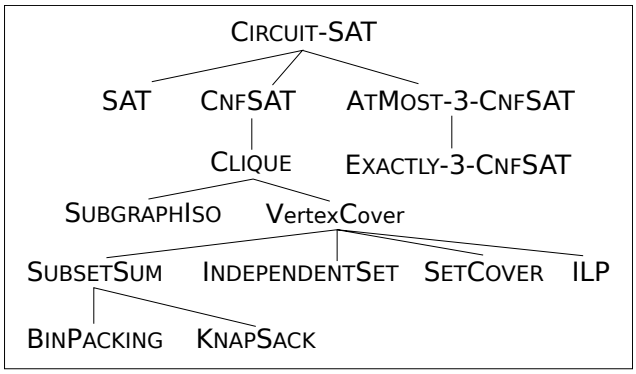
Lecture 11

- Some **NP**-complete problems.
- Reductions between problems.

In this lecture, we consider a number of problems and establish that they are **NP**-complete. As our first "anchor" problem that is **NP**-complete, we adopt Circuit-SAT. The following tree shows the reductions in this lecture that then establish each of those problems to be **NP**-hard.



Claim 66. Circuit-SAT *is **NP**-hard.*

A proof for the above claim is not part of this course. However, we outline one here as it is an important claim. Circuit-SAT is commonly employed as a kind of "anchor" **NP**-hard problem to then prove other problems to be **NP**-hard. A reduction *to* Circuit-SAT is useful as well, as we will discuss in the next lecture.

A proof relies on the following two observations. We credit the first to Dasgupta, et al., "Algorithms," and the second to CLRS.

- Every polynomial-time algorithm, i.e., algorithm that runs in time $O(n^k)$ for some constant k and input size n, can be encoded as a combinational acyclic boolean circuit of size $O(n^k)$. The informal argument behind this observation is that a commodity computer's state at any given moment can be expressed as a combinational boolean circuit. In the next CPU cycle, a combinational circuit takes as input the outputs of the previous cycle's circuit, and produces some output in 1 CPU cycle.

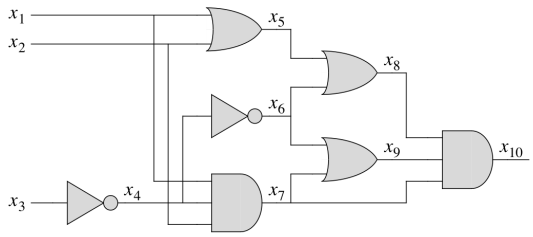
- Recall that if a problem is in **NP**, then there exists what we can call a *verification algorithm*, which takes as input a problem instance and an evidence/certificate/witness, and *accepts* that pair, i.e., outputs 1 if the certificate is indeed valid for that instance. The certificate is polynomially-sized in the instance, and the algorithm runs in time polynomial in its input, and therefore the instance.

  We encode the problem of identifying whether there exists a certificate as an instance of Circuit-SAT. That is, we build a circuit with input wires for a certificate. The logic of the circuit incorporates the problem instance and cross-checking the certificate against the instance. The single output wire of the circuit takes on the value 1 if the cross-check indeed deems that the certificate is valid for the instance. Thus, the Circuit-SAT instance corresponds to asking whether there exists a certificate (value for the input wires) for the problem instance (cause the output wire to be set to 1).

The following claim, which leverages transitivity of $\leq_k$ is useful to show that another problem is **NP**-hard given that a problem, e.g., Circuit-SAT is **NP**-hard.

Claim 67. SAT *is* ***NP****-hard.*

We prove the above claim by showing that Circuit-SAT $\leq_k$ SAT, and then appealing to Claims 66 and 63. The following example from CLRS illustrates a mapping from an instance of Circuit-SAT to an instance of SAT.



$$\begin{aligned}\phi \;=\; & x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \\ & \wedge\; (x_5 \leftrightarrow (x_1 \vee x_2)) \\ & \wedge\; (x_6 \leftrightarrow \neg x_4) \\ & \wedge\; (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ & \wedge\; (x_8 \leftrightarrow (x_5 \vee x_6)) \\ & \wedge\; (x_9 \leftrightarrow (x_6 \vee x_7)) \\ & \wedge\; (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))\,.\end{aligned}$$

We label each wire, including the input and output wires; in the above example, we have 10 wires that are labelled $x_1, \ldots, x_{10}$. Then, we introduce a *clause* in our instance of SAT for the output wire, and each wire that is the output of a gate. We perform a conjunction of the clauses to get our output formula. A clause is a boolean formula that is part of another boolean formula.

In the above example, for instance, we introduce a clause $x_6 \leftrightarrow \neg x_4$ because the wire x_6 is the output wire of a NOT gate, whose input wire is x_4. Similarly, we have a clause $x_9 \leftrightarrow (x_6 \vee x_7)$ because x_9 is the output wire of an OR gate whose inputs are x_6 and x_7.

