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

Solutions to Homework 6

This homework is due 24 hours later than usual, on *Thursday* 2/22 at 8pm. This is because some of the questions (labeled below) rely on the lecture on 2/19.

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" on Canvas 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 is to be in your own words.

Solution:

2. **A powerful diagonalization.**

(7 pts) (a) The *power set* of a set A, denoted $\mathcal{P}(A)$, is defined as the *set of all subsets* of A. For example, $\mathcal{P}(\{0,1\}) = \{\emptyset, \{0\}, \{1\}, \{0,1\}\}$.

Use diagonalization to prove that for any *countably infinite* set A, the power set $\mathcal{P}(A)$ is *uncountably* infinite.

Hint: View each subset as an infinite binary sequence representing whether each element of A is included in the subset or not.

Solution: Let A be a countably infinite set. By definition, there is an enumeration of its elements, as $A = \{a_0, a_1, a_2, a_3, \ldots\}$.

Now suppose for the sake of contradiction that $\mathcal{P}(A)$ is *countably* infinite. (Note that $\mathcal{P}(A)$ is infinite because there are infinitely many 'singleton' subsets $\{a_i\} \in \mathcal{P}(A)$, since A is infinite.) Then by definition, there is an enumeration of all the subsets of A, as

$$A_0, \quad A_1, \quad A_2, \quad A_3, \quad \ldots$$

where each $A_i \subseteq A$. Define the "flipped diagonal" set $D = \{a_i : a_i \notin A_i \text{ for } i \geq 0\}$. Notice that $D \subseteq A$, because all its elements are in A. However, by construction, $D \neq A_i$ for every $i \geq 0$ because $a_i \in D$ if and only if $a_i \notin A_i$. Therefore, D does *not* appear in the above enumeration of $\mathcal{P}(A)$, which contradicts our assumption that $\mathcal{P}(A)$ is countable. Therefore, $\mathcal{P}(A)$ is uncountably infinite.

As an alternative (but closely related) approach, note that each subset $A' \subseteq A$ can be equivalently represented by an infinite binary sequence s, whose ith digit is

$$s_i = \begin{cases} 0 & \text{if } a_i \notin A' \\ 1 & \text{if } a_i \in A' \end{cases}.$$

That is, each subset corresponds to its own infinite binary sequence, and vice versa. In discussion, we proved by diagonalization that there are *uncountably many* infinite binary sequences, i.e., the *set* of infinite binary sequences is uncountable. Since the subsets of A are in exact (bijective) correspondence with the infinite binary sequences, we conclude that there are uncountably many subsets of A.

(6 pts) (b) Prove that any infinite language $L \subseteq \Sigma^*$ has an undecidable subset $L' \subseteq L$.
Hint: Part (a) will be useful.

Solution: We first show that L is *countably* infinite, which we will need in order to use part (a). This is because $L \subseteq \Sigma^*$: since there is an enumeration of Σ^* (in which every string $x \in \Sigma^*$ appears), we can modify it to get an enumeration of L by just removing the strings that are not in L. (Alternatively: the identity function from L to Σ^* is injective, and there is an injective function from Σ^* to $\mathbb{N}$ because Σ^* is countable, so composing these functions yields an injective function from L to $\mathbb{N}$, which shows that L is countable.)

Now suppose for the sake of contradiction that every $L' \in \mathcal{P}(L)$ (i.e., every $L' \subseteq L$) is decidable. This means that each $L' \in \mathcal{P}(L)$ is decided by some *distinct* Turing machine $M_{L'}$, because a particular Turing machine can decide at most one language. (It's even true that L' is decided by *many* different Turing machines, but for this argument it suffices to take just one.)

