Problems marked with **E** are graded on effort, which means that they are graded subjectively on the perceived effort you put into them, rather than on correctness. For bonus questions, we will not provide any insight during office hours or Piazza, and we do not guarantee anything about the difficulty of these questions. We strongly encourage you to typeset your solutions in LaTeX.

If you collaborated with someone, you must state their names. You must write your own solution and may not look at any other student's write-up.

0. If applicable, state the name(s) and uniqname(s) of your collaborator(s).

Solution:

1. In this problem we will consider what conclusions we can draw when learning new information about languages.

   (a) Consider the language TSP, as covered in discussion:

   $$\text{TSP} = \{(\langle G \rangle, k) : G \text{ is an undirected, weighted, complete graph with a tour of weight } \leq k\}$$

   Recall that TSP is NP-Complete.
   For each of the situations described in i., ii., and iii., answer the following questions:

   (1) What, if anything, does this information imply about whether or not TSP $\in$ P?
   (2) What, if anything, does this information imply about whether or not TSP $\in$ NP?
   (3) Using this information, could you conclude anything new about the relationship between the classes P and NP?

   i. You find an algorithm that decides TSP in $O(2^n)$ time, where n is the bit length of the input.

   Solution: This wouldn't allow us to conclude much because it would not tell us if we can solve TSP in polynomial time with respect to the input size. This means we still do not know if TSP $\in$ P. We have already learned that TSP is efficiently verifiable; this information does not change the fact that TSP $\in$ NP. Since we did not learn if TSP $\in P$, this does not allow us to determine whether or not P = NP.

   ii. You find an algorithm that decides TSP in $O(n^5)$ time, where n is the bit length of the input.

   Solution: This implies that TSP $\in P$ because we have found an efficient decider. This does not change that TSP $\in$ NP. Our conclusion that TSP $\in$ P is very important because this also implies P = NP. We know that TSP is NP-Complete, and we have proven that if L is NP-Complete, then $L \in P \implies$ P = NP. So, if TSP $\in$ P, then P = NP.

iii. You prove that no algorithm could decide TSP in polynomial time.

> **Solution:** This would mean that TSP $\notin$ P. We still know that TSP $\in$ NP (this finding does not change that). This means we found a language L such $L \notin$ P but $L \in$ NP, so P $\neq$ NP.

(b) Consider a language L such that $L \in$ NP, but where L is not known to be NP-Complete. For each of the situations described in i. and ii., answer the following questions:

(1) What, if anything, does this information imply about whether or not $L \in$ P?

(2) Using this information, could you conclude anything new about the relationship between the classes P and NP?

i. You find an algorithm that decides this language in $O(n^2)$ time, where n is the bit length of the input.

> **Solution:** This means $L \in$ P. If there is some algorithm that decides L in polynomial time, then by definition $L \in$ P. However, this does not allow us to draw a conclusion about whether or not P = NP. We already know that P is a subset of NP, and finding one more language L such that $L \in$ P and $L \in$ NP does not imply that all languages in NP are also in P - unless L is NP-Complete, but we do not know if it is.

ii. You prove that no algorithm can decide this language in polynomial time.

> **Solution:** By definition, this means that $L \notin$ P. This means that $L \in$ NP but $L \notin$ P, so it would allow us to conclude that P $\neq$ NP.

2. Recall the language VERTEX-COVER as defined in lecture:

$$\text{VERTEX-COVER} = \{\langle G, k\rangle : G \text{ is an undirected graph that has a vertex cover of size } k\}$$

We proved in Lecture 14 that VERTEX-COVER is NP-Hard by showing that 3-SAT $\leq_p$ VERTEX-COVER, and we stated that VERTEX-COVER is NP-Complete.

In this problem, we will perform the reduction from lecture on two examples. Then, we will complete the proof that VERTEX-COVER is NP-Complete by showing that VERTEX-COVER $\in$ NP.

(a) In our reduction, 3-SAT $\leq_p$ VERTEX-COVER, we defined a function f that takes in a Boolean formula ϕ and outputs a graph G and a number k. We showed that f has the following properties:

- $\phi \in$ 3-SAT $\iff$ $f(\phi) = (G, k) \in$ VERTEX-COVER
- $f(\phi)$ can be computed in polynomial time.

