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

Guide to the answers (revised version based on student answers to the test)

Friday February 20, 2026, revised on February 27, 2026

Exercise 1

1.1) Show that the set R of recursive languages is closed with respect to:

- language union (i.e., if $L_1, L_2 \in R$ then $L_1 \cup L_2 \in R$),
- language intersection (if $L_1, L_2 \in R$ then $L_1 \cap L_2 \in R$) and
- language complementation (if $L \in R$ then $\bar{L} = \Sigma^* \setminus L \in R$).

1.2) What can be said about the set RE of recursively enumerable languages?

Solution 1

1.1) If languages $L_1, L_2 \in R$, it means that there are Turing machines $\mathcal{M}_1$ and $\mathcal{M}_2$ that decide the two languages:

$$\forall s \in \Sigma^* \qquad \mathcal{M}_1(s) = \begin{cases} 1 & \text{if } s \in L_1 \\ 0 & \text{otherwise} \end{cases} \qquad \mathcal{M}_2(s) = \begin{cases} 1 & \text{if } s \in L_2 \\ 0 & \text{otherwise} \end{cases}$$

In particular, both machines will always halt.

- The following machine will accept s whenever it belongs to L_1 *or* L_2 and reject it otherwise (hence it decides is $s \in L_1 \cup L_2$):
  - Run $\mathcal{M}_1(s)$; if it accepts, then accept and halt;
  - otherwise, run $\mathcal{M}_2(s)$; if it accepts, then accept and halt;
  - otherwise, since both machines rejected s, reject and halt.

  Since the machine decides $L_1 \cup L_2$, language union is still recursive.

- The following machine will accept s whenever it belongs to *both* L_1 *and* L_2 and reject it otherwise (hence it decides is $s \in L_1 \cap L_2$):
  - Run $\mathcal{M}_1(s)$; if it rejects, then reject and halt;
  - otherwise, run $\mathcal{M}_2(s)$; if it rejects, then reject and halt;
  - otherwise, since both machines accepted s, accept and halt.

  The machine decides $L_1 \cap L_2$, intersection is recursive too.

- To decide the complement of language L_1, we just need to reverse $\mathcal{M}_1$'s answer:
  - Run $\mathcal{M}_1(s)$; if it accepts, then reject and halt;
  - otherwise, accept and halt.

This machine decides $\bar{L}_1$ which is therefore recursive.

1.2) If languages $L_1, L_2 \in RE$, the Turing machines $\mathcal{M}_1$ and $\mathcal{M}_2$ are not guaranteed to halt if the string is not in the language (they *recognize* it but are not guaranteed to *decide* it):

$$\forall s \in \Sigma^* \qquad \mathcal{M}_1(s) = \begin{cases} 1 & \text{if } s \in L_1 \\ \neq 1 & \text{otherwise} \end{cases} \qquad \mathcal{M}_2(s) = \begin{cases} 1 & \text{if } s \in L_2 \\ \neq 1 & \text{otherwise} \end{cases}$$

where "$\neq 1$" can be interpreted as "reject or run forever."

- To prove closure with respect to language union, we need to alternate steps of both machines, accepting whenever one of the two halts. The following machine will accept s whenever it belongs to L_1 *or* L_2 (hence it accepts is $L_1 \cup L_2$):
  - Initialize the simulation of $\mathcal{M}_1(s)$ and $\mathcal{M}_2(s)$;
  - repeat:
    * Run the next step of $\mathcal{M}_1(s)$; if it accepts, then accept and halt;
    * otherwise, run the next step of $\mathcal{M}_2(s)$; if it accepts, then accept and halt;

  This machine halts in an accepting state as soon as either $\mathcal{M}_1(s)$ or $\mathcal{M}_2(s)$ accepts, otherwise it runs forever. It accepts $L_1 \cup L_2$, which is therefore recursively enumerable.

- For the intersection, the same construction shown in 1.1 still works. The following machine will accept s whenever it belongs to *both* L_1 *and* L_2 (hence it accepts $L_1 \cap L_2$):
  - Run $\mathcal{M}_1(s)$; if it rejects, then reject and halt;
  - otherwise, run $\mathcal{M}_2(s)$; if it rejects, then reject and halt;
  - otherwise, since both machines accepted s, accept and halt.