We have seen in class that the set of all Turing machines is countable, i.e., they can be enumerated as $M_0, M_1, M_2, \ldots$. We can 'filter' this list to include only those machines $M_{L'}$ identified above, removing all others. Then because every $L' \in \mathcal{P}(L)$ is

represented in the filtered list by the distinct machine $M_{L'}$ that decides it, it follows that $\mathcal{P}(L)$ is countable. However, this contradicts the result from part (a), which says that $\mathcal{P}(L)$ is uncountable (here is where we use the fact that L is countably infinite). So our initial assumption was false, i.e., some $L' \in \mathcal{P}(L)$ is undecidable, as claimed.

(Alternatively, we can reach the contradictory conclusion that $\mathcal{P}(L)$ is countable via injective functions. Because a Turing machine can decide at most one language, the function from $\mathcal{P}(L)$ to the set of Turing machines that maps L' to $M_{L'}$ is injective. Moreover, there is an injective function from the (countable) set of Turing machines to $\mathbb{N}$. So, composing these two functions yields an injective function from $\mathcal{P}(L)$ to $\mathbb{N}$, as needed.)

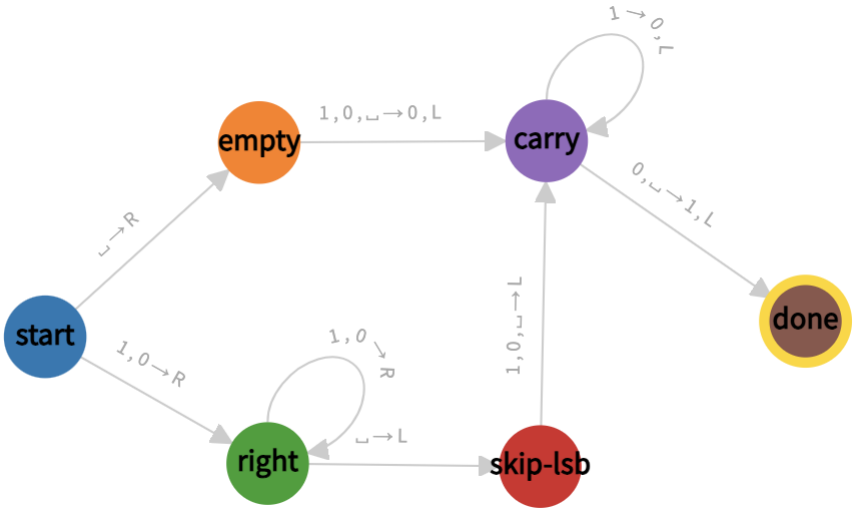
Alternatively, we can prove the claim directly via diagonalization, similarly to the proofs we have seen in class, discussion, or part (a). Since we proved above that L is countably infinite, we can enumerate it as $L = \{x_0, x_1, x_2, \ldots\}$, and we have previously seen that we can enumerate all Turing machines as $M_0, M_1, M_2, \ldots$. Then, we consider the "flipped diagonal" language $L' = \{x_i \in L : x_i \notin L(M_i) \text{ for } i \geq 0\}$, so $L' \subseteq L$ by definition. By construction, $L' \neq L(M_i)$ for every $i \geq 0$ because $x_i \in L'$ if and only if M_i does *not* accept x_i. Thus, L' is not decided by (or even recognized by!) any Turing machine, because our enumeration has *every* Turing machine in it. So, we have shown that there is an undecidable subset $L' \subseteq L$.

3. **Turing machines.**

(7 pts) (a) Go to https://turingmachine.io/ and step through the example code that adds 1 to an input integer (represented as a binary string). Modify this machine to add 2 instead of 1 (scroll down on the page for instructions on how to write the code). Give a picture or screenshot of your new machine as your solution. You do not need to provide justification.

Solution: Note that the computational task here is not a decision problem, so the TMs do not have accept or reject states, just a "done" state to indicate that execution has finished. The output of computation is the contents of the tape (ignoring the infinite sequences of blanks on either side; notice that the TM model on the website considers a two-way infinite tape).