The size of the output SAT formula for such a mapping is linear in the size of the input circuit, and can be computed in linear-time. We claim also that the output formula, ϕ, is satisfiable if and only if the input circuit is satisfiable. The reason is that the circuit is deemed satisfiable if and only if $x_{10} = 1$, which in turn is true if and only if the inputs, x_1, x_2, x_3 in this case, can be assigned boolean values that then cause the internal wires to acquire values that in turn cause x_{10} to take on the value 1. These values exactly cause ϕ to evaluate to 1, which again is the case if and only if $x_{10} = 1$, and all constraints internal to the circuit are satisfied, which map exactly to the clauses.

At this point, we point out that both Circuit-SAT and SAT are **NP**-complete, because apart from each being **NP**-hard, each is also in **NP**.

Claim 68. Circuit-SAT *is* ***NP****-complete.* SAT *is* ***NP****-complete.*

As we have already established that each of Circuit-SAT and SAT is **NP**-hard, all we need to do to prove the above claim is establish that each

is in **NP**. This is easy to do. For CIRCUIT-SAT, non-deterministically pick an input bit for each input wire. For SAT, non-deterministically pick an assignment to the boolean variables.

We now establish more problems in **NP** to be **NP**-hard. First, consider the following observation. For every combinational acyclic boolean circuit that allows AND and OR gates of up to n inputs each, where $n \in \mathbb{N}, n \geq 2$, there exists a combinational boolean circuit whose output is identical on every input, allows only two-input AND and OR gates, and has size $O(nc)$, where c is the size of the original boolean circuit.

The reason is that each n-input AND or OR gate can be expressed as the AND or OR of $n-1$ two-input AND or OR gates. That is, using prefix notation:

$$\text{AND}(x_1\, x_2\, \ldots\, x_n) = \text{AND}(\ldots(\text{AND}(\text{AND}(x_1\, x_2)\, x_3)\ldots)\, x_n)$$

And similarly for OR. Such a mapping can be perceived as a Karp-reduction from the version of CIRCUIT-SAT that allows n-input AND and OR gates to the version of CIRCUIT-SAT that allows two-input AND or OR gates only.

Claim 69. CIRCUIT-SAT*, in which circuits are restricted to two-input AND and OR gates only, is* ***NP****-hard.*

Now, assume that the reduction from CIRCUIT-SAT to SAT that underlies Claim 67 allows such restricted CIRCUIT-SAT instances only. Then, each clause has at most three variables. E.g., $x_5 \leftrightarrow (x_1 \vee x_2)$ may appear, but not $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$. Now, we can write $a \leftrightarrow b$ as $(a \rightarrow b) \wedge (a \leftarrow b)$, which in turn can be rewritten as $(\neg a \vee b) \wedge (a \vee \neg b)$.

As an example, applying this mindset to $x_5 \leftrightarrow (x_1 \vee x_2)$ gives us:

$$\begin{aligned} x_5 \leftrightarrow (x_1 \vee x_2) &\equiv (x_5 \rightarrow (x_1 \vee x_2)) \wedge (x_5 \leftarrow (x_1 \vee x_2)) \\ &\equiv (\neg x_5 \vee x_1 \vee x_2) \wedge (x_5 \vee \neg(x_1 \vee x_2)) \\ &\equiv (\neg x_5 \vee x_1 \vee x_2) \wedge (x_5 \vee (\neg x_1 \wedge \neg x_2)) \\ &\equiv (\neg x_5 \vee x_1 \vee x_2) \wedge (x_5 \vee \neg x_1) \wedge (x_5 \vee \neg x_2) \end{aligned}$$

Similarly, if we have a conjunction, following is an example.

$$\begin{aligned} x_5 \leftrightarrow (x_1 \wedge x_2) &\equiv (\neg x_5 \vee (x_1 \wedge x_2)) \wedge (x_5 \vee \neg(x_1 \wedge x_2)) \\ &\equiv (\neg x_5 \vee x_1) \wedge (\neg x_5 \vee x_2) \wedge (x_5 \vee \neg x_1 \vee \neg x_2) \end{aligned}$$

Thus, every instance of SAT produced by such a reduction has the following two characteristics.

- It is in *Conjunctive Normal Form* (CNF). A propositional boolean expression is said to be in CNF if it is a conjunction of clauses, where each clause is a disjunction of literals (or a single literal), where each literal is either a variable or its negation.

  Note that the output SAT instance being in CNF has nothing to do with the restriction of the CIRCUIT-SAT instances to two-input AND and OR gates. The output even in Claim 67 is, or can be made into, CNF using the same process we demonstrate above for the example $x_5 \leftrightarrow (x_1 \vee x_2)$.

- Each clause has at most three literals.

Leveraging only the first point above, let the problem CNFSAT be the following. Given input a boolean formula in propositional logic that is in CNF, is it satisfiable?

Claim 70. CNFSAT *is* ***NP****-complete.*

We also have the following claim for a restricted version of SAT, by leveraging both points above. Let the problem ATMOST-3-CNFSAT be the following problem. Given input a boolean formula in propositional logic that is in CNF with at most three literals per clause, is it satisfiable?