  If either $\mathcal{M}_1(s)$ or $\mathcal{M}_2(s)$ doesn't halt, then the machine runs forever, thereby not accepting s, as intended. This machine therefore accepts $L_1 \cap L_2$, which is therefore recursively enumerable.

- Finally RE is *not* closed under complementation. In fact, if both L and $\bar{L}$ are RE, it means that we have both a machine $\mathcal{M}$ that accepts L and a machine $\bar{\mathcal{M}}$ that accepts its complement. Then we could build the following machine that alternates the steps of $\mathcal{M}(s)$ and $\bar{\mathcal{M}}(s)$:
  - Initialize the simulation of $\mathcal{M}(s)$ and $\bar{\mathcal{M}}(s)$;
  - repeat:
    * Run the next step of $\mathcal{M}(s)$; if it accepts, then accept and halt;
    * otherwise, run the next step of $\bar{\mathcal{M}}(s)$; if it accepts, then **reject** and halt;

  We are guaranteed that either $\mathcal{M}(s)$ or $\bar{\mathcal{M}}(s)$ will halt and accept; therefore, the constructed machine will either accept s if $s \in L$ or reject s if $s \in \bar{L}$. Therefore it *decides* L, which is thus recursive.
  Therefore, closure wrt complementation only works for RE languages that are also recursive (and we know that there are RE languages that aren't recursive).

Observations

- The only case in which we really need to alternate the execution of the two TMs is the union of RE languages, because either machine could accept. However, alternating the execution of machines also works in the other cases, therefore it isn't an error.
- For the RE case 1.2, it is possible to provide proofs using TMs that enumerate the elements of the language, in place of TMs that recognize them.
- Of course $L_1 \cup L_2 \subseteq L_1$, but even if we know that L_1 is recursive, this does not mean that all of its subsets are. For example, Σ^* is recursive, but HALT $\subseteq \Sigma^*$ isn't.

$\boxed{\textbf{Exercise 2}}$

The FACTORING language is the decision version of the prime factoring problem. It contains all integer pairs (N, k) such that N has a proper divisor not greater than k:

$$\text{FACTORING} = \{(N, k) \in \mathbb{N}^2 \mid \exists m : 1 < m \leq k \wedge N \bmod m = 0\}.$$

In other words, N is divisible by some number greater than 1 but not larger than k. For instance:

- $(36, 3) \in$ FACTORING, because $N = 36$ is divisible by $m = 2$ (which is smaller than $k = 3$);
- $(35, 4) \notin$ FACTORING, because the smallest proper divisor of $N = 35$ is 5, which is larger than $k = 4$ (i.e, no number m between 2 and $k = 4$ divides $N = 35$).

2.1) Write an algorithm (in any language or pseudocode you like) that decides FACTORING. Assuming that integers are encoded with a positional system (e.g. binary), show that your algorithm runs in exponential time with respect to the input size.

2.2) Prove that FACTORING $\in$ **NP**.

2.3) Prove that FACTORING $\in$ **coNP**.

Hint — *For point 2.1, just iterate over all values of $m = 2, \ldots, k$; when you discuss the algorithm's complexity, remember that N and k are the input values, but the input size is much less.*

If, against all odds, you find a polynomial-time algorithm, you should publish your answer on a major Math or CS journal.

$\boxed{\textbf{Solution 2}}$

2.1) Consider the following algorithm:

```
• On input $(N, k)$:
    – for $m = 2, \ldots, k$:
        ∗ if $N \bmod m = 0$ then accept and halt
    – Reject and halt
```

This algorithm clearly decides FACTORING by iterating through all possible divisors of N from 2 to k, and rejects if none is found. While we can assume that arithmetic operations take polynomial time wrt the operand sizes, the whole algorithm is contained in a loop that, on the worst case, requires $O(k)$ iterations.

Remember, however, that the *size* of the input is given by the number of digits needed to represent the input. Let $|k|$ be the size of k: then $|k| = O(\log k)$; therefore, the number of iterations is $O(2^{|k|})$, which is *exponential* wrt the size of the input. The input also includes N, but it has logarithmic size too; moreover, we can assume k to be of the same order of magnitude as $\sqrt{N}$ in the worst realistic case.