The key modification is to skip the carry state for the least-significant bit (furthest right). To do this, we add a state that moves the head left without modifying the LSB after reaching it. We also need a parallel path to handle the empty-string input (which means zero). In the following state-transition diagram, the transition rules are written in a more concise form: if the transition writes the same symbol that it reads, the written symbol is omitted from the rule.



Here is the code that created the solution machine:

```
blank: ' '
start state: start
table:
  # scan to the second rightmost digit
  start:
    [1,0]: {R: right}
    ' ' : {R: empty}
  empty:
    [1,0,' '] : {write: 0, L: carry}
  right:
    [1,0]: R
    ' ' : {L: skip-lsb}
  # left once
  skip-lsb:
    [1,0,' '] : {L: carry}
  # then carry the 1
  carry:
    1       : {write: 0, L}
    [0,' ']: {write: 1, L: done}
  done:
```

(8 pts) (b) Let L be the language of all binary strings that represent non-negative integers that are divisible by 4:

$$L = \{\langle s \rangle : s \bmod 4 = 0\} \cup \{\varepsilon\}.$$

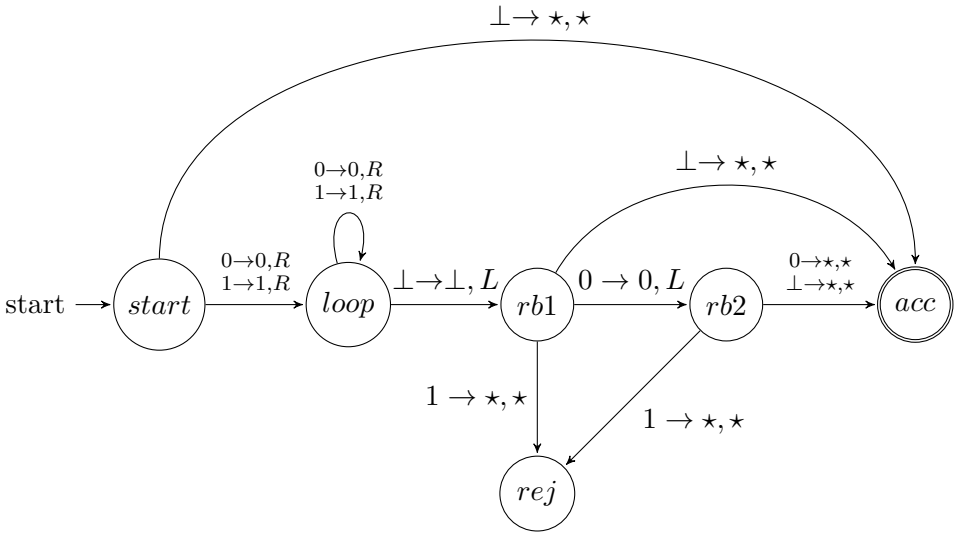
Note that leading zeroes in the representation are acceptable, i.e., 000101010 and 101010 both represent 42. Also, we define ε to represent 0.

Draw a state-machine diagram for a Turing machine that decides L. This drawing can

be done however you like (hand-drawn, using https://turingmachine.io/, drawn in latex, etc.), as long as it is readable. No justification is necessary.

Hint: How is the last bit of the binary representation of an integer related to whether the integer is divisible by 2? Can you extend this observation to divisibility by 4?

Solution: The key idea is to scan all the way to the right end of the input, and check that the last two bits are 0. (If there are fewer than two bits in the input, we check that all of them are 0.) Stars ($\star$) are used to indicate values that don't matter and can be chosen arbitrarily.



4. **Decidability.**

(7 pts) (a) Prove or disprove: any finite language is decidable.

