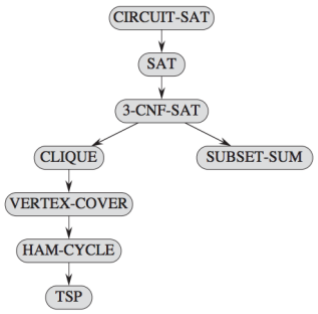
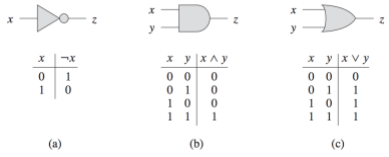
NP-complete:
1, decision problem (0-1):
2, given a directed graph G, vertices u and , and an integer k, does a path exist from u to consisting of at most k edges?
Reduction:


reduction tree:




circuit-sat:



| $x$ | $\neg x$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

(a)

| $x$ | $y$ | $x \wedge y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(b)

| $x$ | $y$ | $x \vee y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

(c)

TSP: formal language:

$$\text{TSP} = \{\langle G, c, k\rangle : G = (V, E) \text{ is a complete graph, } c \text{ is a function from } V \times V \to \mathbb{Z}, k \in \mathbb{Z}, \text{ and } G \text{ has a traveling-salesman tour with cost at most } k\}.$$

***Proof*** We first show that TSP belongs to NP. Given an instance of the problem, we use as a certificate the sequence of $n$ vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most $k$. This process can certainly be done in polynomial time.

To prove that TSP is NP-hard, we show that HAM-CYCLE $\le_P$ TSP. Let $G = (V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$, and we define the cost function $c$ by

$$c(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E, \\ 1 & \text{if } (i,j) \notin E. \end{cases}$$

(Note that because $G$ is undirected, it has no self-loops, and so $c(v, v) = 1$ for all vertices $v \in V$.) The instance of TSP is then $\langle G', c, 0\rangle$, which we can easily create in polynomial time.

We now show that graph $G$ has a hamiltonian cycle if and only if graph $G'$ has a tour of cost at most 0. Suppose that graph $G$ has a hamiltonian cycle $h$. Each edge in $h$ belongs to $E$ and thus has cost 0 in $G'$. Thus, $h$ is a tour in $G'$ with cost 0. Conversely, suppose that graph $G'$ has a tour $h'$ of cost at most 0. Since the costs of the edges in $E'$ are 0 and 1, the cost of tour $h'$ is exactly 0 and each edge on the tour must have cost 0. Therefore, $h'$ contains only edges in $E$. We conclude that $h'$ is a hamiltonian cycle in graph $G$. ■

## 0-1 knapsack problem is a polynomial-time?

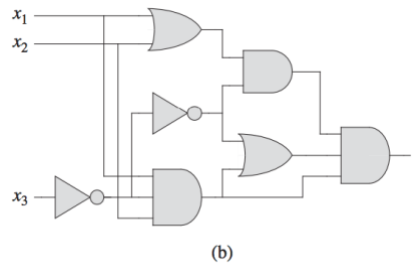
This isn't a polynomial-time algorithm. Recall that the algorithm from Exercise 16.2-2 had running time $\Theta(nW)$ where $W$ was the maximum weight supported by the knapsack. Consider an encoding of the problem. There is a polynomial encoding of each item by giving the binary representation of its index, worth, and weight, represented as some binary string of length $a = \Omega(n)$. We then encode $W$, in polynomial time. This will have length $\Theta(\lg W) = b$. The solution to this problem of length $a+b$ is found in time $\Theta(nW) = \Theta(a * 2^b)$. Thus, the algorithm is actually exponential.

***34.1-5***
Show that if an algorithm makes at most a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then it runs in polynomial time. Also show that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

We show the first half of this exercise by induction on the number of times that we call the polynomial time subroutine. If we only call it zero times, all we are doing is the polynomial amount of extra work, and therefore we have that the whole procedure only takes polynomial time.

Now, suppose we want to show that if we only make $n + 1$ calls to the polynomial time subroutine. Consider the execution of the program up until just before the last call. At this point, by the inductive hypothesis, we have only taken a polynomial amount of time. This means that all of the data that we have constructed so far fits in a polynomial amount of space. This means that whatever argument we pass into the last polynomial time subroutine will have size bounded by some polynomial. The time that the last call takes is then the composition of two polynomials, and is therefore a polynomial itself. So, since the time before the last call was polynomial and the time of the last call was polynomial, the total time taken is polynomial in the input. This proves the claim of the first half of the input.