Even though we can envision many optimizations, nobody has been able to provide a polynomial-time decision algorithm.

2.2) To prove $(N, k) \in$ FACTORING, a suitable certificate is the divisor m that we are looking for. Its size is clearly polynomial (since $m \leq k$), and we need to check that it satisfies all requirements: $2 < m \leq k$ and $N \bmod m = 0$; both operations can be performed in polynomial time.

The existence of the polynomial certificate meand FACTORING $\in$ **NP**.

2.3) To prove that $(N, k) \notin$ FACTORING, we must prove that no suitable m exists. The certificate is the prime factorization of N, consisting of l prime numbers $p_1, \dots, p_l$, possibly repeated. Each such number is clearly less than N, and there is at most a logarithmic number of prime factors; therefore, the size of the certificate is polynomial wrt the size of N. To check the certificate, we need to verify the following:

- that all p_i's are prime (a polytime check exists),
- that their product is indeed N (multiplication is polytime),
- finally, that all p_i's are larger than k: this ensures that no proper divisor m of N can be found in $\{2, \dots, k\}$.

The existence of a polynomial certificate for a negative answer means FACTORING $\in$ **coNP**.

Observations

- In point 2.1, we don't need to check for the primality of m. In fact, if m were composite, it would just mean that N has divisors that are even smaller than m, thus reinforcing the condition. Anyway, checking for primality hasn't considered an error.
- For the same reason stated above, in point 2.2 we don't need to check for primality of the certificate. Even if m were composite, N would have even smaller prime divisors.
- Since the primality of m is not relevant, what if in point 2.3 we used as a certificate the list of *all* divisors of N, prime or not? We could still verify that they are all larger than k. However, we wouldn't be able to verify in polynomial time that the list is complete unless we single out the prime divisors, which are therefore the only useful ones.
  Hence, for point 2.3 we *need* to test primality.
- The existence of Shor's algorithm doesn't mean FACTORING $\in$ **P**. Shor's algorithm is a quantum algorithm, which requires exponential time to be simulated by a classical computer such as a Turing machine, and it only implies FACTORING $\in$ **BQP**. It dosn't even (directly) imply that FACTORING $\in$ **NP**, because it is possible that **BQP** $\not\subseteq$ **NP**.

Exercise 3

Let $G = (V, E)$ be a directed graph (meaning that its edges are *ordered* pairs: $E \subseteq V \times V$).
A *cycle* in G is a sequence of $k \geq 2$ vertices $v_1, \dots, v_k \in V$ such that consecutive vertices are connected by an edge in the correct direction:

$$\forall i = 1, \dots, k-1 \quad (v_i, v_{i+1}) \in E,$$

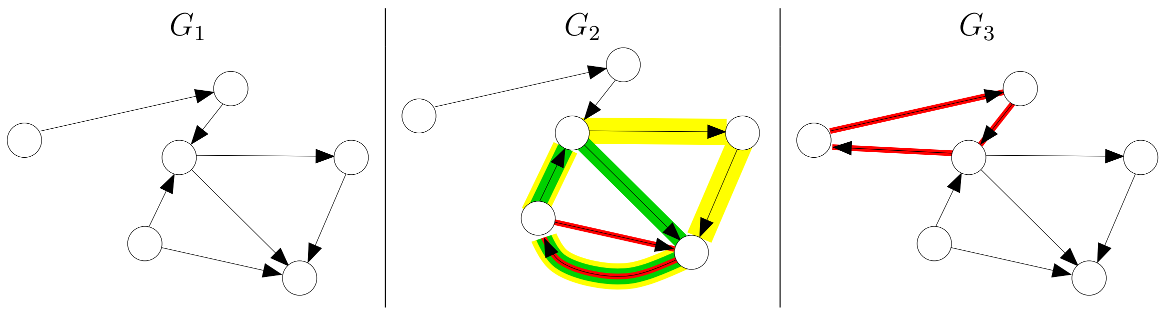
and the last vertex is connected to the first:

$$(v_k, v_1) \in E.$$

We define the language

$$\text{HAS\_CYCLES} = \{G | G \text{ is a directed graph and contains at least one cycle}\}.$$

For example, consider the three directed graphs:



Then:

- $G_1 \notin$ HAS_CYCLES (it doesn't contain cycles);
- $G_2, G_3 \in$ HAS_CYCLES because both contain at least a cycle (highlighted).