Solution: Let $L = \{\ell_1, \ldots, \ell_n\}$ be any finite language. We can write a decider M for L as follows. $M(x)$:

1. If $x = \ell_1$ or $x = \ell_2$ or … or $x = \ell_n$, accept.
2. Otherwise, reject.

We claim that this pseudocode can be implemented as a Turing machine, where each comparison of x against some $\ell_i \in L$ is performed using a finite number of states, and thus there are a finite number of states overall. More specifically, for each $\ell_i \in L$ we have a finite sequence of states that compare each cell of the tape to the corresponding character of ℓ_i, and finally compare the next cell to the blank symbol (to check that the end of the input string has been reached). This can be done with one state per character to be compared, where a match transitions to the next state (or to the accept state if all characters match), and a non-match transitions to a state that moves the head back to the start of the input, then transitions to the next comparison (or to the

reject state if there are no more strings to be compared against). Essentially, all this "hard-codes" the strings of L into the states and transition function of the machine.

That M is a decider for L is clear by inspection: if $x \in L$ then $x = \ell_i$ for some $\ell_i \in L$, so one of the comparisons in the first line will cause the machine to accept. If $x \notin L$, then none of the comparisons in the first line will cause the machine to accept, and hence it will reject on the next line.

It is important to realize that the above strategy would *not* work if L was an *infinite* language. (Indeed, if it did, then every language would be decidable!) One reason is that the "code" (states and transitions) for a Turing machine needs to be finite, so we can't hard-code infinitely many strings into it. Furthermore, even if there was a way for a Turing machine to *generate* the elements of L in some sequence (and it turns out that this is possible for any *recognizable* language), we can't check x against an infinitely long sequence of strings in finite time. Specifically, if $x \notin L$, then the machine would not halt because it would never stop comparing x against the sequence of strings in L.

(5 pts) (b) Let $L = (L_1 \setminus L_2) \cap L_3$, where L_1, L_2, and L_3 are *decidable* languages. State whether L is decidable for **all**, **some** (but not all), or **no** such L_1, L_2, L_3, and prove it.

Note: $A \setminus B := \{x \in A : x \notin B\}$ denotes the *set difference* between A and B.

Solution: L is decidable for **all** such choices of L_1, L_2, and L_3.

Let M_1 be a TM that decides L_1, M_2 be a TM that decides L_2, and M_3 be a TM that decides L_3, which all exist by hypothesis. We construct a TM M that decides $(L_1 \setminus L_2) \cap L_3$ as follows:

M = on input $\langle x \rangle$:
1: Run M_1 on input $\langle x \rangle$
2: Run M_2 on input $\langle x \rangle$
3: Run M_3 on input $\langle x \rangle$
4: **if** M_1 accepted, M_2 rejected, and M_3 accepted **then** accept
5: **else** reject

We now show that M decides $(L_1 \setminus L_2) \cap L_3$:

Observe that $x \in (L_1 \setminus L_2) \cap L_3$
$\implies x \in (L_1 \setminus L_2)$ and $x \in L_3$
$\implies x \in L_1$ and $x \notin L_2$ and $x \in L_3$
$\implies M_1$ accepts x, and M_2 rejects x, and M_3 accepts x
$\implies M$ accepts x .

Also, $x \notin (L_1 \setminus L_2) \cap L_3$
$\implies x \notin (L_1 \setminus L_2)$ or $x \notin L_3$
$\implies x \notin L_1$ or $x \in L_2$ or $x \in L_3$
$\implies M_1$ rejects x, or M_2 accepts x, or M_3 rejects x
$\implies M$ rejects x .

Hence M decides L, so L is indeed decidable.

Here is alternative solution that uses previously proved results in a modular way. We proved in discussion that the set difference between any two decidable languages is also decidable. So, $L_1 \setminus L_2$ is decidable. And we proved in lecture that the intersection of any two decidable languages is also decidable. So, $L = (L_1 \setminus L_2) \cap L_3$ is decidable. (Alternatively, we could observe that $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$ is the intersection of two decidable languages, since we proved that the complement of any decidable language is decidable.)

(7 pts) (c) Let $L = L_1 \cap L_2$, where L_1 and L_2 are both *undecidable*. State whether L decidable for **all**, **some** (but not all), or **no** such L_1, L_2, and prove it.

