 \leq i \leq k-3}$
    $C' = C' \cup C'_i$
  end for
  return 3SAT(V', C')
end function

To see that the conversion runs in polynomial time, consider the following: We construct $O(k)$ clauses and variables at each step — linear w.r.t. to the number of literals in C! We do this w.r.t. the number of clauses in C, so it's certainly polynomial time w.r.t. the length of C.

Correctness follows from the equivalence in satisfiability between each C_i and C'_i - one is satisfiable iff the other is. Thus, all C_i in C are satisfiable iff all the clauses in each C'_i, gathered in C' are. □

4.3 Summary

Now we have a fairly straightforward, if vague, theory of the hardness of algorithmic problems. If we accept *SAT* to be hard, then a reduction from *SAT* to any other problem ProbA proves that that ProbA is also hard. We also have a method of showing problems are easy/tractable: If we can find a polynomial-time algorithm to solve ProbA, then ProbA is easy. This can also be done implicitly via a reduction from ProbA to a known easy problem ProbB.

But what is the relationship between our polynomial-time formalization of easy and our "SAT reduces to this" definition of hard? Why do we think of SAT as hard anyway? The next section moves us into the more formal space of complexity theory to get some answers!

5 P, NP, and NP-Completeness

Just like we defined time complexity classes like $O(n)$, $O(n \log n)$ and so on, we'll define complexity classes for problems.

5.1 Class P

First, for easy problems:

Def (Complexity Class **P**)**.** A problem ProbA∈P iff there exists an algorithm that solves ProbA in worst-case $O(n^k)$ time for some $k \in \mathbb{Z}$.

This is what we've been working with so far, just now with a name — class P! We won't spend much time on this, since this is downstream from everything we've done so far in the course.

What's more interesting is how we formalize hardness! But first, a tangent: **Why not define hardness as $\overline{P}$?** The answer is simple — because that's not a super interesting way of defining hardness. Why? Because the number of interesting problems we can show are in $\overline{P}$ is quite limited.[1] So, we seek other candidate classes of hard problems...

5.2 Class NP and NP-HARD/NP-COMPLETE

Now another tangent: Consider the class of problems whose solutions you can *verify* TRUE in polynomial time! That is, if given an instance of a decision problem (i.e., the input to a decision problem), does there exist some object that can act as a *certificate* of output to the decision problem, such that verifying the output is TRUE with the certificate takes polynomial time. Here it is formally:

Def (Verifier)**.** A **Verifier** V of a problem ProbA is an algorithm that takes as input an instance of ProbA X and a *certificate* c such that $\exists c$ such that $\mathrm{V}(X, c) = \text{TRUE}$ iff the output of ProbA for X is TRUE. A verifier runs in polynomial time if it runs in polynomial time w.r.t. the size of X.

However, understanding what verifiability means is easier if we look at an example. Consider the following problem:

Problem Statement (Composite)**.**
Input: An $x \in \mathbb{Z}_+$
Output: TRUE iff x is a composite number.

Recall that a composite number is just a number that has a non-trivial factor (i.e., $\exists k \in \mathbb{N}, k \neq 1, x$ such that k divides x).

Suppose someone came up to you and claimed that some large integer x is composite. A naive approach to verifying this would be to loop through every integer between 2 and $x-1$ and see if they divide x — this takes $\Omega(x)$ time, which is exponential w.r.t. $n = \lceil \log_2 x \rceil$[2]. However, if they provided you with one of the non-trivial factors of x, we need only check whether the factor they provided is legitimate! Even a naive divisibility test like long division runs in polynomial time w.r.t n! Now, here's the last critical idea: Suppose they give you a false factor of x. Does this mean that x isn't composite? NO! Just that the person hasn't shown that x is composite. If any such factor of x exists, then x is composite, we just haven't been given it yet.

This is the idea underlying verification: We can verify the solution to the problem fast *if* we have access to a good certificate. Having a good certificate gives us certainty that the TRUE answer

[1] However, it would be misleading to suggest that $\overline{P}$ contains no useful problems — for example, determining whether there is a winning strategy in a generalized version of chess, checkers, or Go is known to take at least exponential time w.r.t. the size of the board.