Claim 71. ATMOST-3-CNFSAT *is* ***NP****-complete.*

Another special case is what we can call EXACTLY-3-CNFSAT, in which every clause has exactly three literals.

Claim 72. EXACTLY-3-CNFSAT *is* ***NP****-complete.*

To prove that EXACTLY-3-CNFSAT is **NP**-hard, we can reduce from ATMOST-3-CNFSAT. In the input, we do nothing with clauses that have three literals. We need to do something with clauses that have one or two literals only. Consider a clause that has two literals only, i.e., is of the form $l_1 \vee l_2$.

For such a clause, we introduce a new variable, call it x. We map the clause $l_1 \vee l_2$ to: $(l_1 \vee l_2 \vee x) \wedge (l_1 \vee l_2 \vee \neg x)$. We now claim that an assignment causes $l_1 \vee l_2$ to evaluate to 1 if and only if that assignment causes $(l_1 \vee l_2 \vee x) \wedge (l_1 \vee l_2 \vee \neg x)$ to evaluate to 1.

We can prove this by case-analysis. Suppose some assignment causes $l_1 \vee l_2$ to evaluate to 1. Then at least one of l_1 or l_2 is 1 in that assignment. Without loss of generality, assume $l_1 = 1$. Then, $(l_1 \vee l_2 \vee x) \wedge (l_1 \vee l_2 \vee \neg x) = (1 \vee l_2 \vee x) \wedge (1 \vee l_2 \vee \neg x) = 1 \wedge 1 = 1$.

If in an assignment $l_1 \vee l_2 = 0$, we know $l_1 = l_2 = 0$ under that assignment. Thus, we have $(l_1 \vee l_2 \vee x) \wedge (l_1 \vee l_2 \vee \neg x) = x \wedge \neg x = 0$.

For a clause which has one literal only, call it l, we introduce two new variables x, y and map the clause l to $(l \vee x \vee y) \wedge (l \vee \neg x \vee y) \wedge (l \vee x \vee \neg y) \wedge (l \vee \neg x \vee \neg y)$.

Thus, in our output instance of EXACTLY-3-CNF-SAT, given as input an instance of ATMOST-3-CNF-SAT that has n clauses, we end up with at most $4n$, i.e., $\Theta(n)$, clauses.

We now consider a graph problem, CLIQUE. Given an undirected graph $G = \langle V, E \rangle$, a *clique* in G is a subset of the vertices that induce a complete subgraph of G. That is, $C \subseteq V$ is a clique if and only if for every distinct $u, v \in C$, $\langle u, v \rangle \in E$. The decision problem CLIQUE is: given as input an undirected graph G and an integer k, does G have a clique of size k? It is a decision version of an optimization problem that seeks to maximize the size of a clique in G.

Claim 73. CLIQUE *is* ***NP****-complete.*

CLIQUE is in **NP**: a certificate is a subset of the vertices which form a clique. It is easy to verify that, given a subset, it forms a clique. To show that CLIQUE is **NP**-hard, we reduce from CNFSAT. The following picture from CLRS illustrates the mapping that underlies the reduction. Note that CLRS reduces from EXACTLY-3-CNF-SAT (which they call 3-CNF-SAT) to CLIQUE. However, the mapping works from CNFSAT as well.

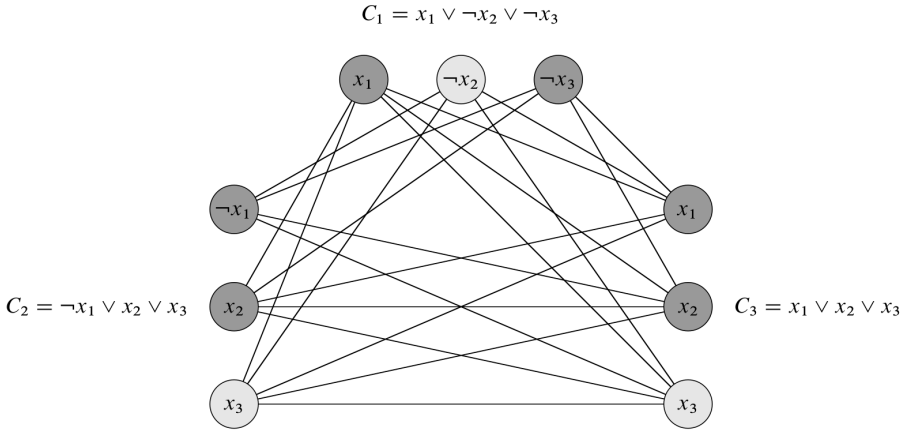


Figure 34.14 The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 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.