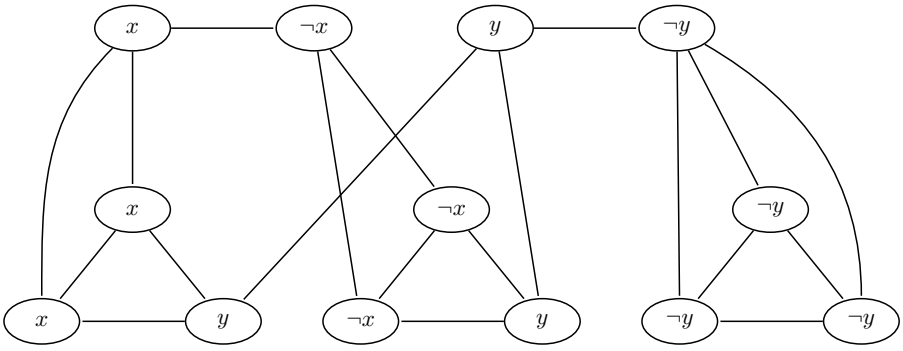
For each of the following formulas ϕ, use the reduction described in lecture to compute $f(\phi)$, which is the corresponding graph G and number k. In particular, you should compute the correct value of k.

- If $\phi \in$ 3-SAT, provide a satisfying assignment α, and show that $f(\phi) = (G, k) \in$ VERTEX-COVER by giving a vertex cover of G of size k (in particular, you should compute the correct value of k) that corresponds to α - similar to the example in the lecture.
- If $\phi \notin$ 3-SAT, state that there is no satisfying assignment of ϕ, then show that $f(\phi) = (G, k) \notin$ VERTEX-COVER by showing that there is no vertex cover of G with size k (in particular, you should compute the correct value of k).

i. $(x \vee y \vee x) \wedge (\neg x \vee y \vee \neg x) \wedge (\neg y \vee \neg y \vee \neg y)$

Solution:
There is no satisfying assignment of ϕ. To see that this is true, suppose that there was a satisfying assignment of ϕ. Since the third clause, $(\neg y \vee \neg y \vee \neg y)$, must be true, $y = False$. Since $y = False$, the second clause implies that $x = False$. But this is a contradiction, since now the first clause, $(x \vee y \vee x)$, evaluates to False.

Now we will show that $f(\phi) \notin$ VERTEX-COVER. First we need to find the correct value for $f(\phi) = (G, k)$ produced by the reduction. The graph produced is shown below.



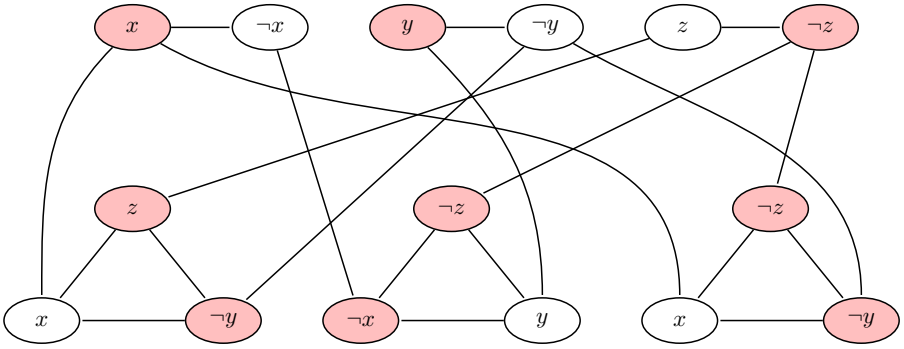
Using our reduction from lecture, we set $k = n + 2m$, where n is the number of variables and m is the number of clauses. Since $n = 2$ and $m = 3$, $k = 8$. So, we will show that the graph produced does not have a vertex cover of size 8.

