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

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

Tuesday, June 23, 2026, revised on June 24, 2026

Exercise 1

Consider the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and the language

$$L = \{\mathtt{a}^n\mathtt{bba}^n | n \in \mathbb{N}\} \subseteq \Sigma^*$$

i.e., a string belongs to L if it contains zero or more a's, followed by two b's and then the same number of a's, for example:

$$\mathtt{bb} \in L, \quad \mathtt{aabbaa} \in L, \quad \mathtt{aaabbaa} \notin L, \quad \mathtt{aaabaaa} \notin L.$$

1.1) Specify a one-tape, three-symbol deterministic Turing machine (on alphabet $\Sigma \cup \{\textvisiblespace\}$) that decides L.
What is the time complexity of your solution with respect to the input size (logarithmic, linear, quadratic, cubic, exponential...), and why?

1.2) Show that $L \in \mathbf{L}$.

1.3) Show that a two-tape DTM could decide L in linear time.

Hint — *1.1: A DTM can be specified as a state-transition diagram or as a transition table indexed by symbols and states. If you cannot do it (or you have no time), then a verbal description is fine as long as it describes elementary actions of a TM (e.g., "count the number of a's" isn't acceptable unless you specify how to do it). Remember that a one-tape DTM is allowed to modify its input (e.g., delete a's as long as they are taken into consideration).*
1.2 and 1.3: here a high-level description is fine, as long as it allows us to agree on the conclusion.

Solution 1

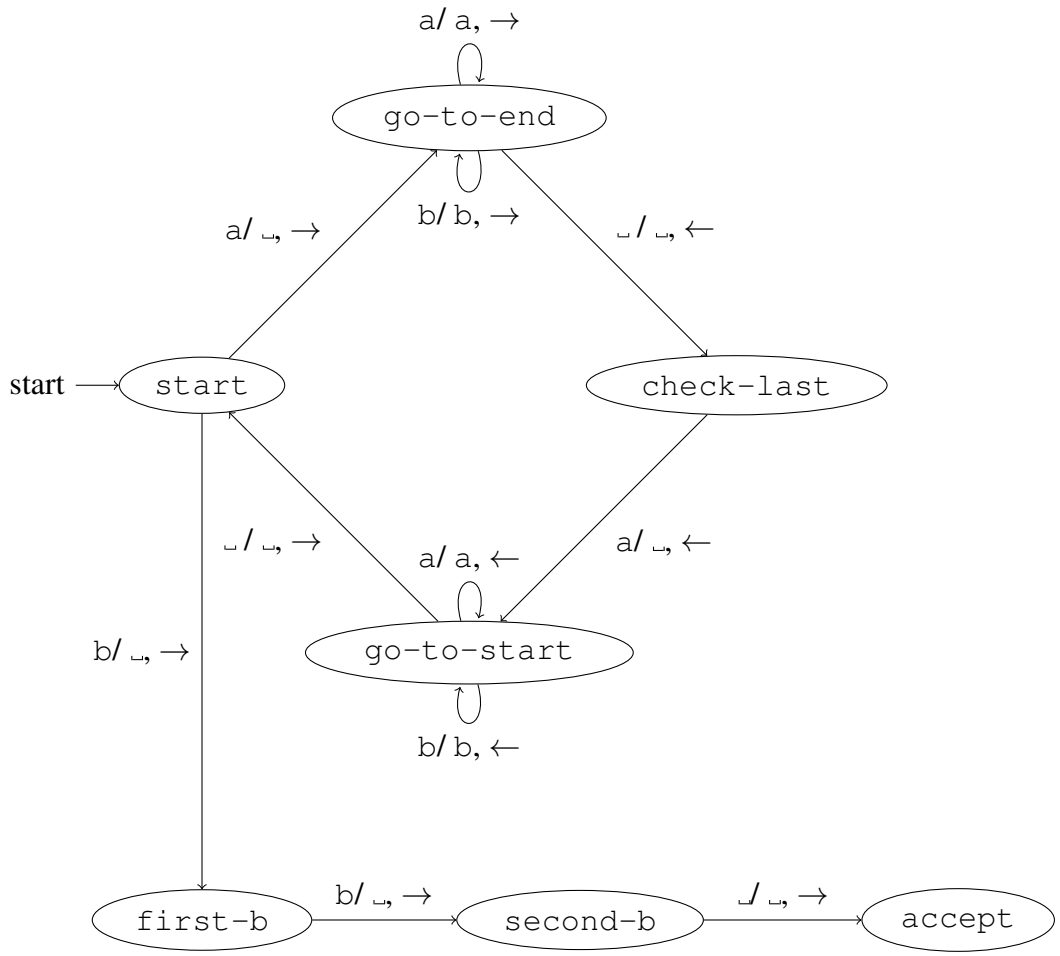
1.1) Let us make a few assumptions:

- the initial position is at the leftmost character of the input string;
- the input string never includes blanks inside (i.e., if we meet a blank, we know that there is no more input on that side);
- the machine has two halting states, one for accepting and one for rejecting the string.