As the picture indicates, suppose the input instance of CNFSAT is $C_1 \wedge C_2 \wedge \ldots \wedge C_k$, where each C_i is a clause, which we represent as $l_1^{(i)} \vee l_2^{(i)} \vee \ldots \vee l_{n_i}^{(i)}$, where each $l_j^{(i)}$ is a literal. In our graph G that is output as part of our instance of CLIQUE, we introduce a unique vertex for every such $l_j^{(i)}$. Between two such vertices, call them $l_{j_1}^{(i_1)}$ and $l_{j_2}^{(i_2)}$, we draw an edge if and only if:

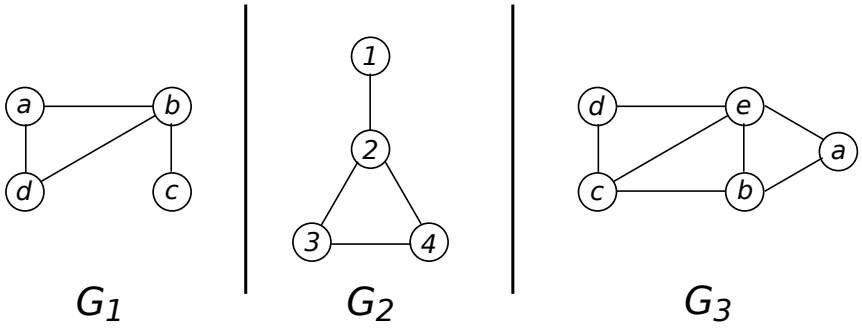
- $i_1 \neq i_2$; that is, $l_{j_1}^{(i_1)}$ and $l_{j_2}^{(i_2)}$ are literals in two different clauses, and,
- $l_{j_1}^{(i_1)}$ is not the negation of $l_{j_2}^{(i_2)}$. CLRS uses the term *consistent* for this. That is, two literals are consistent with one another if one is not the negation of the other.

The output instance of CLIQUE is $\langle G, k \rangle$, i.e., the graph as constructed above, and the integer input set to the number of clauses in the input instance of CNFSAT. We claim that the mapping can be constructed in polynomial-time, and that the input instance of CNFSAT is satisfiable if and only if the output instance of CLIQUE is satisfiable.

For the "polynomial-time" part, we observe that the output is of size, and can be computed in time, quadratic in the size of the input. The number of vertices in the graph is $3 \times$ the number of clauses, i.e., linear in the size of the input. The worst-case for the number of edges is an edge from every vertex that corresponds to a literal to a vertex that corresponds to every literal in every other clause, i.e., quadratic in the size of the input.

For the "if and only if" part, we first consider the following properties of a satisfying assignment to an instance of CnfSAT. In a satisfying assignment, every clause is satisfied, i.e., evaluates to 1. This can happen if and only if in every clause, at least one literal is 1. In the output graph G, we pick exactly those literals from each clause that are 1 in a satisfying assignment, and claim that those vertices form a clique. Such a set of vertices is of course of size k because we have k clauses. They must form a clique because there must be an edge between any two of them: they must consistent with one another, and they belong to different clauses.

We now consider another graph problem SubgraphIso. Given two graphs, $G_1 = \langle V_1, E_1 \rangle, G_2 = \langle V_2, E_2 \rangle$, an *isomorphism* between G_1 and G_2 is a bijection $f : V_1 \to V_2$ with the property $\langle u, v \rangle \in E_1 \iff \langle f(u), f(v) \rangle \in E_2$. In this case, we say that G_1 is isomorphic to G_2, and that the graphs are isomorphic to one another. For example, in the following picture, G_1 and G_2, which happen to be undirected, are isomorphic to one another. An isomorphic mapping $f : G_1 \to G_2$ is: $f(a) = 3, f(b) = 2, f(c) = 1, f(d) = 4$. Neither can be isomorphic to G_3 because no bijection can exist between their respective sets of vertices.



A graph G_1 is said to *subgraph isomorphic* to another graph G_2 if G_1 is isomorphic to a subgraph of G_2. Recall that a graph $H = \langle V_H, E_H \rangle$ is said to be a subgraph of $G = \langle V_G, E_G \rangle$ if and only if $V_H \subseteq V_G, E_H \subseteq E_G$ and

$\langle u, v\rangle \in E_H \implies u, v \in V_H$. In the figure above, each of G_1 and G_2 is subgraph isomorphic to G_3. The problem SUBGRAPHISO is: given as input two graphs, $\langle G_1, G_2\rangle$, is G_1 subgraph isomorphic to G_2?

Claim 74. SUBGRAPHISO *is* ***NP****-complete.*

To prove that SUBGRAPHISO is in **NP**, given a true instance $\langle G_1, G_2\rangle$, we could adopt as certificate $\langle H, f\rangle$, where H is a subgraph of G_2, and $f : V[G_1] \to H$ is a bijection between the vertex-set of G_1 and the vertex-set of that subgraph of G_2. Our verification algorithm would first check that H is indeed a subgraph of G_2, and then that f is indeed an isomorphism between G_1 and H. The certificate is linear in the size of the instance, and the verification algorithm is at worst quadratic.

