Computability and Computational Complexity, A.Y. 2025–2026

Guide to the answers

Thursday, September 3, 2026

Exercise 1

Let $\Sigma = \{0, 1\}$ be the set of binary digits and consider the language $L \subset \Sigma^*$ that contains all and only binary strings with alternating `0` and `1` (i.e., every `0` is followed by `1` and vice versa) with at most one exception (repeated digits `00` or `11`). For instance, the following strings belong to L because there is at most one repetition (underlined):

`0` `11` `01` `001` `1010101` `1010101` `0010101` `1001010` ε.

The following don't (they contain more than one pair of adjacent equal digits, `00` or `11`):

`0011` `0100101010001` `1101011` `0101010010100101010` `0100010101010101`.

Moreover, let $L_n = L \cap \bigcup_{i=0}^{n} \Sigma^i = \{s \in L : |s| \leq n\}$ be the subset of L with strings of length n or less.

1.1) For each of the following properties of a Turing machine $\mathcal{M}$, prove whether they are computable or not, invoking Rice's Theorem if possible:

$$\begin{aligned}
\mathcal{P}_1 &= \{\mathcal{M} : \mathcal{M} \text{ decides } L\} \\
\mathcal{P}_2 &= \{\mathcal{M} : \mathcal{M} \text{ decides } L_5\} \\
\mathcal{P}_3 &= \{\mathcal{M} : \mathcal{M} \text{ decides } L \text{ in less than } 10^6 \text{ steps}\} \\
\mathcal{P}_4 &= \{\mathcal{M} : \mathcal{M} \text{ decides } L_5 \text{ in less than } 10^6 \text{ steps}\}
\end{aligned}$$

1.2) Write a single-tape, deterministic Turing machine (with alphabet $\{\sqcup, 0, 1\}$) that decides L.

Hint — *Be careful in $\mathcal{P}_3$ and $\mathcal{P}_4$: we require $\mathcal{M}$ to halt within 10^6 steps regardless of the input. The machine in 1.2 needs only perform a single scan, provided that it remembers two items of information: (i) the last symbol seen, and (ii) whether it has already met an exception (repeated digit) or not.*

Solution 1

1.1)

- $\mathcal{P}_1$ is non-trivial and semantic, hence it isn't computable.
  The property is clearly non-trivial because there are machines that decide L —as proved by the fact that we are going to build one in 1.2— and machines that don't. The property is semantic because it only refers to the recognized language: suppose $L(\mathcal{M}_1) = L(\mathcal{M}_2)$; then either that language is L (and hence $\mathcal{P}_1(\mathcal{M}1) = \mathcal{P}_1(\mathcal{M}_2) =$ true) or it isn't (in which case $\mathcal{P}_1(\mathcal{M}1) = \mathcal{P}_1(\mathcal{M}_2) =$ true). In both cases, $\mathcal{P}_1(\mathcal{M}1) = \mathcal{P}_1(\mathcal{M}_2)$.
  Hence, it satisfies the hypotheses of Rice's Theorem.

- $\mathcal{P}_2$ is non-trivial and semantic, hence it isn't computable.
  Same reasons as above. The very large upper bound to the number of steps guarantees that at least one machine has the property, hence it is non'trivial.

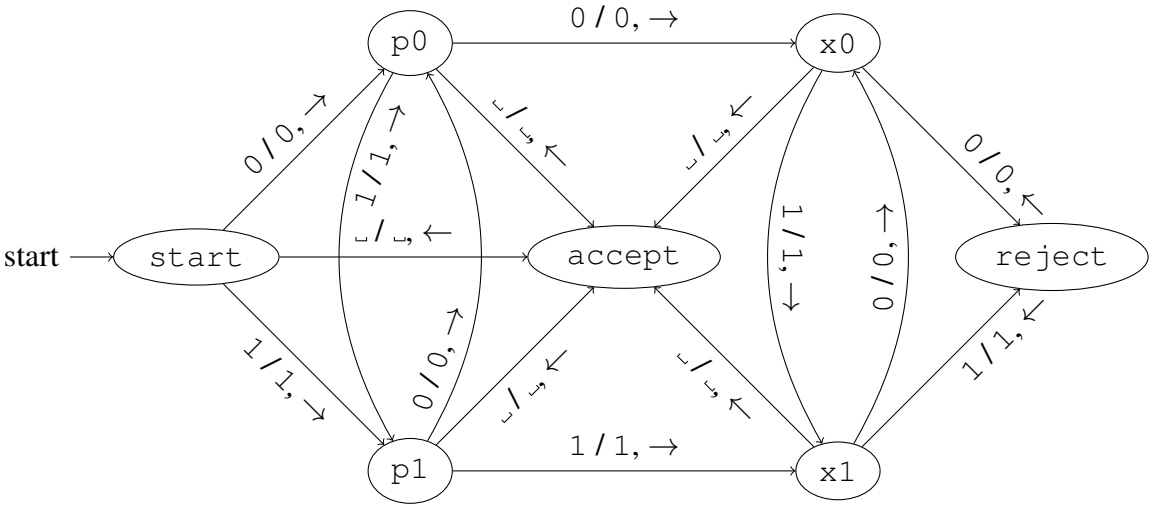
- $\mathcal{P}_3$ is trivial; this means that Rice's theorem isn't applicable, but also that the property is trivially computable.
  Indeed, $\mathcal{P}_3$ is trivially false; no machine can decide all infinite strings in L within a fixed number of steps; if the input x has more than 10^6 symbols, no machine will ever be able to scan it within 10^6 steps, so it cannot determine whether $x \in L$ or not.