Suppose that the graph has a vertex cover of size 8. Let's call this vertex cover C. This means that for every edge (u, v) in the graph, either u or v or both is in C. We will begin by considering each of the edges that exist in the variable gadgets and the clause gadgets. Each variable gadget is a complete graph of size 2, and from this we can see that we need to choose one vertex for each variable gadget. Each clause gadget is a complete graph of size 3, which is an indication that we need to choose 2 vertices within each clause gadget. Since we also know that $k = n + 2m = 8$, this means that we can choose at most one vertex for each variable gadget and at most two vertices for each clause gadget. Thus, C will contain exactly one vertex for each variable gadget and exactly two vertices for each clause gadget.
Let's first consider the $\neg y$ nodes in the rightmost clause. We will use the term "crossing edge" to refer to an edge that crosses between a variable and a clause gadget. Regardless of which two vertices we choose in the clause, there will be

one crossing edge that is not covered. This means that we need to choose the $\neg y$ node from the variable gadget.
Next, we will consider the nodes in the middle clause. We must select the y node to cover its crossing edge. This means we will only pick one of the two $\neg x$ nodes, and whichever one we do not pick will have a crossing edge that is not covered. Thus, we must select the $\neg x$ variable node to cover that edge.
However, here we face a contradiction because then no matter which two nodes we choose from the first clause, there will still be an uncovered crossing edge in the first clause. Since there is no way to choose 8 vertices such that the vertices cover all edges, it cannot be the case that this graph has a vertex cover of size 8.

ii. $(x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee \neg z)$

Solution: The graph produced by the reduction is as follows:



The value k is $k = n + 2m = 9$.
There are several satisfying assignments: $(1, 1, 1)$, $(1, 1, 0)$, $(1, 0, 0)$, $(0, 0, 1)$, and $(0, 0, 0)$. In the graph above, we depict a vertex cover corresponding to the assignment $(1, 1, 0)$. Such a cover includes the vertices x, y, and $\neg z$ out of the variable gadgets, corresponding to the truth value of each variable, and these vertices cover the edges internal to the variable gadgets. Since $(1, 1, 0)$ is a satisfying assignment, each clause has at least one of its crossing edges, connected to a true literal in the clause, covered by a vertex from a variable gadget. We need to pick two vertices from each clause gadget to cover the internal edges, and we choose them to cover any remaining crossing edges. The shaded vertices in the graph above represent one such cover (there are others for this assignment – the second and third clauses each have two crossing edges covered by a vertex from a variable gadget, which gives us two viable choices of which vertices to pick for each clause). We can see that the vertices do cover all edges, and the cover has size 9 as required.

(b) Show that VERTEX-COVER $\in$ NP and conclude that VERTEX-COVER $\in$ NP-Complete.

Solution: We can prove that VERTEX-COVER $\in$ NP by constructing an efficient verifier for VERTEX-COVER.

For a certificate, our verifier can take in a set of vertices so that it can check if this set is a vertex cover of the graph.

This means the input to our verifier will be $(G = (V, E), k, C)$, where C is a set of vertices.

We can construct our verifier in the following way:

1. If C is not a subset of V, reject.
2. If $|C| \neq k$, reject.
3. Loop through all edges (u, v) in E.
    a. If $u \notin C$ and $v \notin C$: Reject
4. Accept

Now we need to prove that this is an efficient verifier by showing that it is correct and it runs in polynomial time with respect to the input size.

$(G = (V, E), k) \in$ VERTEX-COVER $\implies$ G has a vertex cover of size k. Let C be the vertex cover. This means C is a subset of V and has size k. For every edge $(u, v) \in E$, either u or v (or both) is in C. Thus, given C, the verifier accepts.

$(G = (V, E), k) \notin$ VERTEX-COVER $\implies$ G does not have a vertex cover of size k. Let C be any set of vertices. Since G does not have a vertex cover of size k, it must be the case that either C does not have size k or does not contain a vertex for every edge in E. This means that when our verifier is given any certificate, either step 2 or step 3a will fail. Thus, our verifier rejects for all certificates.

We can prove that our verifier runs in polynomial time with respect to the input size by evaluating each step. Step 1 checks that each vertex in C is also a vertex in V, which we can do in $O(|V^2|)$ operations (if we find that C has more elements than V, we can immediately reject). Since a valid k must be $\leq |V|$, we can perform step 2 in $O(|V|)$ operations. For step 3, looping through all edges is $O(|E|)$, and in each iteration of the loop, we need to check at most $|V|$ vertices in C for the existence of u or v, meaning we can complete this loop in $O(|E| \cdot |V|)$ operations. We have shown that our verifier runs in polynomial time with respect to the input size.