To prove that SUBGRAPHISO is **NP**-hard, we reduce from CLIQUE. Given an instance $\langle G, k\rangle$ of CLIQUE, first construct a complete graph, call it H, of k vertices. Then output $\langle H, G\rangle$ as our instance of SUBGRAPHISO. If the input $\langle G, k\rangle$ is a true instance of CLIQUE, then $\langle H, G\rangle$ is a true instance of SUBGRAPHISO because there must be a subgraph of G that is isomorphic to H. If $\langle G, k\rangle$ is a false instance of CLIQUE, then no complete subgraph of k vertices can be subgraph isomorphic to G.

We now consider yet another graph problem, VERTEXCOVER. Recall that the decision problem VERTEXCOVER is: given as input an undirected graph G and an integer k, does G have a vertex cover of size k?

We can show that VERTEXCOVER is **NP**-hard via a reduction from CLIQUE and exploiting the following property for the *complement* of an undirected graph G, represented as $\overline{G}$. Given an undirected graph $G = \langle V, E\rangle$, its complement is $\overline{G} = \langle V, \overline{E}\rangle$, where $\langle u, v\rangle \in E \iff \langle u, v\rangle \notin \overline{E}$. That is, the complement graph has all the same vertices, and an edge between a pair of vertices if and only if the original does not have that edge.

We now make the following claim that readily gives us a reduction between CLIQUE and VERTEXCOVER.

Claim 75. *C is a clique of $G = \langle V, E\rangle$ if and only if $V \setminus C$ is a vertex cover of $\overline{G} = \langle V, \overline{E}\rangle$.*

Proof. For the "only if" direction, suppose $u, v \in V$, $u \neq v$. Then:

$$\begin{aligned}(u \in C \wedge v \in C) &\iff (u \notin V \setminus C \wedge v \notin V \setminus C) \\ &\implies \langle u, v\rangle \in E \\ &\iff \langle u, v\rangle \notin \overline{E}\end{aligned}$$

Taking the contrapositive, we have:

$$\langle u, v\rangle \in \overline{E} \implies ((u \in V \setminus C) \vee (v \in V \setminus C))$$

which means $V \setminus C$ is a set cover of $\overline{G}$.

For the "if" direction, suppose C is not a clique of G. Then there exist distinct $u, v \in V$ such that $\langle u, v\rangle \notin E$ and $u, v \in C$. Now for that particular u, v, we have: $\langle u, v\rangle \in \overline{E}$ and yet $u \notin V \setminus C$ and $v \notin V \setminus C$. Thus, $V \setminus C$ is not a vertex cover of $\overline{G}$. □

Claim 76. VertexCover *is* ***NP****-complete.*

For the **NP**-hard part, we reduce from Clique. Given an instance $\langle G = \langle V, E\rangle, k\rangle$ of Clique we output as our instance of VertexCover, $\langle \overline{G}, |V| - k\rangle$. This reduction function is in fact invertible. That is, given an instance $\langle G = \langle V, E\rangle, k\rangle$ of VertexCover, we can output $\langle \overline{G}, |V| - k\rangle$ as an instance of Clique.

Clique and VertexCover are also intimately tied to another graph property and problem, IndependentSet. Given an undirected graph $G = \langle V, E\rangle$, an *independent set*, $I \subseteq V$, has the property that for any two distinct $u, v \in I$, $\langle u, v\rangle \notin E$. Thus, an independent set is a subset of the vertices, no two of whom have an edge between them.

Claim 77. *$C \subseteq V$ is a vertex cover of $G = \langle V, E\rangle$ if and only if $V \setminus C$ is an independent set of G.*

Proof. For the "only if" direction, suppose $C \subseteq V$ is a vertex cover of $G = \langle V, E\rangle$. Then, $\langle u, v\rangle \in E \implies ((u \in C) \vee (v \in C))$. The contrapositive is: $((u \in V \setminus C) \wedge (v \in V \setminus C)) \implies \langle u, v\rangle \notin E$.

For the "if," suppose $C \subseteq V$ is not a vertex cover of G. That is, there is some pair of distinct $u, v \in V$ such that $\langle u, v\rangle \in E \wedge u \notin C \wedge v \notin C$. That is exactly $\langle u, v\rangle \in E \wedge u \in V \setminus C \wedge v \in V \setminus C$. □

Thus, to show VertexCover $\leq_k$ IndependentSet, given as input an instance of VertexCover $\langle G = \langle V, E\rangle, k\rangle$, we output as an instance of IndependentSet, $\langle G, |V| - k\rangle$.

We now transition from graph problems to set problems. We start with a straightforward reduction from VertexCover to SetCover. SetCover is the following problem: given as input a set G, a set $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ where each $S_i \subseteq G$, and a natural number k, does there exist a subset of $\mathcal{S}$ of size k whose union is G?

For example, if $G = \{1, 2, 3, 4\}, \mathcal{S} = \{\{2\}, \{2, 3, 4\}, \{1, 2\}, \{2, 3\}\}$ and $k = 3$, then the answer is '`true`.' However, if for those same $G, \mathcal{S}$, we set $k = 1$, the answer is '`false`.'

