1. Suppose you are given a magic black box that somehow answers the following decision problem in polynomial time:

- **Input:** A CNF formula \( \varphi \) with \( n \) variables \( x_1, x_2, \ldots, x_n \).
- **Output:** True if there is an assignment of True or False to each variable that satisfies \( \varphi \).

Using this black box as a subroutine, describe an algorithm that solves the following related search problem in polynomial time:

- **Input:** A CNF formula \( \varphi \) with \( n \) variables \( x_1, \ldots, x_n \).
- **Output:** A truth assignment to the variables that satisfies \( \varphi \), or None if there is no satisfying assignment.

[Hint: You can use the magic box more than once.]

**Solution:** For any CNF formula \( \varphi \) with variables \( x_1, \ldots, x_n \), let \( \varphi_{x_i=1} \) be the CNF formula obtained from \( \varphi \) by setting \( x_i \) to True and simplifying the formula; if \( x_i \) is a literal in a clause \( C \) we remove the clause \( C \) from the formula, and if \( \neg x_i \) is a literal in a clause \( C \) we remove the \( \neg x_i \) from the clause (note that if \( C \) contains only \( \neg x_i \) then we obtain an empty clause which we interpret as not being satisfiable by any assignment). Similarly, let \( \varphi_{x_i=0} \) be the CNF formula obtained from \( \varphi \) by setting \( x_i \) to False and simplifying.

Suppose \( \text{Sat}(\varphi) \) returns True if \( \varphi \) is satisfiable and False otherwise. Then the following algorithm constructs a satisfying assignment for \( \varphi \) or correctly reports that no such assignment exists.

```plaintext
SatAssignment(\varphi):
if \( \text{Sat}(\varphi) = \text{False} \)
    return None
for \( i \leftarrow 1 \) to \( n \)
    if \( \text{Sat}(\varphi_{x_i=1}) \)
        \( \varphi' \leftarrow \varphi_{x_i=1} \)
        \( A[i] \leftarrow \text{True} \)
    else
        \( \varphi' \leftarrow \varphi_{x_i=0} \)
        \( A[i] \leftarrow \text{False} \)
return \( A[1 \ldots n] \)
```

The correctness of this algorithm follows by induction from the following observation:

**Claim 1.** The CNF formula \( \varphi_{x_i=1} \) is satisfiable if and only if \( \varphi \) has a satisfying assignment where \( x_i = \text{True} \).

**Proof:** First, if \( \varphi_{x_i=1} \) has a satisfying assignment, then we can augment that satisfying assignment by setting \( x_i = \text{True} \) and this will satisfy \( \varphi \) (note that the only clauses we removed from \( \varphi \) to obtain \( \varphi_{x_i=1} \) have \( x_i \) in them, and hence setting \( x_i = \text{True} \) will satisfy all those clauses).

On the other hand, if \( \varphi \) has a satisfying assignment where \( x_i = \text{True} \), then that assignment restricted to the variables other than \( x_i \) will satisfy \( \varphi_{x_i=1} \); the reasoning is tedious.

\[\Box\]
The algorithm runs in polynomial time. Specifically, suppose $\text{Sat}(\varphi)$ runs in $O(N^c)$ time, where $N$ the total size of $\varphi$ (sum of the clause sizes). Then $\text{SatAssignment}(\varphi)$ runs in time $O(nN^c)$ since the formula size is only decreasing in each iteration and there are at most $n$ iterations.
2. An **independent set** in a graph $G$ is a subset $S$ of the vertices of $G$, such that no two vertices in $S$ are connected by an edge in $G$. Suppose you are given a magic black box that somehow answers the following decision problem *in polynomial time*:

- **Input**: An undirected graph $G$ and an integer $k$.
- **Output**: True if $G$ has an independent set of size $k$, and False otherwise.

(a) Using this black box as a subroutine, describe algorithms that solves the following optimization problem *in polynomial time*:

- **Input**: An undirected graph $G$.
- **Output**: The size of the largest independent set in $G$.

[Hint: You've seen this problem before.]