3. Show that L_{HALT} is NP-Hard.

Solution: We proceed by reduction from SAT. Notice that any NP-Hard language suffices to show this claim. The only differences in the proof will be the mapping. We have arbitrarily chosen to use SAT.

We need to generate a mapping $f : \Sigma^* \to \Sigma^*$ that has the following properties:

- $\phi \in \text{SAT} \implies f(\phi) \in L_{\text{HALT}}$
- $\phi \notin \text{SAT} \implies f(\phi) \notin L_{\text{HALT}}$
- f is polytime computable

where ϕ is a Boolean formula.

f outputs the pair (M_ϕ, ε), where M_ϕ is a Turing machine. M_ϕ behaves as follows on input x:

M_ϕ = "On input x:
1. Loop through all possible Boolean variable assignments a over the variables in ϕ.
2. If $\phi(a)$ is true, accept x.
3. If no assignment yields true, loop on x."

Notice that f outputs a Turing machine that is linear in size with respect to ϕ, so f is an efficient mapping.

Suppose $\phi \in \text{SAT}$. Then there is some variable assignment a such that $\phi(a)$ is true, and M eventually accepts ε, so $(M, \varepsilon) \in L_{\text{HALT}}$.

Suppose $\phi \notin \text{SAT}$. Then ϕ evaluated on *any* assignment a is false, and M eventually loops on ε, so $(M, \varepsilon) \notin L_{\text{HALT}}$.

Therefore $\text{SAT} \leq_p L_{\text{HALT}}$.

4. (a) Prove the transitive property for polytime mapping reductions. That is, prove that if $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.

Solution: Because $A \leq_p B$, there is a polytime computable function $f\colon \Sigma^* \to \Sigma^*$. f has the following properties:

- $x \in A \implies f(x) \in B$.
- $x \notin A \implies f(x) \notin B$
- f is polytime computable.

Because $B \leq_p C$, there is a polytime computable function $g\colon \Sigma^* \to \Sigma^*$. g has the following properties:

- $x \in B \implies g(x) \in C$.
- $x \notin B \implies g(x) \notin C$
- g is polytime computable.

Consider the function $g \circ f\colon \Sigma^* \to \Sigma^*$. Because $g \circ f(x) = g(f(x))$, $g \circ f(x)$ has the following properties:

- $x \in A \implies g \circ f(x) \in C$

  This can be seen by noticing that, for $x \in A$, $f(x) \in B$. Because $f(x) \in B$, it follows that $g(f(x)) \in C$.

- $x \notin A \implies g \circ f(x) \notin C$

  By similar reasoning as above, we obtain this conclusion.

- $g \circ f$ is polytime computable

  f is polytime computable, and its output is also of polynomial length in $|x|$.

  g is also polytime computable, but notice that it takes as input $f(x)$, which may be longer than $|x|$. However, because $f(x)$ is polytime computable, $|f(x)|$ is also of polynomial length with respect to $|x|$. This implies that g takes a polynomial amount of time to compute with respect to $|x|$.

  Because both f and g are polytime computable on the input x, their composition is also polytime computable as desired.

(b) Prove that if $A \leq_p B$ and A is NP-Hard, then B is NP-Hard.

Solution: A is NP-Hard so for all languages $L \in \mathsf{NP}$, we have that $L \leq_p A$. Because $A \leq_p B$, we can apply the transitivity of polytime mapping reductions to conclude that for all $L \in \mathsf{NP}$, $L \leq_p B$. Therefore B is NP-Hard.

(c) Prove that if $A \leq_p B$ and $B \in \mathsf{NP}$ then $A \in \mathsf{NP}$.

Solution: We know that $B \in \mathsf{NP}$, therefore we must have an efficient verifier for B, which we will call V_B. We also know that $A \leq_p B$, therefore there exists a polytime computable function f such that $w \in A \leftrightarrow f(w) \in B$. Using this information, we can construct an efficient verifier for A, V_A.
V_A on input (x, c) :