A reduction from VertexCover is straightforward. Given an instance $\langle\langle V, E\rangle, k\rangle$ of VertexCover, we map it to an instance $\langle G, \mathcal{S}, k\rangle$ of SetCover, where $G = E$, and we have a set $S_u \in \mathcal{S}$ corresponding to each $u \in V$ where S_u contains those edges on which u is incident. Consequently, we are able to state the following claim.

Claim 78. SetCover *is* ***NP****-complete.*

We now show that one more problem related to sets is **NP**-complete. Let SubsetSum be the following decision problem: given as input a set S of natural numbers and another natural number k, does there exist a subset of S the sum of whose members is k? This is a slightly different version of SubsetSum than we discussed before. In this version, (i) every item is a natural number, i.e., a positive integer, and, (ii) there is a second input, which is another natural number. The earlier version allowed S to be a set of integers (positive, negative and 0), and asked whether there is a subset that adds up to 0.

We make the following claim about this version of the problem.

Claim 79. SubsetSum *is* ***NP****-complete.*

To prove that SubsetSum is **NP**-hard, we reduce from VertexCover. Suppose we are given an instance of VertexCover, $\langle G = \langle V, E\rangle, k\rangle$. We need to devise a function that produces an instance of SubsetSum, $\langle S, k'\rangle$. We assume that $k \in [0, |V| - 1]$.

We assume our arithmetic is performed base-10. Every member of S, and the number k', comprises $\lfloor \log_{10}(|V|-1) \rfloor + 1 + |V| + |E|$ digits, perhaps with leading 0's. The digits are used as follows.

- The leading $\lfloor \log_{10}(|V|-1) \rfloor + 1$ digits are either 0 or 1 each, or used to encode the value k in base-10.
- The next $|V|$ digits are each either 0 or 1 only, as we discuss below.
- The last $|E|$ digits are each either 0, 1 or 2 only, as we discuss below.

Let $V = \{u_1, u_2, \ldots, u_{|V|}\}$. For each u_i, we introduce two numbers into S, call these n_{i_i}, n_{i_2}. The number n_{i_1} has the following for its digits:

1. For the first $\lfloor \log_{10}(|V|-1) \rfloor + 1$ digits, n_{i_1} has the value 1, i.e., $00\ldots01$.
2. We designate one of the next $|V|$ digits, say the i^{th} digit for the vertex u_i. In n_{i_1}, we set that digit to 1, and set all other digits in these $|V|$ digits to 0.
3. We designate one of the last $|E|$ digits for each edge. For each of those digits that corresponds to an edge which is incident on u_i, we set that digit to 1 in n_{i_1}.

The number n_{i_2} has 0 in every digit except the digit in the sequence of $|V|$ digits that corresponds to vertex u_i; that digit in n_{i_2} is 1. So n_{i_2} looks like $00\ldots010\ldots0$.

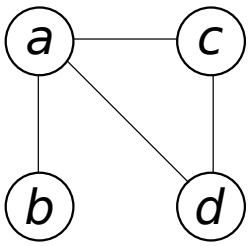
The mindset behind the above is the following. We can either choose vertex u_i to be in our vertex cover, or not. If we choose it to be in our vertex cover, this corresponds to choosing n_{i_1} to contributing to our sum, which comes with a "cost" and a "payoff." The leading value of 1 for the leading $\lfloor \log_{10}(|V|-1) \rfloor + 1$ is the "cost" – the vertex u_i eats up one of the spots amongst the size k we are allowed for a vertex cover. The payoff is the digits that are set for the edges that are incident on it. The number n_{i_2} corresponds to the situation that we do not choose u_i to be in our vertex cover. In this case, we have neither a cost nor a payoff; only the digit that is set to indicate this vertex's identity.

Corresponding to each edge $\langle u_i, u_j \rangle \in E$, we introduce one number in to S, call it $m_{i,j}$. This number, $m_{i,j}$ has 0's in the leading $\lfloor \log_{10}(|V| - 1) \rfloor + 1$ digits, and 0's in the next $|V|$ digits. For the last $|E|$ digits, it has all 0's except for the digit that corresponds to this edge, which is set to 1. So, $m_{i,j}$ looks like $0 \ldots 010 \ldots 0$, where the "1" appears in that digit that is designated for this edge.

Finally, k' is the following value. The first $\lfloor \log_{10}(|V| - 1) \rfloor + 1$ digits of k' is the base-10 encoding of the value k from the input VERTEXCOVER instance. E.g., if $\lfloor \log_{10}(|V| - 1) \rfloor + 1 = 5$ and $k = 103$, then the leading five digits of k' are 00103. The next $|V|$ digits are all 1, and the last $|E|$ digits are all 2. So k' looks like: $k\,11 \ldots 12 \ldots 2$.

That's it. We output this $\langle S, k' \rangle$ as our instance of SUBSETSUM and claim that it is computable in polynomial-time and has the "if and only if" property. We show an example below.