**Solution**: Suppose $\text{IndSet}(V, E, k)$ returns True if the graph $(V, E)$ has an independent set of size $k$, and False otherwise. Then the following algorithm returns the size of the largest independent set in $G$:

```
\text{MaxIndSetSize}(V, E):
  \text{for } k \leftarrow 1 \text{ to } V
    \text{if } \text{IndSet}(V, E, k + 1) = \text{False}
      \text{return } k
```

A graph with $n$ vertices cannot have an independent set of size larger than $n$, so this algorithm must return a value. If $G$ has an independent set of size $k$, then it also has an independent set of size $k - 1$, so the algorithm is correct.

The algorithm clearly runs in polynomial time. Specifically, if $\text{IndSet}(V, E, k)$ runs in $O((V + E)^c)$ time, then $\text{MaxIndSetSize}(V, E)$ runs in $O((V + E)^{c+1})$ time.

Yes, we could have used binary search instead of linear search. Whatever. ■
(b) Using this black box as a subroutine, describe algorithms that solves the following search problem in polynomial time:

- **INPUT**: An undirected graph $G$.
- **OUTPUT**: An independent set in $G$ of maximum size.

**Solution (delete vertices)**: I’ll use the algorithm $\text{MaxIndSetSize}(V, E)$ from part (a) as a black box instead. Let $G - v$ denote the graph obtained from $G$ by deleting vertex $v$, and let $G - N(v)$ denote the graph obtained from $G$ by deleting $v$ and all neighbors of $v$.

```plaintext
MaxIndSet(G):
S ← ∅
k ← MaxIndSetSize(G)
for all vertices v of G
    if MaxIndSetSize(G - v) = k
        G ← G - v
    else
        G ← G - N(v)
        add v to S
return S
```

Correctness of this algorithm follows inductively from the following claims:

**Claim 2.** $\text{MaxIndSetSize}(G - v) = k$ if and only if $G$ has an independent set of size $k$ that excludes $v$.

**Proof:** Every independent set in $G - v$ is also an independent set in $G$; it follows that $\text{MaxIndSetSize}(G - v) \leq k$.

Suppose $G$ has an independent set $S$ of size $k$ that does excludes $v$. Then $S$ is also an independent set of size $k$ in $G - v$, so $\text{MaxIndSetSize}(G - v)$ is at least $k$, and therefore equal to $k$.

On the other hand, suppose $G - v$ has an independent set $S$ of size $k$. Then $S$ is also a maximum independent set of $G$ (because $|S| = k$) that excludes $v$. □

The algorithm clearly runs in polynomial time.

**Solution (add edges)**: I’ll use the algorithm $\text{MaxIndSetSize}(V, E)$ from part (a) as a black box instead. Let $G + uv$ denote the graph obtained from $G$ by adding edge $uv$.

```plaintext
MaxIndSet(G):
k ← MaxIndSetSize(G)
if k = 1
    return any vertex
for all vertices u
    for all vertices v
        if $u \neq v$ and $uv$ is not an edge
            if MaxIndSetSize(G + uv) = k
                G ← G + uv
S ← ∅
for all vertices v
    if deg(v) < V - 1
        add v to S
return S
```
The algorithm adds every edge it can without changing the maximum independent set size. Let $G'$ denote the final graph. Any independent set in $G'$ is also an independent set in the original input graph $G$. Moreover, the largest independent set in $G'$ is also a largest independent set in $G$. Thus, to prove the algorithm correct, we need to prove the following claims about the final graph $G'$:

**Claim 3.** The maximum independent set in $G'$ is unique.

**Proof:** Suppose the final graph $G'$ has more than two maximum independent sets $A$ and $B$. Pick any vertex $u \in A \setminus B$ and any other vertex $v \in A$. The set $B$ is still an independent set in the graph $G' + uv$. Thus, when the algorithm considered edge $uv$, it would have added $uv$ to the graph, contradicting the assumption that $A$ is an independent set.

**Claim 4.** Suppose $k > 1$. The unique maximum independent set of $G'$ contains vertex $v$ if and only if $\text{deg}(v) < V - 1$.

**Proof:** Let $S$ be the unique maximum independent set of $G'$, and let $v$ be any vertex of $G$. If $v \in S$, then $v$ has degree at most $V - k < V - 1$, because $v$ is disconnected from every other vertex in $S$.

On the other hand, suppose $\text{deg}(v) < V - 1$ but $v \notin S$. Then there must be at least vertex $u$ such that $uv$ is not an edge in $G'$. Because $v \notin S$, the set $S$ is still an independent set in $G' + uv$. Thus, when the algorithm considered edge $uv$, it would have added $uv$ to the graph, and we have a contradiction.

The algorithm clearly runs in polynomial time.
To think about later:

3. Formally, a **proper coloring** of a graph \( G = (V, E) \) is a function \( c: V \rightarrow \{1, 2, \ldots, k\} \), for some integer \( k \), such that \( c(u) \neq c(v) \) for all \( uv \in E \). Less formally, a valid coloring assigns each vertex of \( G \) a color, such that every edge in \( G \) has endpoints with different colors. The **chromatic number** of a graph is the minimum number of colors in a proper coloring of \( G \).

   Suppose you are given a magic black box that somehow answers the following decision problem in polynomial time:

   - **INPUT:** An undirected graph \( G \) and an integer \( k \).
   - **OUTPUT:** True if \( G \) has a proper coloring with \( k \) colors, and False otherwise.

   Using this black box as a subroutine, describe an algorithm that solves the following **coloring problem** in polynomial time:

   - **INPUT:** An undirected graph \( G \).
   - **OUTPUT:** A valid coloring of \( G \) using the minimum possible number of colors.

   *[Hint: You can use the magic box more than once. The input to the magic box is a graph and only a graph, meaning only vertices and edges.]*

   **Solution:** First we build an algorithm to compute the minimum number of colors in any valid coloring.

   ```
   CHROMATICNUMBER(G):
   for k ← V down to 1
     if COLORABLE(G, k - 1) = False
       return k
   ```

   Given a graph \( G = (V, E) \) with \( n \) vertices \( v_1, v_2, \ldots, v_n \), the following algorithm computes an array \( \text{color}[1..n] \) describing a valid coloring of \( G \) with the minimum number of colors.

   ```
   COLORING(G):
   k ← CHROMATICNUMBER(G)
   \langle—— add a disjoint clique of size \( k \) ——\rangle
   H ← G
   for c ← 1 to k
     add vertex \( z_c \) to \( G \)
     for i ← 1 to c - 1
       add edge \( z_i z_c \) to \( H \)
   \langle—— for each vertex, try each color ——\rangle
   for i ← 1 to n
     for c ← 1 to k
       add edge \( v_i z_c \) to \( H \)
     for c ← 1 to k
       remove edge \( v_i z_c \) from \( H \)
       if COLORABLE(H, k) = True
         \( \text{color}[i] \leftarrow c \)
         break inner loop
       add edge \( v_i z_c \) from \( H \)
   return \( \text{color}[1..n] \)
   ```
In any \( k \)-coloring of \( H \), the new vertices \( z_1, \ldots, z_k \) must have \( k \) distinct colors, because every pair of those vertices is connected. We assign \( \text{color}[i] \leftarrow c \) to indicate that there is a \( k \)-coloring of \( H \) in which \( v_i \) has the same color as \( z_c \). When the algorithm terminates, \( \text{color}[1..n] \) describes a valid \( k \)-coloring of \( G \).

To prove that the algorithm is correct, we must prove that for all \( i \), when the \( i \)th iteration of the outer loop ends, \( G \) has a valid \( k \)-coloring that is consistent with the partial coloring \( \text{color}[1..i] \). Fix an integer \( i \). The inductive hypothesis implies that when the \( i \)th iteration of the outer loop begins, \( G \) has a \( k \)-coloring consistent with the first \( i - 1 \) assigned colors. (The base case \( i = 0 \) is trivial.) If we connect \( v_i \) to every new vertices except \( z_c \), then \( v_i \) must have the same color as \( z_c \) in any valid \( k \)-coloring. Thus, the call to \textsc{Colorable} inside the inner loop returns \text{True} if and only if \( H \) has a \( k \)-coloring in which \( v_i \) has the same color as \( z_c \) (and the previous \( i - 1 \) vertices are also colored). So \textsc{Colorable} must return \text{True} during the second inner loop, which completes the inductive proof.

This algorithm makes \( O(kn) = O(n^2) \) calls to \textsc{Colorable}, and therefore runs in polynomial time.

\[ \blacksquare \]