To see that it could take exponential time if we were to allow polynomially many calls to the subroutine, it suffices to provide a single example. In particular, let our polynomial time subroutine be the function that squares its input. Then our algorithm will take an integer $x$ as input and then square it $\lg(x)$ many times. Since the size of the input is $\lg(x)$, this is only linearly many calls to the subroutine. However, the value of the end result will be $x^{2^{\lg(x)}} = x^x = 2^{x\lg(x)} = 2^{\lg(x)2^{\lg(x)}} \in \omega(2^{2^{\lg(x)}})$. So, the output of the function will require exponentially many bits to represent, and so the whole program could not of taken polynomial time.




(b)

The formula in figure 34.8b is

$$((x_1 \vee x_2) \wedge (\neg(\neg x_3))) \wedge (\neg(\neg x_3) \vee ((x_1) \wedge (\neg x_3) \wedge (x_2))) \wedge ((x_1) \wedge (\neg x_3) \wedge (x_2))$$

We can cancel out the double negation to get that this is the same expression as

$$((x_1 \vee x_2) \wedge (x_3)) \wedge ((x_3) \vee ((x_1) \wedge (\neg x_3) \wedge (x_2))) \wedge ((x_1) \wedge (\neg x_3) \wedge (x_2))$$

Then, the first clause can only be true if $x_3$ is true. But the last clause can only be true if $\neg x_3$ is true. This would be a contradiction, so we cannot have both the first and last clauses be true, and so the boolean circuit is not satisfiable since we would be taking the and of these two quantities which cannot both be true.

## using reduction to prove L is NP-C:

***Lemma 34.8***
If $L$ is a language such that $L' \le_P L$ for some $L' \in$ NPC, then $L$ is NP-hard. If, in addition, $L \in$ NP, then $L \in$ NPC.

***Proof*** Since $L'$ is NP-complete, for all $L'' \in$ NP, we have $L'' \le_P L'$. By supposition, $L' \le_P L$, and thus by transitivity (Exercise 34.3-2), we have $L'' \le_P L$, which shows that $L$ is NP-hard. If $L \in$ NP, we also have $L \in$ NPC. ■

In other words, by reducing a known NP-complete language $L'$ to $L$, we implicitly reduce every language in NP to $L$. Thus, Lemma 34.8 gives us a method for proving that a language $L$ is NP-complete:

1. Prove $L \in$ NP.
2. Select a known NP-complete language $L'$.
3. Describe an algorithm that computes a function $f$ mapping every instance $x \in \{0,1\}^*$ of $L'$ to an instance $f(x)$ of $L$.
4. Prove that the function $f$ satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.
5. Prove that the algorithm computing $f$ runs in polynomial time.

(Steps 2–5 show that $L$ is NP-hard.) This methodology of reducing from a single known NP-complete language is far simpler than the more complicated process of showing directly how to reduce from every language in NP. Proving CIRCUIT-SAT $\in$ NPC has given us a "foot in the door." Because we know that the circuit-satisfiability problem is NP-complete, we now can prove much more easily that other problems are NP-complete. Moreover, as we develop a catalog of known NP-complete problems, we will have more and more choices for languages from which to reduce.

***Theorem 34.9***
Satisfiability of boolean formulas is NP-complete.

***Proof*** We start by arguing that SAT $\in$ NP. Then we prove that SAT is NP-hard by showing that CIRCUIT-SAT $\le_P$ SAT; by Lemma 34.8, this will prove the theorem.

To show that SAT belongs to NP, we show that a certificate consisting of a satisfying assignment for an input formula $\phi$ can be verified in polynomial time. The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression, much as we did in equation (34.2) above. This task is easy to do in polynomial time. If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable. Thus, the first condition of Lemma 34.8 for NP-completeness holds.

To prove that SAT is NP-hard, we show that CIRCUIT-SAT $\le_P$ SAT. In other words, we need to show how to reduce any instance of circuit satisfiability to an instance of formula satisfiability in polynomial time. We can use induction to express any boolean combinational circuit as a boolean formula. We simply look at the gate that produces the circuit output and inductively express each of the gate's inputs as formulas. We then obtain the formula for the circuit by writing an expression that applies the gate's function to its inputs' formulas.