1 Compute $f(x)$

2 Run $V_B(f(x), c)$ and return the same

Analysis:
Case $x \in A$: then $f(x) \in B$, therefore there is some certificate c such that running V_B on $f(x)$ and c would cause the verifier of B to accept. Thus running V_A on this certificate c would cause it to accept.
Case $x \notin A$: then $f(x) \notin B$, therefore there does not exist a certificate c which would result in V_B accepting $f(x)$ and c. Since we know that running V_B on $f(x)$ with any certificate c would cause V_B to reject, this means that V_A will reject x with all possible certificates.
We have shown that our verifier V_A is correct, but to complete our proof, we must show that V_A runs in polynomial time. Since f is a polynomial time computable function, $|f(x)|$ is polynomial in $|x|$. V_B runs in polynomial time by definition, so it

runs in time polyomial with respect to $|f(x)|$ when given $f(x)$ as input. This is also polynomial with respect to $|x|$, using the same reasoning as in part (a). Thus, both steps in V_A run in polynomial time with respect to $|x|$.
Since we have constructed an efficient verifier for A, $A \in NP$.

(d) Show that polytime mapping reduction is stricter than Turing reduction by showing that $A \leq_p B \implies A \leq_T B$. Show that $A \leq_T B \not\Longrightarrow A \leq_p B$ by exhibiting a pair of of languages A, B such that $A \leq_T B$ yet $A \not\leq_p B$ (this shows that the converse is not true).
Hint: There exist languages known to not be in P. Some such languages are

$$\text{GO} = \left\{(G, n) : \begin{array}{c} G \text{ is a configuration of an } n \times n \text{ Go board with} \\ \text{a winning strategy for the starting player} \end{array}\right\}$$

and

$$\text{BOUNDED-HALT} = \left\{(\langle M \rangle, x, k) : \begin{array}{c} k \text{ is a binary number and } M \text{ halts on } x \\ \text{within } k \text{ steps} \end{array}\right\}.$$

Solution: Suppose that $A \leq_p B$. Then there exists a function $f \colon \Sigma^* \to \Sigma^*$ such that:

- $x \in A \implies f(x) \in B$
- $x \notin A \implies f(x) \notin B$
- f is polytime computable

We will use f to construct a decider D_A for A with access to a decider for B which we will denote as D_B.

$D_A =$ "On input x:
1. Run D_B on $f(x)$
2. If D_B accepts $f(x)$, accept x.
3. Otherwise, reject x."

Suppose $x \in A$. Then we have that $f(x) \in B$, so D_B accepts $f(x)$. Therefore D_A accepts x as desired.
Suppose $x \notin A$. Then we have that $f(x) \notin B$, so D_B rejects $f(x)$. Therefore D_A rejects as desired.
In either case, D_A halts so D_A is a decider for A and $A \leq_T B$. This shows that $A \leq_p B \implies A \leq_T B$.
Now we show the converse is false - that is $A \leq_T B \not\Longrightarrow A \leq_p B$.
Let A be either of GO or BOUNDED-HALT. Notice that A is decidable. If A is GO, the size of the board is finite so all possible subsequent board configurations can be enumerated and tested to see if the starting player has a winning strategy. If A is BOUNDED-HALT, the simulator for M need only run for k steps and accept or reject appropriately.

Regardless of our choice for B, we immediately have that $A \leq_T B$ because A is decidable. We will choose $B \in \mathsf{P}$.

Suppose, for the sake of contradiction, we also have that $A \leq_p B$. Then there exists a function $f\colon \Sigma^* \to \Sigma^*$ such that:

- $x \in A \implies f(x) \in B$
- $x \notin A \implies f(x) \notin B$
- f is polytime computable

We can use f to construct a polytime decider for A, which we chose to be not in P. Let B be decidable in polynomial time by D_B. Our polytime decider D_A for A takes x as input and behaves as $D_B(f(x))$. If D_B accepts (resp. rejects) $f(x)$, our constructed decider will also accept (resp. reject) x. Notice that we have the following properties about D_A:

- $x \in A \implies D_A$ accepts x

  If $x \in A$, then $f(x) \in B$. Because $f(x) \in B$, $D_B(f(x))$ accepts $f(x)$ so $D_A(x)$ accepts x.

- $x \notin A \implies D_A$ rejects x

  If $x \notin A$, then $f(x) \notin B$. Because $f(x) \notin B$, $D_B(f(x))$ rejects $f(x)$ so $D_A(x)$ rejects x.