3.1) Prove that HAS_CYCLES $\in$ **P**.
3.2) Prove that HAS_CYCLES $\in$ **NL**.
Hint — *While some (pseudo-) code would be ideal, a word-only high-level description of the decision algorithms is fine.*

Solution 3

3.1) Since we know that **NL** $\subseteq$ **P**, the proof of point 3.2 would imply HAS_CYCLES $\in$ **P**. However, let us also see a possible algorithm; if we don't care much about efficiency (but still wand everything to be polynomial), we can iterate through all nodes and, for each of them, follow all paths and se if we meet the same node again.

```
• for $s \in V$:
    – queue $\leftarrow [s]$
    – visited $\leftarrow \{\}$
    – step $\leftarrow 0$
    – repeat as long as queue is not empty:
        * $v \leftarrow$ queue.extract()
        * for $v'$ such that $(v, v') \in E$ and $v' \notin$ visited:
            · if $v' = s$: accept and halt
            · queue.insert($v'$)
            · visited $\leftarrow$ visited $\cup \{v\}$
• Reject
```

The algorithm iterates all nodes s and visits all nodes reachable from s. If it meets s again during the search, then the graph has a cycle and the algorithm halts. If the iteration ends without ever repeating a node, the last line rejects.
Observe that the algorithm is composed of three nested loops, each running for at most $|V|$ iterations. Therefore, its time complexity is polynomial and HAS_CYCLES $\in$ **P**.
3.2) Consider the following algorithm, inspired by the non-deterministic algorithm for STCON:

```
• Non-deterministically choose $s \in V$.
• $v \leftarrow s$
• Repeat for at most $|V|$ iterations:
    – Non deterministically choose $v'$ such that $(v, v') \in E$.
    – If $v' = s$ accept and halt.
    – $v \leftarrow v'$.
• Reject.
```

The "lucky" branch of the computation chooses the initial node s in the cycle (if there is one) and follows the right path leading to s itself, ending with acceptance.
If there are no cycles, then no branch is lucky and they all reject.
Observe that the algorithm uses a fixed number of variables, each referring to a node in the graph, and a counter All such variables require $O(\log |V|)$ space. Therefore, the whole non-deterministic algorithm requires $O(\log |V|)$ space too.

Observations

- For point 3.1, a much shorter answer would be fully acceptable, as long as the method is clearly polytime. For example:
  "Choose a node as starting point of a breadth-first (or depth-first) visit, keeping track of visited nodes; if a node is visited twice, then accept and halt. If at the end all nodes have been visited, reject and halt. Otherwise, repeat the procedure starting from one of the non-visited nodes."
- Another unrelated solution for point 3.1:

```
– Remove all self-loops (edges from a node to itself)
– as long as $G$ is not empty and there are nodes with no outgoing edges ("sinks"):
    * remove all sinks from the graph;
– if $G$ is not empty then accept, else reject
```

At the end of the ”pruning” loop, all surviving nodes have an outgoing edge leading to another node; by following a trail of outgoing edges, sooner or later we are bound to revisit some node, hence having a loop.

- For point 3.2, the initial non-deterministic choice of the starting node s couls be replaced by a deterministic loop. Only time complexity would change, but space would remain the same (and the algorithm might gain some clarity).
- Again for point 3.2, we could also think of a reduction to the **NL** algorithm for STCON by asking if $(G, v, v) \in$ STCON for some v.
  However, we need to consider that STCON always accepts paths from a node to itself because one-node (singleton) paths are allowed, while we are looking for a non-singleton path from a node to itself. Thus we need a modified version of the algorithm that only accepts after having made at list one iteration.
  Or, even better, we iterate on all arcs (s, t) and check if we can cycle back from t to s by asking $(G, t, s) \in$ STCON. For the same reasons stated above, the algorithm would still be **NL**.