Unfortunately, this straightforward method does not amount to a polynomial-time reduction. As Exercise 34.4-1 asks you to show, shared subformulas—which arise from gates whose output wires have fan-out of 2 or more—can cause the size of the generated formula to grow exponentially. Thus, the reduction algorithm

***Theorem 34.10***
Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

***Proof*** The argument we used in the proof of Theorem 34.9 to show that SAT $\in$ NP applies equally well here to show that 3-CNF-SAT $\in$ NP. By Lemma 34.8, therefore, we need only show that SAT $\le_P$ 3-CNF-SAT.

We break the reduction algorithm into three basic steps. Each step progressively transforms the input formula $\phi$ closer to the desired 3-conjunctive normal form.

The first step is similar to the one used to prove CIRCUIT-SAT $\le_P$ SAT in Theorem 34.9. First, we construct a binary "parse" tree for the input formula $\phi$, with literals as leaves and connectives as internal nodes. Figure 34.11 shows such a parse tree for the formula

$$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2. \tag{34.3}$$

Should the input formula contain a clause such as the OR of several literals, we use associativity to parenthesize the expression fully so that every internal node in the resulting tree has 1 or 2 children. We can now think of the binary parse tree as a circuit for computing the function.

***Theorem 34.11***
The clique problem is NP-complete.

***Proof*** To show that CLIQUE $\in$ NP, for a given graph $G = (V, E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for $G$. We can check whether $V'$ is a clique in polynomial time by checking whether, for each pair $u, v \in V'$, the edge $(u, v)$ belongs to $E$.

We next prove that 3-CNF-SAT $\le_P$ CLIQUE, which shows that the clique problem is NP-hard. You might be surprised that we should be able to prove such a result, since on the surface logical formulas seem to have little to do with graphs.

The reduction algorithm begins with an instance of 3-CNF-SAT. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a boolean formula in 3-CNF with $k$ clauses. For $r =$

***Theorem 34.15***
The subset-sum problem is NP-complete.

***Proof*** To show that SUBSET-SUM is in NP, for an instance $\langle S, t\rangle$ of the problem, we let the subset $S'$ be the certificate. A verification algorithm can check whether $t = \sum_{s \in S'} s$ in polynomial time.

We now show that 3-CNF-SAT $\le_P$ SUBSET-SUM. Given a 3-CNF formula $\phi$ over variables $x_1, x_2, \ldots, x_n$ with clauses $C_1, C_2, \ldots, C_k$, each containing exactly

three distinct literals, the reduction algorithm constructs an instance $\langle S, t\rangle$ of the subset-sum problem such that $\phi$ is satisfiable if and only if there exists a subset of $S$ whose sum is exactly $t$. Without loss of generality, we make two simplifying assumptions about the formula $\phi$. First, no clause contains both a variable and its negation, for such a clause is automatically satisfied by any assignment of values to the variables. Second, each variable appears in at least one clause, because it does not matter what value is assigned to a variable that appears in no clauses.

***34.3-2***
Show that the $\le_P$ relation is a transitive relation on languages. That is, show that if $L_1 \le_P L_2$ and $L_2 \le_P L_3$, then $L_1 \le_P L_3$.

**Exercise 34.3-2**

Suppose $L_1 \le_P L_2$ and let $f_1$ be the polynomial time reduction function such that $x \in L_1$ if and only if $f_1(x) \in L_2$. Similarly, suppose $L_2 \le_P L_3$ and let $f_2$ be the polynomial time reduction function such that $x \in L_2$ if and only if $f_2(x) \in L_3$. Then we can compute $f_2 \circ f_1$ in polynomial time, and $x \in L_1$ if and only if $f_2(f_1(x)) \in L_3$. Therefore $L_1 \le_P L_3$, so the $\le_P$ relation is transitive.

***34.3-6***
A language $L$ is ***complete*** for a language class $C$ with respect to polynomial-time reductions if $L \in C$ and $L' \le_P L$ for all $L' \in C$. Show that $\emptyset$ and $\{0,1\}^*$ are the only languages in P that are not complete for P with respect to polynomial-time reductions.