- D_A is polytime computable

  D_A is the same as $D_B(f(x))$ so if $D_B(f(x))$ is polytime computable, so is D_A. Both $f(x)$ and D_B are polytime computable, and polynomials are closed under composition so $D_B(f(x))$ is polytime computable.

But this implies that $A \in \mathsf{P}$, contradicting the fact that $A \notin \mathsf{P}$ (as we chose earlier). Therefore, there can be no polytime mapping reduction from A to B. In other words, $A \not\leq_p B$.

[Another less interesting solution]

This solution applies the fact that there are nuances in the definition of mapping reduction which do not interact nicely with Σ^* and $\varnothing$.

Let A be a decidable language that is neither Σ^* nor $\varnothing$. Let B be $\varnothing$. Then we have $A \leq_T B$ because A is decidable.

However, $L \not\leq_p \Sigma^*$ because there is no way to map $x \notin L$ to $y \notin \Sigma^*$ (no such y exists). Because no such y exists, it is impossible to construct the mapping reduction.

Similarly, $L \not\leq_p \varnothing$ because there is no way to map $x \in L$ to $y \in \varnothing$, because there are no elements in $\varnothing$.

5. We can define the language 0-1 KNAPSACK as follows:

$$\text{0-1 KNAPSACK} = \left\{ (V, W, n, v, C) : \begin{array}{r} |V| = |W| = n \text{ and there exists indices} \\ i \in 1, \ldots, n \text{ such that } \sum_i V[i] \geq v \text{ and} \\ \sum_i W[i] \leq C \end{array} \right\}.$$

This corresponds to the following problem:

- There are n items
- Each item i has a value $V(i)$ and a weight $W(i)$
- We have a target value of at least v
- We have a weight capacity limit of at most C

Is it possible to select a subset of the items such that the total value of the subset is at least v but the total weight of the subset is at most C?

(a) Show that 0-1 KNAPSACK is in NP.

Solution: We begin by showing that 0-1 KNAPSACK $\in$ NP. The verifier operates on instances of 0-1 KNAPSACK and a subset σ of $\{1, 2, \ldots, n\}$. The verifier computes $\Sigma_{i \in \sigma} V[i]$ and checks that $V[i] \geq v$ and computes $\Sigma_{i \in \sigma} W[i]$ and checks that $W[i] \leq C$. If both inequalities hold, then the verifier accepts. Otherwise it rejects.

We now show that the verifier does indeed show that 0-1 KNAPSACK $\in$ NP.

First, this verifier is efficient. Computing the sum of a subset of weights is efficient in the size of the weights W. Similarly, computing the sum of a subset of values is efficient in the size of the values V. All other operations are constant time, so the verifier is efficient with respect to the input size.

Suppose $x \in$ 0-1 KNAPSACK. Then there exists a subset of items satisfying the weight and value criteria. Let the indices of those items be σ. This shows that there exists a witness that "convinces" the verifier that $x \in$ 0-1 KNAPSACK.

Suppose $x \notin$ 0-1 KNAPSACK. Then let σ be any subset of indices. Suppose that the items which correspond to σ satisfy the condition that their total weight is less than the knapsack capacity. Then because $x \notin$ 0-1 KNAPSACK, it must be the case that their total value is less that the target. Suppose instead that the items meet or surpass the target value. Then because $x \notin$ 0-1 KNAPSACK, it must be the case that the total weight of the items surpasses the knapsack capacity. In either case, no σ "convinces" the verifier that $x \in$ 0-1 KNAPSACK.

This allows us to conclude that 0-1 KNAPSACK $\in$ NP.

(b) Now show that 0-1 KNAPSACK is NP-Hard and conclude that 0-1 KNAPSACK is NP-Complete. You can use SUBSET-SUM, an NP-Hard language, to do so.

We define the language SUBSET-SUM as follows:

$$\text{SUBSET-SUM} = \left\{ (S, k) : \text{There exists } S^* \subseteq S \text{ such that } \sum_{s \in S^*} s = k \right\}.$$

That is, $(S, k) \in$ SUBSET-SUM iff there is a subset of numbers in S such that their sum is k.

Solution: We now show that 0-1 KNAPSACK is NP-Hard by showing SUBSET-SUM $\leq_p$ 0-1 KNAPSACK. The reduction f works as follows on input (S, k) as follows:

(1) Fix an ordering on the set S.