[2] We conventionally think of the size of an integer input as the length of it's binary representation. If you haven't taken Computer Systems yet, wait until then to figure out why that length is $O(\log_2 n)$

to our decision problem is correct. However, being given a bad certificate only lets us conclude that the certificate is bad, not that the answer is FALSE.

Now, lets consider the verifier for the Composite problem:

function CompositeVerifier(x, c)
  Check c is an integer $2 \leq c \leq x - 1$
  if c divides x **then**
    return TRUE
  end if
  return FALSE
end function

And we can confirm that all of the steps in the pseudocode can be done in polynomial time w.r.t. $n = \lceil \log x \rceil$.

Another thing to pull from the Composite Problem is the idea that some things that are easy to verify are hard to solve in polynomial time![3] This motivates the idea that looking at the class of things that are verifiable in polynomial time is of interest! In fact, this is the class that we call NP:

Def (Complexity Class NP)**.** A problem ProbA ∈NP iff there exists a polynomial-time verifier for ProbA.

Now we finally get to the million-dollar question[4]: Is $P = NP$? That is, are all problems verifiable in polynomial time solvable in polynomial time? This is the P vs. NP problem!

This is hopefully a philosophically interesting question! It might seem intuitive that problems we can verify quickly permit quick solutions, but as of right now, we don't know! Additionally, many interesting and practically useful problems happen to be in NP, so the class NP is of practical interest!

Now, here is a quick check of your understanding of the definitions of P and NP. Is $P \subseteq NP$? This *should* be an immediate yes!

Statement 11. $P \subseteq NP$

The idea: Just ignore C and verify whether the answer is true by solving it yourself in polynomial time!

Proof. Let ProbA $\in P$. Thus there exists some algorithm A that solves ProbA in polynomial time. Then consider the following verifier for ProbA:

function AVerifier(X, c)
  return A(X)
end function

It runs in polynomial time w.r.t. X, since A runs in polynomial time w.r.t. X. If the answer to instance X is TRUE, then A(X) will return true and any c can act as a certificate. The same goes in reverse: No c will have AVerifier return true if the answer to instance X is FALSE. □

[3] In actuality, the Composite problem is solvable in polynomial time, but the algorithm relies on some pretty advanced number theory — The AKS primality test wasn't found until 2002! On the other hand, showing the Composite problem is verifiable in polynomial time requires no mathematical knowledge beyond the definition of a composite number.

[4] Literally — see the corresponding Millenium Prize problem

This leaves us with a bit of a problem: NP cannot be our class containing hard problems, because it contains all of our known easy problems as well! What we are actually aiming for are the *hardest* problems in class NP. Of course, we know how to determine relative difficulty! This gives us two new complexity classes:

Def (Complexity Class **NP-HARD**)**.** A problem ProbA$\in$ NP-HARD iff for every problem ProbB$\in$NP, ProbB reduces to ProbA in polynomial time.

Def (Complexity Class **NP-COMPLETE**)**.** A problem ProbA$\in$ NP-COMPLETE iff ProbA$\in$NP and is NP-HARD.

A bit less formally, NP-HARD consists of all problems as hard or harder than every problem in NP, and NP-COMPLETE contains all problems that are NP-HARD and also in NP.

We are going to use these two classes as our formalization of our idea of a hard problem. The reason I use both classes here is that, for our purposes, every problem we talk about is going to be NP-COMPLETE, and thus NP-HARD. Occasionally, the distinction will become relevant, but only because I'll ask you to demonstrate that a new problem is within class NP.

You might initially be worried because it might seem hard to show that *anything* is NP-HARD. How do you show that EVERY problem in NP reduces to a single problem? This is a fair concern, and we won't have the tools in this course to demonstrate how this is done (for that, take a theory of computation course!). It is worth noting that proving that there is an NP-COMPLETE problem is a major result in the field — the Cook-Levin Theorem. What problem did they show is NP-COMPLETE? None other than SAT!

Theorem 5.1 (Cook-Levin)**.**
SAT is NP-COMPLETE.