**Exercise 34.3-6**

Suppose that $\emptyset$ is complete for $P$. Let $L = \{0,1\}^*$. Then $L$ is clearly in $P$, and there exists a polynomial time reduction function $f$ such that $x \in \emptyset$ if and only if $f(x) \in L$. However, it's never true that $x \in \emptyset$, so this means it's never true that $f(x) \in L$, a contradiction since every input is in $L$. Now suppose $\{0,1\}^*$ is complete for $P$, let $L' = \emptyset$. Then $L'$ is in $P$ and there exists a polynomial time reduction function $f'$. Then $x \in \{0,1\}^*$ if and only if $f'(x) \in L'$. However $x$ is always in $\{0,1\}^*$, so this implies $f'(x) \in L'$ is always true, a contradiction because no binary input is in $L'$.

Finally, let $L$ be some language in $P$ which is not $\emptyset$ or $\{0,1\}^*$, and let $L'$ be any other language in $P$. Let $y_1 \notin L'$ and $y_2 \in L'$. Since $L \in P$, there

exists a polynomial time algorithm $A$ which returns 0 if $x \notin L$ and 1 if $x \in L$. Define $f(x) = y_1$ if $A(x)$ returns 0 and $f(x) = y_2$ if $A(x)$ returns 1. Then $f$ is computable in polynomial time and $x \in L$ if and only if $f(x) \in L'$. Thus, $L' \le_P L$.

***24.3-5***
Professor Newman thinks that he has worked out a simpler proof of correctness for Dijkstra's algorithm. He claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path, and therefore the path-relaxation property applies to every vertex reachable from the source. Show that the professor is mistaken by constructing a directed graph for which Dijkstra's algorithm could relax the edges of a shortest path out of order.

**Exercise 24.3-5**

Consider the graph on 5 vertices $\{a, b, c, d, e\}$, and with edges $(a, b), (b, c), (c, d), (a, e), (e, c)$ all with weight 0. Then, we could pull vertices off of the queue in the ordera, $e, c, b, d$. This would mean that we relax $(c, d)$ before $(b, c)$. However, a shortest pat to $d$ is $(a, b), (b, c), (c, d)$. So, we would be relaxing an edge that appears later on this shortest path before an edge that appears earlier.

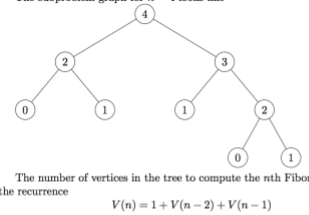
***24.3-10***
Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex $s$ may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from $s$ in this graph.

**Exercise 24.3-10**

The proof of correctness, Theorem 24.6, goes through exactly as stated in the text. The key fact was that $\delta(s, y) \le \delta(s, u)$. It is claimed that this holds because there are no negative edge weights, but in fact that is stronger than is needed. This always holds if $y$ occurs on a shortest path from $s$ to $u$ and $y \neq s$ because all edges on the path from $y$ to $u$ have nonnegative weight. If any had negative weight, this would imply that we had "gone back" to an edge incident with $s$, which implies that a cycle is involved in the path, which would only be the case if it were a negative-weight cycle. However, these are still forbidden.

***15.1-5***
The Fibonacci numbers are defined by recurrence (3.22). Give an $O(n)$-time dynamic-programming algorithm to compute the $n$th Fibonacci number. Draw the subproblem graph. How many vertices and edges are in the graph?

The subproblem graph for $n = 4$ looks like




The number of vertices in the tree to compute the $n$th Fibonacci will follow the recurrence

$$V(n) = 1 + V(n-2) + V(n-1)$$

And has initial condition $V(1) = V(0) = 1$. This has solution $V(n) = 2 * Fib(n) - 1$ which we will check by direct substitution. For the base cases, this is simple to check. Now, by induction, we have

$$V(n) = 1 + 2 * Fib(n-2) - 1 + 2 * Fib(n-1) - 1 = 2 * Fib(n) - 1$$

The number of edegs will satisfy the recurrence

$$E(n) = 2 + E(n-1) + E(n-2)$$

and having base cases $E(1) = E(0) = 0$. So, we show by induction that we have $E(n) = 2 * Fib(n) - 2$. For the base cases it clearly holds, and by induction, we have

$$E(n) = 2 + 2 * Fib(n-1) - 2 + 2 * Fib(n-2) - 2 = 2 * Fib(n) - 2$$

We will present a $O(n)$ bottom up solution that only keeps track of the the two largest subproblems so far, since a subproblem can only depend on the solution to subproblems at most two less for Fibonacci.