Suppose in our input instance of VERTEXCOVER, we have the following graph, G.



So we have $|V| = |E| = 4$, and $\lfloor \log_{10}(|V| - 1) \rfloor + 1 = 1$. So all of our numbers are 9 digits. Below are the numbers in S, and k'. The table shows

the manner in which they are chosen as per the above scheme.

		$\lfloor \log_{10}(\lvert V \rvert - 1) \rfloor + 1$	a	b	c	d	$\langle a, b \rangle$	$\langle a, c \rangle$	$\langle a, d \rangle$	$\langle c, d \rangle$
a	n_{a_1}	1	1	0	0	0	1	1	1	0
	n_{a_2}	0	1	0	0	0	0	0	0	0
b	n_{b_1}	1	0	1	0	0	1	0	0	0
	n_{b_2}	0	0	1	0	0	0	0	0	0
c	n_{c_1}	1	0	0	1	0	0	1	0	1
	n_{c_2}	0	0	0	1	0	0	0	0	0
d	n_{d_1}	1	0	0	0	1	0	0	1	1
	n_{d_2}	0	0	0	0	1	0	0	0	0
$\langle a, b \rangle$		0	0	0	0	0	1	0	0	0
$\langle a, c \rangle$		0	0	0	0	0	0	1	0	0
$\langle a, d \rangle$		0	0	0	0	0	0	0	1	0
$\langle c, d \rangle$		0	0	0	0	0	0	0	0	1
k'		k	1	1	1	1	2	2	2	2

Thus, $|S| = 12$. Now, if we set $k = 1$, we need to choose exactly one from amongst $n_{a_1}, n_{b_1}, n_{c_1}, n_{d_1}$. Suppose we choose a for this. Then, we have chosen, to be in our sum, $n_{a_1}, n_{b_2}, n_{c_2}, n_{d_2}$, for a sum of 111111110. We can also choose any and all from amongst the edges, but the maximum sum we can reach is 111112221; i.e., we cannot achieve k'.

However, if we set, for example, $k = 3$, then $k' = 311112222$. And now if we choose b, c, d for the subscript 1, and a for the subscript 2, we have $n_{b_1} + n_{c_1} + n_{d_1} + n_{a_2} = 311111112$. We now add to that sum the numbers that correspond to the edges $\langle a, b \rangle, \langle a, c \rangle, \langle a, d \rangle$, and we get k'.

Integer Linear Programming (ILP) is another well-known problem. It is a discrete version of a problem whose continuous version is tractable. We now establish that ILP is **NP**-complete. The version of ILP we address is: given an $m \times n$ matrix $\mathbf{A} \in \mathbb{Z}^{m \times n}$, and an $m \times 1$ matrix $\mathbf{b} \in \mathbb{Z}^m$, does there exist an $n \times 1$ matrix $\mathbf{x} \in \{0, 1\}^n$ such that $\mathbf{Ax} \leq \mathbf{b}$?

We observe that the above corresponds to m equations, each of the form:

$$a_{i,1}x_1 + a_{i,2}x_2 + \ldots + a_{i,n}x_n \leq b_i$$

where each $a_{i,j}, b_i \in \mathbb{Z}$, and we require each x_j is either 0 or 1.

Claim 80. ILP *is* ***NP****-complete.*

We first observe that ILP is in **NP** — a certificate is a solution, $\mathbf{x}$, which is of size $\Theta(n)$. To prove that the problem is **NP**-hard, we reduce from VertexCover. Given an instance of VertexCover, $\langle G, k\rangle$ where $G = \langle V, E\rangle$ and $k \in \mathbb{N}$, assume that $V = \{1, \ldots, n\}$. We introduce the variables $x_1, \ldots, x_n$; the intent is that $x_j = 1$ if and only if vertex j is in a vertex cover.

We introduce the following constraints:

$$\begin{aligned} x_a + x_b &\geq 1 && \forall \langle a, b\rangle \in E \\ x_1 + x_2 + \ldots + x_n &\leq k \end{aligned}$$

Note that a constraint of the form $S \geq T$ is equivalent to $-S \leq -T$. We now claim, via a straightforward proof, that the input instance of VertexCover is true if and only if the output instance of ILP is true. If we perceive the output ILP instance as the two matrices $\mathbf{A}$ and $\mathbf{b}$, $\mathbf{A}$ is $(|E| + 1) \times |V|$, and b is $(|E| + 1) \times 1$, which is at worst polynomial, specifically quadratic, in the size of the input VertexCover instance $\langle G, k\rangle$.

We finally consider another family of problems, which we can call "containment" or "fitting." We start with BinPacking: given as input (i) n bins each of capacity some $c \in \mathbb{N}$, (ii) m items, with volumes $v_1, v_2, \ldots, v_m$, and (iii) $k \in \mathbb{N}$, does there exist a set of k bins that fit all m items? If we put an item in a bin, then the bin's capacity is subtracted by that item's volume.