At this point, we can simply perform a series of scans that keep removing the leftmost and rightmost a's until we run out of them[1]; finally we count what remains, checking that there are exactly two b's and nothing else. The following diagram shows the machine; note that the rejecting state is not represented; we just assume that all transitions that are not represented in the diagram go to the rejection state.

[1] Sort of what we did to check that a string is a palyndrome, but without having to remember the last read symbol, because it must be an a.

The top lozenge keeps removing the leftmost and the righmost a; as soon as there are no more a's on the left, we check that the remaining symbols are exactly two b's (bottom states: we count the b's using two separate states, first-b and second-b):



You can analyze and test the machine at

`https://morphett.info/turing/turing.html?55adb5d8ae8a19e03460e0cc08fd952b`

About the complexity, observe that the first part of the machine (the lozenge states) repeats a $O(n)$ scan of the input string, every time reducing the string size of a constant ($O(1)$) size. Therefore, it repeats $O(n)$ times a $O(n)$ scan, therefore it runs in $O(n^2)$ time: it is quadratic with respect to the input size.
With a 1-tape TM we cannot do better, since we need to compare things that are at opposite ends of the string.

1.2) Remember that space complexity classes are defined in terms of a two-tape DTM, where the input is written in a read-only tape. The following machine scans the input once and maintains a single counter on the read-write tape:

- initialize the counter on the read/write tape to zero
- while the input is a
  - increase the counter on the read/write tape
  - move right

- if the input is `b`
  - move right
- if the input is `b`
  - move right
- while the input is `a`
  - decrease the counter on the read/write tape
  - move right
- if the input is ␣ and the counter on the read/write tape is zero
  - accept

In all "else" cases, not shown for simplicity, the machine must reject the input.
The machine counts all `a`'s on the left-hand side of the string, then it requires to pass exactly two `b`'s and then decreases the same counter for every scanned `a`. Every anomalous symbol leads to immediate rejection. Since the counter is incremented at most once per scanned symbol, its size never exceeds the logarithm of the string size, therefore it requires $O(\log n)$ space.

1.3) One could observe that the time complexity of the algorithm 1.2 is (at most) $O(n \log n)$ because at each of the $O(n)$ scanned symbols at most one increase or decrease operation are needed (requiring at most $O(\log n)$ steps). Even better, increasing and decreasing the counter actually requires amortized constant time, so the overall complexity is actually $O(n)$.
A simpler way to have a linear TM is to do something similar to the above but, instead of maintaining a counter, just copy the initial string of `a`'s from the first to the second tape; once the two central `b`'s are found, we proceed to remove one `a`from the second tape for every `a`scanned on the first. If we end up with an empty second tape then we accept; if anything doesn't work according to plan, then we reject. See the following two-tape machine:

`http://turingmachinesimulator.com/shared/ccwnazcxwg`

Since the input is just scanned once, always left to right, and the machine halts at the end of the input, then the tiem complexity of the algorithm is $O(n)$.

Exercise 2

Let DOUBLE-SAT be the language of CNF Boolean formulas with *at least two* satisfying assignments.

2.1) Show that DOUBLE-SAT $\in$ **NP**.

2.2) Show that DOUBLE-SAT is **NP**-complete.

Hint — *For 2.2, try to reduce SAT to DOUBLE-SAT. Given CNF formula $f(x_1, \ldots, x_n)$, add a dummy variable y to it obtaining $f'(x_1, \ldots, x_n, y)$ such that f' is satisfiable if and only if f is too, but y can be either true or false, so that f' has twice the number of solutions as f.*

Solution 2

2.1) Given a CNF formula $f(x_1, \ldots, x_n)$, to prove that it belongs to DOUBLE-SAT we just need to provide two different assignments $y_1, \ldots, y_n$ and $z_1, \ldots, z_n$ such that both $f(y_1, \ldots, y_n)$ and $f(z_1, \ldots, z_n)$ are true. The two assignments have polynomial size wrt the formula (just one bit per variable, therefore they are much shorter) and can be checked by evaluating f on them.
Basically, a certificate for DOUBLE-SAT consists of two different certificates for SAT that can be checked independently (after checking that they are different).