Now, once we know one NP-COMPLETE problem, showing other problems are NP-HARD is much more doable — simply reduce SAT to that problem!

Statement 12. If SAT reduces to some problem ProbA, then ProbA is NP-HARD.

Proof. By the Cook-Levin Thm, SAT is NP-COMPLETE. Thus, for any arbitrary problem ProbB$\in$NP, there exists a reduction from that problem to SAT. Let B2SAT represent the function that converts a problem instance X of ProbB to an instance (V, C) of SAT.

Then note that since we are given a reduction from SAT to ProbA. Let SAT2A represent the function that converts a problem instance (V, C) of SAT to an instance Y of ProbA.

Construct a reduction from ProbB to ProbA as follows, for some algorithm A that solves ProbA:

function B(X)
  return A(SAT2A(B2SAT(X)))
end function

It is a polynomial-time reduction because SAT2A and B2SAT are both polynomial time. It's correctness follows from the correctness of the two conversions. □

In fact, this idea of just chaining problem instance conversions can be used to show a much more general claim:

Statement 13. If ProbA reduces to ProbB, and ProbB reduces to ProbC, then ProbA reduces to ProbC. That is, reduction is *transitive*.

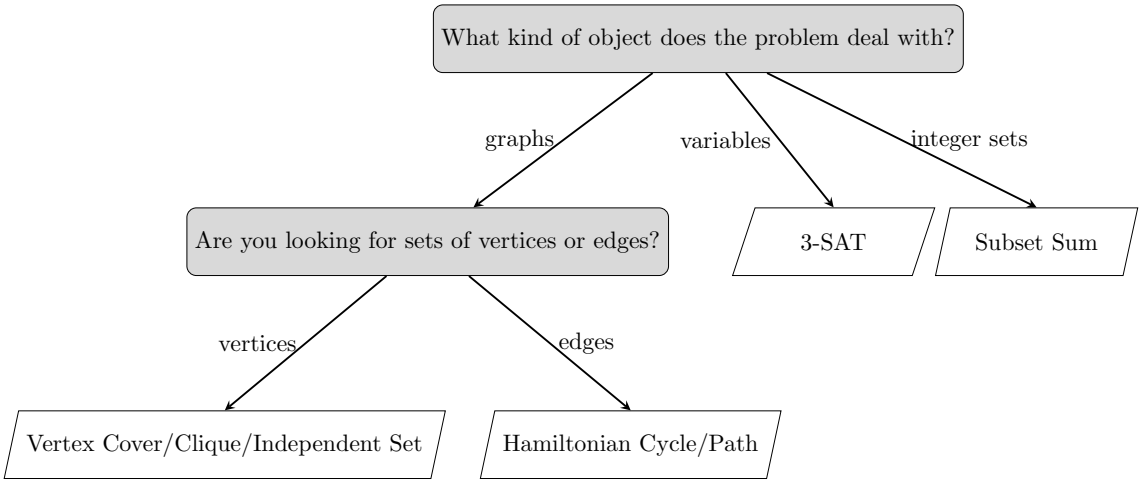

Figure 1: How to determine what problem to reduce from.

and give us our most useful tool:

Corollary 2. If ProbA reduces to ProbB and ProbA is NP-HARD, then ProbB is NP-HARD.

This is how we will practically show a problem is hard — we will reduce a known NP-HARD problem to the problem of interest, demonstrating that it is also NP-HARD! This is, of course, what we've already been doing: We've shown 3-SAT is NP-HARD via our prior reduction! We just didn't have the term for it yet!

5.3 Practically proving HARDness

Now, reducing from 3-SAT/SAT to a graph problem or a integer set problem is annoying. What is more useful in this class is to reduce from known NP-HARD problems that are similar in spirit to the problem of interest. Specifically, you want to reduce from very *simple* problems with a similar spirit. Are you dealing with a problem that is looking for a subset of the vertices in a graph? Consider reducing from Clique or Vertex Cover. How about if you're looking at a subset of edges? Consider Hamiltonian Path. If you are dealing with finding good assignments to variables, consider actually working with 3-SAT! this advice is summarized in Figure **??**.