Claim 81. BinPacking *is* ***NP****-complete.*

The $\in$ **NP** part is straightforward. For the $\in$ **NP**-hard part, we reduce from SubsetSum. Given an instance of SubsetSum, $\langle S, k\rangle$, let $S = \{a_1, a_2, \ldots, a_n\}$, and $A = \sum_{a \in S} a$; that is, A is the sum of all the items in S.

In our instance of BinPacking, we set $c = 2A$, $k = 2$ and $m = n + 2$, where $v_1 = a_1, \ldots, v_n = a_n$, $v_{n+1} = 2A - k$, and $v_{n+2} = A + k$.

If the input instance of SubsetSum is true, then, the items $a_1, \ldots, a_n$ can be partitioned such that one partition, call it p_1, has sum k and the other has sum $A - k$. Thus, we can fit the items that correspond to the ones in p_1

and the item with volume $v_{n+1} = 2A - k$ into one bin, for a total volume of $2A-k+k = 2A$. The other items, which have total volume $A-k+A+k = 2A$ fit in the other bin. Thus, the instance of BinPacking is true.

Suppose the input instance of SubsetSum is false, yet, for the purpose of contradiction, the corresponding instance of BinPacking is true. As the sum of volumes of the items is $4A$, both bins have to be full. Now consider the item with volume $v_{n+1} = 2A - k$. The remainder of the items in that bin must add up in volume to k, which contradicts the assumption that the input instance of SubsetSum is false.

We now consider KnapSack: given n items, where each item is a pair $\langle v_i, w_i \rangle$, where v_i is the value of, or payoff from, the item, and w_i is the volume or weight it consumes, and a "knapsack," which is a pair $\langle V, W \rangle$, does there exist a subset of the n items whose value adds up to at least V, and whose weights add up to at most W?

Claim 82. KnapSack *is* ***NP****-complete.*

Showing that KnapSack is in **NP** is easy: a certificate is a subset of items whose values add up to at least V and weights add up to at most W.

To show that KnapSack is **NP**-hard, we reduce from SubsetSum. Given an instance of SubsetSum, $\langle S, k \rangle$, let $S = \{a_1, \ldots, a_n\}$. In our KnapSack instance, we introduce n items, where for the i^{th} item, $v_i = w_i = a_i$, i.e., its volume and weight are both a_i. We set $V = W = k$. We observe that the sum of values of some items adds up to at least V if and only if their sum of values adds up to at most V, i.e., exactly V, because $V = W$, and the value and weight of each item is the same.

Weak NP-hardness We now consider an interesting property of KnapSack. Consider the following optimization version, KnapSackOpt: given n items, $\langle v_1, w_1 \rangle, \ldots, \langle v_n, w_n \rangle$, and a capacity W for the knap sack, what is the maximum total value we can achieve without exceeding weight W? A polynomial-time algorithm exists for the decision version if and only if one exists for this optimization version.

It turns out that this problem possesses a kind of optimal substructure, which can be exploited to yield an algorithm based on dynamic programming.

Suppose we impose some ordering of the items, $1, \ldots, n$, and proceed starting with the first. Suppose the current capacity of the knap sack is C, and we are trying to determine whether we should take the i^{th} item or not.

If we take the i^{th} item, we have a payoff, but have to incur a cost. The payoff is that v_i is added to our total value. The cost is that our capacity decreases to $C - w_i$. Or we can choose not to take it, and we get no payoff, but incur no cost either. We make the choice from amongst those two options based on whichever one maximizes the total value we achieve.

That is, suppose $V[i, C]$ is the maximum total value we can achieve from only the items $1, \ldots, i$, given capacity C. Then, the final solution we seek is $V[n, W]$. And a recurrence is:

$$V[i, C] = \begin{cases} -\infty & \text{if } C < 0 \\ 0 & \text{if } i = 0 \text{ or } C = 0 \\ \max\{v_i + V[i-1, C-w_i], V[i-1, C]\} & \text{otherwise} \end{cases}$$

We can realize the above recurrence as an algorithm based on dynamic programming which fills in the two-dimensional array $V[0 \ldots n, 0 \ldots W]$. The algorithm runs in time $O(nW)$, and in $\Theta(nW)$ in the worst-case.

Is this a polynomial-time algorithm? The answer is: 'depends.' On the base we adopt for our encoding. If our encoding is unary (base-1), then this is a polynomial-time algorithm. But if it is binary (base-2), or any base-x, where $x > 1$, then this is not a polynomial-time algorithm because on input, W can be encoded with $O(\lg W)$ bits only, and W is exponential in $\lg W$.

Such an algorithm, for which there exists an encoding for which it runs in polynomial-time, but not for all encodings, is called a *pseudo-polynomial time* algorithm. The underlying problem, KnapSack in this case, for which a pseudo-polynomial time algorithm exists, but is **NP**-hard, is said to be *weakly* **NP**-hard. Examples of problems that are *strongly* **NP**-hard are VertexCover and SAT.