***15.3-5***

Suppose that in the rod-cutting problem of Section 15.1, we also had limit $l_i$ on the number of pieces of length $i$ that we are allowed to produce, for $i = 1, 2, \ldots, n$. Show that the optimal-substructure property described in Section 15.1 no longer holds.

**Exercise 15.3-5**

The optimal substructure property doesn't hold because the number of pieces of length $i$ used on one side of the cut affects the number allowed on the other. That is, there is information about the particular solution on one side of the cut that changes what is allowed on the other.

To make this more concrete, suppose the rod was length 4, the values were $l_1 = 2, l_2 = l_3 = l_4 = 1$, and each piece has the same worth regardless of length. Then, if we make our first cut in the middle, we have that the optimal solution for the two rods left over is to cut it in the middle, which isn't allowed because it increases the total number of rods of length 1 to be too large.

***15.4-5***

Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of $n$ numbers.

**Algorithm 3** PRINT-LCS(c,X,Y)

```
n = c[X.length, Y.length]
Initialize an array s of length n
i = X.length and j = Y.length
while i > 0 and j > 0 do
    if x_i == y_j then
        s[n] = x_i
        n = n − 1
        i = i − 1
        j = j − 1
    else if c[i − 1, j] ≥ c[i, j − 1] then
        i = i − 1
    else
        j = j − 1
    end if
end while
for k = 1 to s.length do
    Print s[k]
end for
```

**Algorithm 4** MEMO-LCS-LENGTH-AUX(X,Y,c,b)

```
m = |X|
n = |Y|
if c[m, n]! = 0 or m == 0 or n == 0 then
    return
end if
if x_m == y_n then
    b[m, n] =↖
    c[m,n] =MEMO-LCS-LENGTH-AUX(X[1,..., m-1],Y[1,...,n-1],c,b) +1
else if MEMO − LCS − LENGTH − AUX(X[1,..., m − 1], Y, c, b) ≥
MEMO − LCS − LENGTH − AUX(X, Y[1,..., n − 1], c, b) then
    b[m, n] =↑
    c[m,n] =MEMO-LCS-LENGTH-AUX(X[1,..., m-1],Y,c,b)
else
    b[m, n] =←
    c[m,n] =MEMO-LCS-LENGTH-AUX(X,Y[1,...,n-1],c,b)
end if
```

**Algorithm 5** MEMO-LCS-LENGTH(X,Y)

```
let c be a (passed by reference) |X| by |Y| array initiallized to 0
let b be a (passed by reference) |X| by |Y| array
MEMO-LCS-LENGTH-AUX(X,Y,c,b)
return c and b
```

Then, just run the LCS algorithm on these two lists. The longest common subsequence must be monotone increasing because it is a subsequence of $L'$ which is sorted. It is also the longest monotone increasing subsequence because being a subsequence of $L'$ only adds the restriction that the subsequence must be monotone increasing. Since $|L| = |L'| = n$, and sorting $L$ can be done in $o(n^2)$ time, the final running time will be $O(|L||L'|) = O(n^2)$.

***15-2 Longest palindrome subsequence***

A ***palindrome*** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

```
def lps(str):
    n = len(str)

    # Create a table to store results of subproblems
    L = [[0 for x in range(n)] for x in range(n)]

    # Strings of length 1 are palindrome of length 1
    for i in range(n):
        L[i][i] = 1

    # Build the table. Note that the lower diagonal values o
    # useless and not filled in the process. The values are
    # manner similar to Matrix Chain Multiplication DP solut
    # https://www.geeksforgeeks.org/dynamic-programming-set-
    # cl is length of substring
    for cl in range(2, n+1):
        for i in range(n-cl+1):
            j = i+cl-1
            if str[i] == str[j] and cl == 2:
                L[i][j] = 2
            elif str[i] == str[j]:
                L[i][j] = L[i+1][j-1] + 2
            else:
                L[i][j] = max(L[i][j-1], L[i+1][j]);

    return L[0][n-1]
```

***16.2-3***

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

**Exercise 16.2-3**

At each step just pick the lightest (and most valuable) item that you can pick. To see this solution is optimal, suppose that there were some item $j$ that we included but some smaller, more valuable item $i$ that we didn't. Then, we could replace the item $j$ in our knapsack with the item $i$. it will definitely fit because $i$ is lighter, and it will also increase the total value because $i$ is more valuable.