Solution: L decidable for **some** (but not all) such choices of L_1, L_2.
Consider the case where $L_1 = L_2$. In this case $L = L_1 \cap L_2 = L_1$, which is undecidable by hypothesis.
Now let L_1 be some undecidable language and let $L_2 = \overline{L_1}$ be its complement. We showed in lecture that if a language L is decidable then its complement is also decidable. By contrapositive, if a language is undecidable, then its complement is also undecidable. So, both L_1 and L_2 are undecidable. Then $L_1 \cap L_2 = \emptyset$, which is decidable (by a decider which just ignores its input and rejects).

5. **Haltchecker.** (Warning: This problem is related to material from the lecture on 2/19.)

(8 pts) (a) Recall from lecture the definition $L_{\text{HALT}} := \{(\langle M \rangle, x) : M \text{ halts on input } x\}$. In class you saw a Turing reduction from L_{ACC} to L_{HALT}, proving that L_{HALT} is undecidable.
This linked video describes a different argument that L_{HALT} is undecidable, without using a reduction.
Formalize the argument in the video as a rigorous proof.

Solution: Note: This was the first existence proof for an undecidable language, and it is due to Alan Turing.
Suppose for the sake of contradiction that some Turing machine H decides L_{HALT}. Using H, we design a Turing machine X such that $X(\langle M \rangle)$ halts if and only if $H(\langle M \rangle, \langle M \rangle)$ rejects, for any Turing machine M.
More precisely, the Turing machine X is defined as follows:

$X(\langle M \rangle)$:
1: **if** $H(\langle M \rangle, \langle M \rangle)$ accepts **then loop**
2: **else accept**

(Note: Although our definition of X does not explicitly include the simple machines P and N defined in the video, it implicitly reflects their behavior.)

We prove the above claim about the behavior of $X(\langle M \rangle)$. First observe that its call to $H(\langle M \rangle, \langle M \rangle)$ halts, because H is a decider. If H accepts, then X loops, by definition. If H rejects, then X accepts (and hence halts), again by definition. Thus, $X(\langle M \rangle)$ halts exactly when $H(\langle M \rangle, \langle M \rangle)$ rejects, and loops otherwise.

Now we consider the behavior of $X(\langle X \rangle)$. Because X is a well defined Turing machine, there are only two possibilities to consider: that $X(\langle X \rangle)$ halts, and that it does not halt (it loops).

Case 1: If $X(\langle X \rangle)$ *halts*, then by the above claim, it must be that $H(\langle X \rangle, \langle X \rangle)$ rejects. Since H is a decider for L_{HALT}, this means that $X(\langle X \rangle)$ *does* ***not*** *halt*, which contradicts the hypothesis for this case.

Case 2: If $X(\langle X \rangle)$ *does* ***not*** *halt*, then by the above claim, it must be that $H(\langle X \rangle, \langle X \rangle)$ accepts. Since H is a decider for L_{HALT}, this means that $X(\langle X \rangle)$ *halts*, which again contradicts the hypothesis for this case.

In either case we get a contradiction, so we conclude that X cannot exist, hence neither can H. That is, L_{HALT} is undecidable.

(7 pts) (b) Consider the following language:

$$L_{\text{HALT}<2^{|x|}} = \{(\langle M \rangle, x) : M(x) \text{ halts within } 2^{|x|} \text{ steps}\} .$$

Prove or disprove: $L_{\text{HALT}<2^{|x|}}$ is decidable.

Solution: $L_{\text{HALT}<2^{|x|}}$ is decidable. An algorithm for deciding $L_{\text{HALT}<2^{|x|}}$ is as follows: given $\langle M, x \rangle$, simulate running $M(x)$ for up to $2^{|x|}$ steps. Accept if $M(x)$ halts within that number of steps; otherwise, reject.