2.2) Since we already know that SAT is **NP**-complete, we just need to prove that SAT $\leq_{\mathbf{P}}$ DOUBLE-SAT.
Take any CNF $f(x_1, \ldots, f_n)$, and let's follow the suggestion of adding a dummy variable y. Since we want that y doesn't influence the satisfiability of the formula, we can add the clause $(y \vee \neg y)$, which is always true. Therefore we define the new formula

$$f'(x_1, \ldots, x_n, y) = f(x_1, \ldots, x_n) \wedge (y \vee \neg y).$$

Observe that if f is not satisfiable (i.e., $f \notin$ SAT), then neither is f', because it contains the same clauses and one more.
On the other hand, if f is satisfiable (i.e., $f \in$ SAT), then f' is too, because the additional clause is always true; moreover, for every satisfying assignment $(x_1, \ldots, x_n)$ to f there are two satisfying assignment to f', namely $(x_1, \ldots, x_n, y = \text{true})$ and $(x_1, \ldots, x_n, y = \text{false})$, hence

$$f \in \text{SAT} \Leftrightarrow f' \in \text{DOUBLE-SAT}.$$

The reduction from f to f' is polynomial on the size of f because it consists of adding a constant-size clause.
An alternative definition for f' is by doubling all clauses. If

$$f(x_1, \ldots, x_n) = C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

then we can define f' as follows:

$$f'(x_1, \ldots, x_n, y) = (C_1 \vee y) \wedge (C_1 \vee \neg y) \wedge (C_2 \vee y) \wedge (C_2 \vee \neg y) \wedge \cdots \wedge (C_m \vee y) \wedge (C_m \vee \neg y);$$

again, we have twice the assignments of f.

Observations

Another definition yet was suggested by some students:

$$f'(x_1, \ldots, x_n, y) = (C_1 \vee y) \wedge (C_2 \vee y) \wedge \cdots \wedge (C_m \vee y),$$

adding "$\vee y$" to all clauses, and claiming that if $f \notin$ SAT then f' only has one solution ($y =$ true), and therefore $f' \notin$ DOUBLE-SAT.
Actually, f has 2^n solution, because as soon as y is set to true every assignment to the other variables is fine, therefore $f' \in$ DOUBLE-SAT independent of the satisfiability of f, therefore the reduction doesn't work.

Exercise 3

Let's consider a universal Turing machine $\mathcal{U}$ able to simulate 1-tape TMs on alphabet $\{0, 1, \sqcup\}$. Given a string $s \in \{0,1\}^*$, remember that its *Kolmogorov complexity* $K_{\mathcal{U}}(s)$ is the size of the smallest Turing machine (in $\mathcal{U}$'s notation) that, when started on an empty tape, outputs s and halts.

Show that the set

$$\left\{s \in \{0,1\}^* | K_{\mathcal{U}}(s) \leq 10^6\right\}$$

(binary strings having Kolmogorov complexity not larger than 10^6) is recursively enumerable.

Solution 3

The required set is the set of all the outputs of all machines of size not larger than 10^6 that halt when executed on an initially empty tape.

First of all, since the set is obviously finite (the number of considered machines is finite, and each outputs at most one string), then it is implicitly recursive, in the sense that there must be a machine that decides it.

To prove that the set is RE in a constructive way, let us use the "procedural" definition of RE: a set is RE if there is a TM that prints all and only its elements.

To achieve this, we "just" need to simulate all machines of size not larger than 10^6 "in parallel", i.e. by alternating a step of each machine. Every time a simulated machine halts, we take its tape and write it to some dedicated output tape. We cannot simulate them one after the other because some of them is never going to halt.

If a string $s \in \{0,1\}^*$ is such that $K_{\mathcal{U}}(s) \leq 10^6$, it means that one of the TMs that we are simulating will produce it as its output before halting, therefore it will eventually be written.

If, on the other hand, $K_{\mathcal{U}}(s) > 10^6$, then none of the simulated machines will ever produce s, and it will never be written out.