**Algorithm 1** 0-1 Knapsack(n,W)

```
1:  Initialize an n + 1 by W + 1 table K
2:  for j = 1 to W do
3:      K[0, j] = 0
4:  end for
5:  for i = 1 to n do
6:      K[i, 0] = 0
7:  end for
8:  for i = 1 to n do
9:      for j = 1 to W do
10:         if j < i.weight then
11:             K[i, j] = K[i − 1, j]
12:         end if
13:         K[i, j] = max(K[i − 1, j], K[i − 1, j − i.weight] + i.value)
14:     end for
15: end for
```

***34.5-2***

Given an integer $m \times n$ matrix $A$ and an integer $m$-vector $b$, the ***0-1 integer-programming problem*** asks whether there exists an integer $n$-vector $x$ with elements in the set $\{0, 1\}$ such that $Ax \leq b$. Prove that 0-1 integer programming is NP-complete. (*Hint:* Reduce from 3-CNF-SAT.)



A certificate would be the $n$-vector $x$, and we can verify in polynomial time that $Ax \leq b$, so 0-1 integer linear programming (01LP) is in NP. To prove that 01LP is NP-hard, we show that 3-CNF-SAT $\leq_P$ 01LP. Let $\phi$ be 3-CNF formula with $n$ input variables and $k$ clauses. We construct an instance of 01LP as follows. Let $A$ be a $k + 2n$ by $2n$ matrix. For $1 \leq i \leq k$, set entry $A(i, j)$ to -1 if $1 \leq j \leq n$ and clause $C_i$ contains the literal $x_j$. Otherwise set it to 0. For $n+1 \leq j \leq 2n$, set entry $A(i, j)$ to -1 if clause $C_i$ contains the literal $\neg x_{j-n}$, and 0 otherwise. When $k+1 \leq i \leq k+n$, set $A(i, j) = 1$ if $i - k = j$ or $i - k = j - n$, and 0 otherwise. When $k + n + 1 \leq i \leq k + 2n$, set $A(i, j) = -1$ if $i - k - n = j$ or $i - k - n = j - n$, and 0 otherwise. Let $b$ be a $k + 2n$-vector. Set the first $k$ entries to -1, the next $n$ entries to 1, and the last $n$ entries to -1. It is clear that we can construct $A$ and $b$ in polynomial time.

We now show that $\phi$ has a satisfying assignment if and only if there exists a 0-1 vector $x$ such that $Ax \leq b$. First, suppose $\phi$ has a satisfying assignment. For $1 \leq i \leq n$, f $x_i$ is true, make $x[i] = 1$ and $x[n + i] = 0$. If $x_i$ is false, set $x[i] = 0$ and $x[n + i] = 1$. Since clause $C_i$ is satisfied, there must exist some literal in it which makes it true. If it is $x_j$, then $x[j] = 1$ and $A(i, j) = -1$, so we get a contribution of -1 to the $i^{th}$ row of $b$. Since every entry in the upper $k$ by $2n$ submatrix of $A$ is nonpositive and every entry of $x$ is nonnegative, there can be no positive contributions to the $i^{th}$ row of $b$. Thus, we are guaranteed that the $i^{th}$ row of $Ax$ is at most -1. The same argument applies if the literal $\neg x_j$ makes clause $i$ true. For $1 \leq m \leq n$, at most one of $x_m$ and $\neg x_m$ can be true, so at most one of $x[m]$ and $x[m + n]$ can be true. When we multiply row $k + m$ by $x$, we get the number of 1's among $x[m]$ and $x[m + n]$. Thus, the $(k + m)^{th}$ row of $b$ is at most 1, as required. Finally, when we multiply row $k + n + m$ of $A$ by $x$, we get negative 1 times the number of 1's among $x[m]$ and $x[m + n]$. Since this is at least 1, the $(k + n + m)^{th}$ row of $b$ is at most -1. Therefore all inequalities are satisfied, so $x$ is a 0-1 solution to $Ax = b$.