(2) For the i^{th} element of S (denoted as s_i), generate an item with weight s_i and value s_i.

(3) Set both v and C to be k and set n to be $|S|$.

(4) Output the items, v, C and n.

First, we show that this reduction is efficient. Step (1) can be performed in linear time. Step (2) can be performed in time linear to $|S|$. Both step (3) and step (4) can be performed in constant time. The sum of these time complexities is polynomial with respect to the input size.

Now we show that, $(S, k) \in$ SUBSET-SUM $\implies f((S, k)) \in$ 0-1 KNAPSACK. Suppose the sum of elements of $S^* \subseteq S$ sums to k. Then we can take the corresponding items. The sum of the items' weight is $k = C$ and the sum of the items' value is $k = v$. Therefore $f((S, k)) \in$ 0-1 KNAPSACK.

Notice that we can either show
$(S, k) \notin$ SUBSET-SUM $\implies f((S, k)) \notin$ 0-1 KNAPSACK or
$f((S, k)) \in$ 0-1 KNAPSACK $\implies (S, k) \in$ SUBSET-SUM.
The two propositions are logically equivalent. The one that we prove depends on which of the two is easiest. We provide the proof of the latter within this proof and provide the proof of the former as a supplement after the conclusion that 0-1 KNAPSACK is NP-Complete.

Suppose $f((S, k)) \in$ 0-1 KNAPSACK. Then we have a set of items where each item has weight equal to its value. Additionally, the target value and capacity are the same. The problem instance this maps from is in SUBSET-SUM because we can use the items that sum to the target weight and value. This is because we have the inequalities:

$$v[1] + v[2] + \cdots + v[n] \geq v$$
$$w[1] + w[2] + \cdots + w[n] \leq C$$

The weights and values produced by $f((S, k))$ are such that $v[i] = w[i]$ for all i, so we can rewrite the above as

$$x_1 + x_2 + \cdots + x_n \geq k$$
$$x_1 + x_2 + \cdots + x_n \leq k$$

where $x_i = v[i] = w[i]$. We can then rewrite this to go back to something that looks more like a subset sum instance.

$$x_a + x_b + \cdots + x_z = k$$

It follows that the items which show that $f((S,k)) \in$ 0-1 KNAPSACK imply that $(S,k) \in$ SUBSET-SUM.

From the above three claims, we have that 0-1 KNAPSACK is NP-Hard. Therefore, we can conclude that 0-1 KNAPSACK is NP-Complete.

[Supplemental proof: $(S,k) \notin$ SUBSET-SUM $\implies f((S,k)) \notin$ 0-1 KNAPSACK]
Suppose $(S,k) \notin$ SUBSET-SUM. Then any subset S' of S satisfies the following:

$$\Sigma_{s \in S'} s \neq k$$

Consider how the mapping worked. If we take any subset of items we will be have two equalities

$$\Sigma_{i \in \subseteq \{1,\ldots,n\}} v[i] = x$$
$$\Sigma_{i \in \subseteq \{1,\ldots,n\}} w[i] = y$$

which will be the same as the following equations, since $s_i = v[i] = w[i]$:

$$\Sigma_{i \in I \subseteq \{1,\ldots,n\}} s_i = z$$
$$\Sigma_{i \in I \subseteq \{1,\ldots,n\}} s_i = z$$

But notice that $z \neq k$ by assumption and that z represents both the weight of the items taken and the value attained from the items taken. If $z < k$, then the items do not satisfy the value requirement. If $z > k$, then the items do not satisfy the capacity requirement. Therefore no subset of items simultaneously satisfies the weight and value requirements, so $f((S,k)) \notin$ 0-1 KNAPSACK.
Since we know that 0-1 KNAPSACK $\in$ NP and 0-1 KNAPSACK is NP-Hard, thus we can conclude that 0-1 KNAPSACK $\in$ NP-Complete.

E 6. Watch the video at `https://www.youtube.com/watch?v=a3ww0gwEszo` (This is a link to the YouTube song "Find the Longest Path.") List as many 376-relevant concepts from the song as you can. Have fun! (*Please note:* If you have any language barriers or need any type of accessibility assistance, reach out to course staff at `eecs376-F20@umich.edu`).

Solution: