EECS 376: Foundations of Computer Science
University of Michigan, Winter 2024

Solutions to Homework 7

This homework has 8 questions, for a total of 100 points and 5 extra-credit points.
Unless otherwise stated, each question requires *clear*, *logically correct*, and *sufficient* justification to convince the reader.
For bonus/extra-credit questions, we will provide very limited guidance in office hours and on 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 name(s). You must *write your own solution* for all problems and *may not use any other student's write-up.*

(0 pts) 0. **Before you start; before you submit.**

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

Solution:

(10 pts) 1. **Self assessment.**

Carefully read and understand the posted solutions to the previous homework; you may also find the video "walkthroughs" in the Canvas Media Gallery helpful. Identify one part for which your own solution has the most room for improvement (e.g., has unsound reasoning, doesn't show what was required, could be significantly clearer or better organized, etc.). Copy or screenshot this solution, then in a few sentences, explain what was deficient and how it could be fixed.

(Alternatively, if you think one of your solutions is significantly *better* than the posted one, copy it here and explain why you think it is better.)

If you didn't turn in the previous homework, then (1) state that you didn't turn it in, and (2) pick a problem that you think is particularly challenging from the previous homework, and explain the answer in your own words. You may reference the answer key, but your answer should be in your own words.

Solution:

2. **Deciding undecidability.**

For each of the following languages, show that it is undecidable via a Turing reduction from a language we have already shown is undecidable.

(As a refresher, you may wish to re-read Handout 2: Turing Reductions.)

(10 pts) (a) $L_{\text{DISJOINT}} = \{(\langle M_1\rangle, \langle M_2\rangle) : M_1, M_2 \text{ are TMs and } L(M_1) \cap L(M_2) = \emptyset\}$.

Solution: We show that L_{DISJOINT} is undecidable by showing that $L_{\text{ACC}} \leq_T L_{\text{DISJOINT}}$ (and using the fact that L_{ACC} is undecidable). Let D_{DISJOINT} be an oracle ("black box") that decides L_{DISJOINT}. Using D_{DISJOINT}, we construct the following decider D_{ACC} for L_{ACC}.

```
1: function $D_{\text{ACC}}(\langle M \rangle, x)$
2:     construct the Turing machine "$M_1(w)$: accept"
3:     construct the Turing machine "$M_2(w)$: output $M(x)$"
4:     output the opposite of $D_{\text{DISJOINT}}(\langle M_1 \rangle, \langle M_2 \rangle)$
```

Let us first understand the behavior of M_1 and M_2. First, observe that M_1 accepts all w and hence $L(M_1) = \Sigma^*$. Second, observe that if M accepts x, then by construction, M_2 accepts every w, so $L(M_2) = \Sigma^*$ and $(\langle M_1 \rangle, \langle M_2 \rangle) \notin L_{\text{DISJOINT}}$, since $L(M_1) \cap L(M_2) = \Sigma^* \neq \emptyset$. If M does not accept x, then by construction, M_2 does not accept any input, so $L(M_2) = \emptyset$ and $(\langle M_1 \rangle, \langle M_2 \rangle) \in L_{\text{DISJOINT}}$ since $L(M_1) \cap L(M_2) = \emptyset$. So, we have shown the key property that $(\langle M \rangle, x) \in L_{\text{ACC}}$ *if and only if* $(\langle M_1 \rangle, \langle M_2 \rangle) \notin L_{\text{DISJOINT}}$.

Correctness analysis. We now show that D_{ACC} decides L_{ACC}. First, D_{ACC} halts on all inputs—i.e., it decides *some* language—because it just constructs the code of Turing machines M_1, M_2 (from $\langle M \rangle$ and x) and invokes D_{DISJOINT} on them, and D_{DISJOINT} halts on all inputs by hypothesis.

(Importantly, observe that while M_2 *might loop*, D_{ACC} *does not run/simulate* M_2—it just *constructs the code* of M_2 and invokes D_{DISJOINT} on it. And since D_{DISJOINT} does not loop, D_{ACC} does not loop either!)

By the key property above, the hypothesis on D_{DISJOINT}, and the code of D_{ACC},

$$\begin{aligned}(\langle M \rangle, x) \in L_{\text{ACC}} &\iff (\langle M_1 \rangle, \langle M_2 \rangle) \notin L_{\text{DISJOINT}} \\ &\iff D_{\text{DISJOINT}}(\langle M_1 \rangle, \langle M_2 \rangle) \text{ rejects} \\ &\iff D_{\text{ACC}}(\langle M \rangle, x) \text{ accepts.}\end{aligned}$$

Therefore, by definition, D_{ACC} decides L_{ACC}, as needed.

Here's an alternative solution using $L_\emptyset = \{\langle M \rangle : M \text{ is a TM and } L(M) = \emptyset\}$. From the course notes, we know that $L_\emptyset$ is undecidable. So, it suffices to show that $L_\emptyset \leq_T L_{\text{DISJOINT}}$.

Let D_{DISJOINT} be an oracle that decides L_{DISJOINT}. Using D_{DISJOINT}, we construct the following decider $D_\emptyset$ for $L_\emptyset$.

```
1: function $D_\emptyset(\langle M \rangle)$
2:     output $D_{\text{DISJOINT}}(\langle M \rangle, \langle M \rangle)$
```

Clearly, $D_\emptyset$ is a decider as it halts on all inputs (since D_{DISJOINT} does). Next, observe that $D_\emptyset$ decides $L_\emptyset$ because $L(M) = L(M) \cap L(M)$, and so $\langle M \rangle \in L_\emptyset$ if and only if $(\langle M \rangle, \langle M \rangle) \in L_{\text{DISJOINT}}$.

(10 pts) (b) Let $S = \{\langle 183\rangle, \langle 280\rangle, \langle 281\rangle, \langle 370\rangle, \langle 376\rangle\}$ and define the language

$$L_{\text{EECS}} = \{\langle M\rangle : M \text{ is a TM that accepts every string in } S \text{ and loops on every other string}\}\ .$$

Solution: We show that L_{EECS} is undecidable by showing that $L_{\text{ACC}} \le_T L_{\text{EECS}}$ (and using the fact that L_{ACC} is undecidable). Let D_{EECS} be an oracle ("black box") that decides L_{EECS}. Using D_{EECS}, we construct the following decider D_{ACC} for L_{ACC}.

```
function $D_{\text{ACC}}(\langle M\rangle, x)$
    construct the following Turing machine $M'$:
    function $M'(w)$
        if $w \in S$ then output $M(x)$
        loop forever
    output $D_{\text{EECS}}(\langle M'\rangle)$
```

Let us first understand the behavior of M'. Observe that if M accepts x, then by construction, M' accepts every string $w \in S$ and loops on every other string, so $\langle M'\rangle \in L_{\text{EECS}}$. If M does not accept x, then M' does not accept any string at all (it rejects or loops on any input), so $\langle M'\rangle \notin L_{\text{EECS}}$. So, we have shown the key property that $(\langle M\rangle, x) \in L_{\text{ACC}}$ *if and only if* $\langle M'\rangle \in L_{\text{EECS}}$.

Correctness analysis. We now show that D_{ACC} decides L_{ACC}. First, D_{ACC} halts on all inputs—i.e., it decides *some* language—because it just constructs the code of Turing machine M' (from $\langle M\rangle$ and x) and invokes D_{EECS} on it, and D_{EECS} halts on all inputs by hypothesis.

(Importantly, observe that while M' *does loop* on some inputs, D_{ACC} *does not run/simulate M'*—it just *constructs the code* of M' and invokes D_{EECS} on it. And since D_{EECS} does not loop, D_{ACC} does not loop either!)

By the key property above, the hypothesis on D_{EECS}, and the code of D_{ACC},

$$\begin{aligned}(\langle M\rangle, x) \in L_{\text{ACC}} &\iff \langle M'\rangle \in L_{\text{EECS}} \\ &\iff D_{\text{EECS}}(\langle M'\rangle) \text{ accepts} \\ &\iff D_{\text{ACC}}(\langle M\rangle, x) \text{ accepts.}\end{aligned}$$

Therefore, by definition, D_{ACC} decides L_{ACC}, as needed.

Alternatively, we could have used the key property to analyze correctness as follows. First, we follow the $\implies$ implications above that get that $(\langle M\rangle, x) \in L_{\text{ACC}} \implies D_{\text{ACC}}(\langle M\rangle, x)$ accepts. Then, we also use the key property to see that

$$\begin{aligned}(\langle M\rangle, x) \notin L_{\text{ACC}} &\implies \langle M'\rangle \notin L_{\text{EECS}} \\ &\implies D_{\text{EECS}}(\langle M'\rangle) \text{ rejects} \implies D_{\text{ACC}}(\langle M\rangle, x) \text{ rejects.}\end{aligned}$$

Therefore, D_{ACC} decides L_{ACC}.

3. **NP languages.** Prove that each of the following languages is in NP.

(7 pts) (a) AffordableMenu $= \{\langle L, k\rangle : L$ is a list of menu items with positive integer prices, and there are at least k distinct menu items that have total price *exactly* $10k\}$.

Solution: To show that AffordableMenu $\in$ NP, we construct an efficient verifier for it.

A valid certificate for instance $\langle L, k\rangle$ is a list of at least k distinct menu items whose total price is exactly $10k$. The following verifier checks that the presented certificate satisfies these constraints.

```
1: function V(⟨L, k⟩, C)
2:     Check that C is a subset of L of cardinality |C| ≥ k. If not, reject.
3:     Check that the total price of the items in C is exactly 10k. If not, reject.
4:     Accept.
```

Correctness: we claim that $\langle L, k\rangle \in$ AffordableMenu *if and only if* there exists some C that makes $V(\langle L, k\rangle, C)$ accept.

If $\langle L, k\rangle \in$ AffordableMenu, then by definition there are at least k distinct menu items that have total price $10k$. Letting C be the list of these items makes $V(\langle L, k\rangle, C)$ accept.

For the other direction, if $V(\langle L, k\rangle, C)$ accepts, then by definition of V's code, C is a subset of L of size $|C| \geq k$, and the total price of the items in C is exactly $10k$. By definition of the language, $\langle L, k\rangle \in$ AffordableMenu.

Efficiency: We claim that V runs in time polynomial in the length of its first input $\langle L, k\rangle$. Checking that C is a subset of L can be done in polynomial time, because it simply involves checking that C has no more elements than L, and that each element of C is distinct and is an element of L. Checking whether the total price of the elements in C equals $10k$ can be done in polynomial time because it involves looking up and summing $|C| \leq |L|$ prices in L, and both the cardinality $|L|$ and the sizes of the prices are at most the size of $\langle L, k\rangle$.

(7 pts) (b) IndependentSet $= \{\langle G, k\rangle : G$ is an undirected graph with an independent set of size $k\}$. An independent set in a graph is a subset C of the vertices for which the graph has no edge between any pair of vertices in C.

Solution: To show that IndependentSet $\in$ NP, we construct an efficient verifier for it.

A valid certificate for instance $\langle G, k\rangle$ is an independent set of size k in G An efficient verifier is described below:

```
1: function $V(\langle G,k\rangle, C)$
2:     Check if $C$ is a subset of vertices in $G$. If not, reject.
3:     Check if its cardinality $|C| = k$. If not, reject.
4:     for each pair of (distinct) vertices $u, v \in C$ do
5:         check if edge $(u,v)$ is in $G$. If so, reject.
6:     Accept.
```

Correctness: If $\langle G,k\rangle \in$ IndependentSet, then there exists some subset C of k vertices in G having no edge between any pair. Such a C makes $V(\langle G,k\rangle, C)$ accept, as needed.

Conversely, if $V(\langle G,k\rangle, C)$ accepts, then by the checks that V performs, C is an independent set of size k in G, so $\langle G,k\rangle \in$ IndependentSet, as needed.

Efficiency: Let n denote the number of vertices in G, which is at most the size of $\langle G,k\rangle$. Checking if C is a subset of k vertices in G takes $O(n)$ time. Checking if an edge exists between each pairs of vertices in C takes $O(|C|^2) = O(n^2)$ time. Thus, the running time is polynomial in the input size, i.e., the verifier is efficient.

4. **Understanding P, NP, and poly-time mapping reductions.**

(7 pts) (a) We claim that $\mathsf{P} \subseteq \mathsf{NP}$, i.e., for any language $L \in \mathsf{P}$, we have that $L \in \mathsf{NP}$ as well. Here is an *incomplete* proof of this fact, which you will complete.

By the hypothesis that $L \in \mathsf{P}$, there is an efficient Turing machine M that decides L. We define the following efficient verifier V for L.

```
1: function $V(x, c)$
2:     [MISSING PSEUDOCODE]
```

State what the missing pseudocode should be, and prove that V is indeed an efficient verifier for L.

Solution: The missing pseudocode is "output $M(x)$ (ignoring c)."

For efficiency, V runs in polynomial time in the length of its first input x, because M runs in polynomial time in the length of its input.

For correctness, notice that if $x \in L$, then $M(x)$ accepts, so there exists some c that makes $V(x,c)$ accept. (In fact, $V(x,c)$ accepts for *every* c, but the existence of some c is enough for our purposes.) If $x \notin L$, then $M(x)$ rejects, so $V(x,c)$ rejects for every c.

Therefore, V is an efficient verifier for L, by definition.

(10 pts) (b) Show that poly-time mapping reductions are transitive. That is, if $A \le_p B$ and $B \le_p C$, then $A \le_p C$.

Solution: Because $A \le_p B$, there is a poly-time computable function $f\colon \Sigma^* \to \Sigma^*$ where

$$x \in A \iff f(x) \in B\ .$$

Similarly, because $B \leq_p C$, there is a poly-time computable function $g\colon \Sigma^* \to \Sigma^*$ where

$$y \in B \iff g(y) \in C \ .$$

Consider the function $h\colon \Sigma^* \to \Sigma^*$ defined as $h(x) = g(f(x))$ (that is, $h = g \circ f$ is the *composition* of g and f). Observe that by the above properties,

$$x \in A \iff f(x) \in B \iff h(x) = g(f(x)) \in C \ .$$

Finally, we claim that h is poly-time computable, and hence $A \leq_p C$.
To prove the claim, we make the following observations.

- Since f is poly-time computable, on any input x its output $y = f(x)$ can be computed in time $O(|x|^c)$ for some constant c. In particular, the output *length* $|y| = O(|x|^c)$, because y must be written down as part of outputting it.
- Next, since g is also poly-time computable, on any input y, the output $g(y)$ can be computed in time $O(|y|^{c'})$ for some constant c'.
- Therefore, $h(x) = g(f(x))$ is computable in time

$$O(|x|^c) + O(|f(x)|^{c'}) = O(|x|^c + |x|^{c \cdot c'}) = O(|x|^{\max\{c, c \cdot c'\}}) \ ,$$

  which is polynomial in $|x|$ because both c and $c \cdot c'$ are constants. (Either one could potentially be largest, because it might be that $c' < 1$, but this does not matter for the final result.)

So, we have shown that on any input x, the output $h(x)$ can be computed in time polynomial in $|x|$, i.e., h is poly-time computable, as claimed.

(10 pts) (c) Let $C = A \cup B$ where both A, B are languages in NP. State, with proof, whether C is in NP for *all*, *some* (but not all), or *no* such A, B.

Solution: This holds for *all* such A, B.
Let V_A, V_B respectively be associated efficient verifiers for A, B. Below we give an efficient verifier V_C for the language $C = A \cup B$.
Conceptually, a certificate for an instance x of C is just a certificate for x as an instance of A, or as an instance of B. Since A and B are generic NP languages, we don't know the expected "format" of either kind of certificate, so we just treat certificates as opaque strings. The verifier simply checks the certificate using the verifiers V_A and V_B, accepting if either one accepts.

```
1: function $V_C(x, c)$)
2:     if $V_A(x, c)$ accepts then accept
3:     if $V_B(x, c)$ accepts then accept
4:     reject
```

We show that V_C is indeed an efficient verifier for $A \cup B$. It is clear that V_C runs in polynomial time in its first argument, because both V_A, V_B do. (To be precise, we are also relying on the fact that without loss of generality, c has size polynomial in $|x|$. This allows V_C to call both V_A, V_B with it, in polynomial time.)

If $x \in A \cup B$, then either $x \in A$, in which case there exists some c such that $V_A(x, c)$ accepts; or $x \in B$, in which case there exist some c_B such that $V_B(x, c)$ accepts. Either way, for such c, we have that $V_C(x, c)$ accepts, as required.

If $x \notin A \cup B$, then $x \notin A$ and $x \notin B$, so $V_A(x, c)$ and $V_B(x, c)$ both reject for *every* c, hence $V_C(x, c)$ also rejects for every c, as required.

An alternative correct solution is to treat the certificate for V_C as a pair $c_C = (c_A, c_B)$, and run each verifier on x with the corresponding certificate, using the same logic to make a decision. The analysis proceeds similarly.

As a remark, the above proof easily adapts to show that the *intersection* of any two languages in NP is also in NP. However, the proof does not easily adapt to work for the *complement*, *difference*, or *set difference*—as an exercise, try to see why. Indeed, it is *unknown* whether NP is closed under any of these operations!

5. **Understanding coNP.**

This question explores the complexity class $\mathsf{coNP} = \{\overline{L} : L \in \mathsf{NP}\}$, i.e., the set of languages whose complement languages are in NP. (Note that this is *not* the complement of NP itself, which is the set of languages that are not in NP.)

Recall that conceptually, NP is the set of languages whose "yes" instances can be verified efficiently, given suitable certificates. Symmetrically, coNP is the set of languages whose "no" instances can be verified efficiently (given suitable certificates).

(5 pts) (a) Prove that P is closed under set complement. That is, for any $L \in \mathsf{P}$, we have that $\overline{L} \in \mathsf{P}$.

Solution: If $L \in P$, there exists an efficient decider M_L for L. We can construct an efficient decider $M_{\overline{L}}$ that, on input x, simply runs $M(x)$ and outputs the opposite answer. (Equivalently, we can just swap the accept and reject states of M.) It is obvious that $M_{\overline{L}}$ is an efficient decider, and $M_{\overline{L}}(x)$ accepts if and only if $M(x)$ rejects, which happens if and only if $x \notin L$. Therefore, $M_{\overline{L}}$ decides $\overline{L}$, as desired.

(9 pts) (b) Use the previous part to prove that if $\mathsf{P} = \mathsf{NP}$, the following set inclusions hold: (i) $\mathsf{NP} \subseteq \mathsf{coNP}$ (that is, for any $L \in \mathsf{NP}$, we have $L \in \mathsf{coNP}$), and (ii) $\mathsf{coNP} \subseteq \mathsf{NP}$.

Solution: Observe that if $\mathsf{P} = \mathsf{NP}$, then because P is closed under set complement by part (a), so is NP. That is, if $L \in \mathsf{NP}$, then $\overline{L} \in \mathsf{NP}$ (which also means $\overline{L} \in \mathsf{NP} \implies L \in \mathsf{NP}$ by definition of complement). Then we have

$$\begin{aligned} L \in \mathsf{NP} &\iff \overline{L} \in \mathsf{NP} && \text{(by set complement closure of NP)} \\ &\iff L = \overline{\overline{L}} \in \mathsf{coNP} && \text{(by definition of coNP)} \end{aligned}$$

Hence $\mathsf{NP} \subseteq \mathsf{coNP}$ and $\mathsf{coNP} \subseteq \mathsf{NP}$.

Alternatively, we might show the two set inclusions individually, by showing the two implications separately.

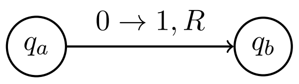
(5 pts) (c) Conclude from the previous part that if $\mathsf{NP} \neq \mathsf{coNP}$, then $\mathsf{P} \neq \mathsf{NP}$.
(Therefore, you could collect the \$1 million Clay Math prize "simply" by proving that $\mathsf{NP} \neq \mathsf{coNP}$!)

Solution: We prove the contrapositive: if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{NP} = \mathsf{coNP}$. By part (b), $\mathsf{P} = \mathsf{NP}$ implies $\mathsf{coNP} \subseteq \mathsf{NP}$ and $\mathsf{NP} \subseteq \mathsf{coNP}$, so $\mathsf{NP} = \mathsf{coNP}$.

(10 pts) 6. **Cook-Levin windows.**

Recall from lecture that one critical ingredient in the proof of the Cook-Levin theorem is to specify which contents of the 2×3 "windows" of the verifier's computation tableau are valid.

Specify *all* valid 2×3 window contents that could appear in two adjacent rows of the tableau, if those rows represent the verifier TM taking the following transition:



You do not need to provide justification.

For simplicity, you can assume that the symbols in the cells of the window are restricted to $\{0, 1, \perp, q_a, q_b\}$. Recall that there can be other symbols in the rows, such as the "boundary" symbol # and "separator" symbol \$, but we will ignore those here. To avoid explicitly writing out too many windows, you can use notation like $0/1/\perp$ to indicate that a cell could contain either 0, 1, or $\perp$. Also, to specify that a cell in the bottom row of a window must contain the same character as the cell above it, you can write "same". For example, writing

0/1	$\perp$	1
same	0	$\perp$

represents the following two windows:

0	$\perp$	1
0	0	$\perp$

1	$\perp$	1
1	0	$\perp$

Be sure to include all valid window contents that represent the head of the TM being in *both* rows of the window, all those that represent the head being in *only one* of the rows of the window, and all those that represent the head not being in the window at all. Do not include any invalid windows.

Solution: The valid window contents are as follows.

0/1/⊥	0/1/⊥	0/1/⊥
same	same	same

0/1/⊥	0/1/⊥	q_a
same	same	1

0/1/⊥	q_a	0
same	1	q_b

q_a	0	0/1/⊥
1	q_b	same

0	0/1/⊥	0/1/⊥
q_b	same	same

(5 EC pts) 7. **Optional extra-credit problem: NP in exponential time.**

Let EXP be the class of all languages that are decidable in exponential time, i.e., in time $O(2^{n^k})$ for some constant k (where n is the length of the input).

It remains unknown whether $\mathsf{NP} = \mathsf{EXP}$, but it is known that $\mathsf{P} \neq \mathsf{EXP}$. Prove that $\mathsf{NP} \subseteq \mathsf{EXP}$. In other words, show that any problem that can be verified efficiently can be decided in exponential time.

Solution: Consider an arbitrary language $L \in \mathsf{NP}$; we will show that $L \in \mathsf{EXP}$ (and thus $\mathsf{NP} \subseteq \mathsf{EXP}$).

By hypothesis, L has an efficient verifier $V(\cdot,\cdot)$ which is polynomial time in the length of the first input, and where $x \in L$ if and only if there exists a certificate c such that $V(x,c)$ accepts. A key observation is that because V runs in at most some polynomial $|x|^k$ time (for some constant $k \geq 1$), it can *read at most* $|x|^k$ *bits* of any given certificate c during its execution. Thus, the accept/reject behavior of $V(x,c)$ is determined by x and *just the first* $|x|^k$ *bits of* c.

With the above observation, to decide L we can simply do a brute-force search, running the verifier with every possible certificate of appropriate length, and accepting if any of these runs accepts. More precisely, we define the following program M:

```
function $M(x)$
    for each certificate $c$ of length at most $|x|^k$ do
        if $V(x,c)$ accepts then accept
    reject
```

We claim that M decides L. If $x \in L$, then by definition there exists some certificate c of length $\leq |x|^k$ such that $V(x,c)$ accepts. Thus, $M(x)$ will find such a c and accept. If $x \notin L$, then $V(x,c)$ rejects for every certificate c, so $M(x)$ rejects, as required.

We now analyze the running time of $M(x)$. There are fewer than 2^{t+1} bitstrings of length at most t, so the loop will iterate fewer than $2^{|x|^k+1}$ times. Each iteration runs the verifier, which takes at most $|x|^k$ time. Thus $M(x)$ takes time at most

$$2^{|x|^k+1} \cdot |x|^k = 2^{|x|^k+1+k\log|x|} = O(2^{|x|^k}).$$

Thus, M is an exponential-time decider for L, so $L \in \mathsf{EXP}$, as claimed.

Solution: Here is an alternative solution. We will show that SAT $\in \mathsf{EXP}$ and conclude that $\mathsf{NP} \subseteq \mathsf{EXP}$ because SAT is NP-hard.

```
function $D(\phi)$
    for each assignment $y$ of variables in $\phi$ do
        Compute $\phi(y)$
        if $\phi(y)$ is true then accept
    reject
```

First, D halts on any input because ϕ has a finite number of possible assignments.

$\phi \in \text{SAT} \implies$ There exists some satisfying assignment y for $\phi \implies D$ accepts

$\phi \notin \text{SAT} \implies$ There does not exist some satisfying assignment y for $\phi \implies D$ rejects

Therefore, we can conclude that D is a decider for SAT. The decider runs in $O(|\phi| \cdot 2^{|\phi|})$ as there are $2^m = O(2^{|\phi|})$ possible assignments, where $m \leq |\phi|$ is the number of variables in ϕ, and it takes $O(|\phi|)$ to check if an assignment satisfies ϕ. Therefore, SAT $\in \mathsf{EXP}$.

Since, for every language $L \in \mathsf{NP}$, we have $L \leq_p$ SAT by the Cook-Levin theorem. So, $L \in \mathsf{EXP}$. Therefore, $\mathsf{NP} \subseteq \mathsf{EXP}$.

In more detail, for any language L in NP, given an input of size n, we can construct a $O(n^{2k})$ Boolean formula that is satisfiable exactly when the input is in language L. Running the SAT solver D on the formula takes time $O(|\phi| \cdot 2^{|\phi|}) = O(n^{2k} \cdot 2^{O(n^{2k})}) = O(2^{n^{k'}})$ for some constant k'. We can therefore construct a decider that runs in exponential time for every language in NP by first converting the input and efficient verifier to a Boolean formula and then running our decider D for SAT on the formula. Therefore, $\mathsf{NP} \subseteq \mathsf{EXP}$.