- $\mathcal{P}_4$ is non-trivial and non-semantic; Rice's theorem isn't applicable. It is computable, as shown below.
  It is neither trivial, nor semantic: we can take a machine $\mathcal{M}_1$ having property $\mathcal{P}_4$ and add a million dummy steps before halting. The machine still recognizes L_5, but it does not have property $\mathcal{P}_4$.
  In order to test for $\mathcal{M} \in \mathcal{P}_4$, we just need to run $\mathcal{M}$ for at most 10^6 steps on all possible combinations of symbols reachable in 10^6 steps. Since all acceptable strings are much shorter (5 symbols at most), if the test succeeds we are sure that $\mathcal{M} \in \mathcal{P}_4$.

1.2) As suggested need to introduce separate states for all cases that the TM must remember; let's give them some mnemonic name:

- `p0`: previous symbol was a `0`, no exception found yet;
- `p1`: previous symbol was a `1`, no exception found yet;
- `x0`: previous symbol was a `0`, one exception found;
- `x1`: previous symbol was a `1`, one exception found;

None of the states listed above is eligible as initial, because in the beginning no previous symbol is memorized. The stateset above must thus be completed with a separate initial state (`start`); let us introduce two separate halting states for acceptance (`accept`) and rejection (`reject`).



Observations

- Why cannot we just apply to $\mathcal{P}_3$ the same reasoning we made about $\mathcal{P}_4$? Because L also contains strings that are much longer than 10^6 symbols, and if we just limit our tests to shorter strings we miss all of them. A procedure that only tests for strings up to a given size will accept a machine even if it fails on larger inputs (e.g., by rejecting strings that are longer that 10^6 symbols, but still in L).

- It's important to keep a separate initial state, otherwise the machine will "remember" a non-existing previous symbol and discard too many strings.

Exercise 2

Let $L_1, L_2 \in$ **NP**, and suppose **P** $\neq$ **NP**. Does $L_1 \cup L_2 \in$ **NP**? Does $L_1 \setminus L_2 \in$ **NP**?
Try to be as clear and formal as possible.

Solution 2

If $L_1 \in$ **NP**, then there is a non-deterministic Turing machine $\mathcal{N}_1$ that decides L_1 in polynomial time. Likewise, there is a NDTM $\mathcal{N}_2$ such that decides L_2 in polynomial time. Also remember that "acceptance" in non-deterministic machines requires at least one accepting computation branch.
The following NDTM , let's call it $\mathcal{N}_{1\cup2}$, decides $L_1 \cup L_2$ in polynomial time:

- On input x:
  - if $\mathcal{N}_1(x)$ accepts, then accept and halt;
  - if $\mathcal{N}_2(x)$ accepts, then accept and halt;
  - reject and halt.

If $x \in L_1$, then at least one branch in the computation in $\mathcal{N}_1(x)$ will accept, therefore the same branch accepts in $\mathcal{N}_{1\cup2}$. If $x \notin L_1$, then all branches in $\mathcal{N}_1(x)$ will reject and computation will continue with $\mathcal{N}_2(x)$. In this case, if $x \in L_2$ at least one branch in the computation of $\mathcal{N}_2(x)$ will accept, and this causes acceptance by the corresponding branch of the computation of $\mathcal{N}_{1\cup2}$. Finally, if x doesn't belong to neither L_1 or L_2, then **all** computational branches in both $\mathcal{N}_1(x)$ and $\mathcal{N}_2(x)$ will reject, therefore all branches in $\mathcal{N}_{1\cup2}(x)$ end with rejection, as required.
Another way to prove this is to use deterministic computation and certificates. $L_1 \in$ **NP** means that there is a polynomial-time TM $\mathcal{M}_1$ such that $x \in L_1$ if and only if there is a polynomially-sized certificate c such that $\mathcal{M}_1$ accepts (x, c) (likewise for L_2 and corresponding machine M_2). Then $L_1 \cup L_2$ can be decided by the following machine (let's call it $\mathcal{M}_{1\cup2}$) accepting a string x and a certificate c:

- On input (x, c):
  - if $\mathcal{M}_1(x, c)$ accepts, then accept and halt;
  - if $\mathcal{M}_2(x, c)$ accepts, then accept and halt;
  - reject and halt.

If $x \in L_1$, then we can provide a certificate c suitable for $\mathcal{M}_1$; otherwise, if $x \in L_2$, $\mathcal{M}_1(x, c)$ will always reject, but we can provide a certificate suitable for $\mathcal{M}_2$; finally, if $x \notin L_1$ and $x \notin L_2$, then no matter what certificate we provide both $\mathcal{M}_1(x, c)$ and $\mathcal{M}_2(x, c)$ will reject, therefore $\mathcal{M}_{1\cup2}(x)$ will always reject and fail.
As for $L_1 \setminus L_2$, notice that in order for x to belong to the difference we must check both $x \in L_1$ (which again is possible in non-deterministic polynomial time), and $x \notin L_2$. The latter is not guaranteed to be verifiable in non-deterministic polynomial time, unless $L \in$ **coNP**, which we don't know. Therefore we cannot say whether $L_1 \setminus L_2 \in$ **NP** without more information about L_2.

Observations

About $L_1 \setminus L_2$, suppose we define the NDTM $\mathcal{N}_{1\setminus2}$ as:

- On input x:

- if $\mathcal{N}_1(x)$ rejects, then reject and halt;
- if $\mathcal{N}_2(x)$ accepts, then reject and halt;
- accept and halt.

Shouldn't this machine decide $L_1 \setminus L_2$ in non-deterministic polynomial time? After all, it is just the concatenation of two NDTMs: it checks for the two cases that must be rejected ($x \notin L_1$ and $x \in L_2$) and after ruling them out it accepts the remaining strings!
The problem with this is that NDTMs allow for spurious rejections by many computational branches (remember, it is sufficient that one of the branches accepts). Therefore, even if $x \in L_2$, we can expect many branches in $\mathcal{N}_2(x)$ to reject anyway, and $\mathcal{N}_{1\cup 2}$ would accept x even if it shouldn't.
This comes from the fact that, if the NDTM $\mathcal{N}$ recognizes a given language L, its complement "if $\mathcal{N}(x)$ accepts then reject, else accept" does **not** recognize the complement language $\bar{L}$. Remember that acceptance and rejection are not symmetric in **NP**.

Exercise 3

Consider the following classical **NP**-complete languages:

$$\begin{aligned} \text{INDEPENDENT SET} &= \{(G,k) : \text{undirected graph } G \text{ has an independent set of size } k\} \\ \text{VERTEX COVER} &= \{(G,k) : \text{undirected graph } G \text{ has a vertex cover of size } k\}. \end{aligned}$$

Describe a polynomial-time reduction from INDEPENDENT SET to VERTEX COVER.

Hint — *Remember that, given an undirected graph $G = (V, E)$ an independent set is a subset of nodes $V' \subseteq V$ with no connections between any pair (i.e., $v_1, v_2 \in V' \Rightarrow \{v_1, v_2\} \notin E$).*
A vertex cover is a subset of nodes $V' \subseteq V$ such that every edge in E has at least one endpoint in V' (i.e., $e \in E \Rightarrow e \cap V' \neq \emptyset$).
For the reduction observe that, if $V' \subseteq V$ is a vertex cover and $v_1, v_2 \notin V'$, then the edge $\{v_1, v_2\} \notin E$ (but explain why).

Solution 3

See the lecture notes.

Observations

The exercise requires to describe the reduction. This means not only to prove the suggested fact, but also to show how that fact allows us to convert an instance (G, k) of INDEPENDENT SET into the equivalent instance of VERTEX COVER. A correct answer must therefore show *how* to change the instance. In this case, the INDEPENDENT SET instance $(G = (V, E), k)$ is mapped to the equivalent VERTEX COVER instance $(G, |V| - k)$.