This algorithm decides $L_{\text{HALT}<2^{|x|}}$ because it halts on any input $(\langle M \rangle, x)$: it halts after running the simulation for up to $2^{|x|}$ steps, which is finite because $|x|$ is finite (since x is a string). By construction, it accepts $(\langle M \rangle, x)$ if and only if $M(x)$ halts within $2^{|x|}$ steps, i.e., if and only if $(\langle M \rangle, x) \in L_{\text{HALT}<2^{|x|}}$.

6. **Turing reductions.** (Warning: This problem relies on material from the lecture on 2/19.)

Before embarking on this problem, read Handout 2: Turing Reductions.

(7 pts) (a) Prove or disprove: if $L \leq_T L'$ for some language L', then L is decidable.

Solution: This statement is false. As a counterexample, we showed in lecture that $L_{\text{ACC}} \leq_T L_{\text{HALT}}$, and that L_{ACC} is undecidable. (There are many similar examples given in lecture and discussion.)

As an alternative counterexample, we could simply observe that every language is trivially Turing-reducible to itself, and that an undecidable language exists.

As a remark, it is true that if $L \leq_T L'$ for some *decidable* language L', then L is necessarily decidable.

(7 pts) (b) Prove or disprove: $L \leq_T L'$ for any decidable language L and any language L'.

Solution: This statement is true. Because L is decidable, there is a TM B that decides it. Given an oracle B' that decides L', a machine that decides L given access to B' is simply B itself!

The point is this: in a Turing reduction from L to L', even though an oracle that decides L' is available, the reduction *is not required to use it* (and does not do so, in this case).

(7 pts) (c) Define

$$L_{\text{AND-LOOP}} = \{(\langle M \rangle, x, y) : M \text{ is a TM that loops on input } x \text{ and loops on input } y\} .$$

Prove that $L_{\text{HALT}} \leq_T L_{\text{AND-LOOP}}$. Note that since L_{HALT} is undecidable, this means that $L_{\text{AND-LOOP}}$ is undecidable.

Solution: Given an oracle $M_{\text{AND-LOOP}}$ for $L_{\text{AND-LOOP}}$, construct a machine M_{HALT} that decides L_{HALT} as follows: $M_{\text{HALT}}(\langle M \rangle, x)$ simply calls $M_{\text{AND-LOOP}}(\langle M \rangle, x, x)$ and outputs the opposite answer.

Suppose $(\langle M \rangle, x) \in L_{\text{HALT}}$. Then $M(x)$ does not loop. So, $M_{\text{AND-LOOP}}(\langle M \rangle, x, x)$ rejects. Then because $M_{\text{HALT}}(\langle M \rangle, x)$ returns the opposite answer, it accepts, as needed.

Now suppose $(\langle M \rangle, x) \notin L_{\text{HALT}}$. Then $M(x)$ loops. So, $M_{\text{AND-LOOP}}(\langle M \rangle, x, x)$ accepts. Then because $M_{\text{HALT}}(\langle M \rangle, x)$ returns the opposite answer, it rejects, as needed.

(7 pts) (d) Let $L_{\text{SELF-LOOP}} = \{\langle M \rangle : M \text{ is a TM that loops on input } \langle M \rangle\}$. Prove that $L_{\text{SELF-LOOP}}$ is undecidable, via a Turing reduction from a problem that we proved undecidable in class.

Solution: We give a Turing reduction from L_{BARBER} to $L_{\text{SELF-LOOP}}$. Given an oracle $M_{\text{SELF-LOOP}}$ for $L_{\text{SELF-LOOP}}$, we construct a machine M_{BARBER} that decides L_{BARBER} as follows.

$M_{\text{BARBER}}(\langle M \rangle)$ runs $M_{\text{SELF-LOOP}}(\langle M \rangle)$; it accepts, then accept. Otherwise, run $M(\langle M \rangle)$ and output the opposite answer.

We now prove that M_{BARBER} is indeed a decider for L_{BARBER}.

Suppose $\langle M \rangle \in L_{\text{BARBER}}$, i.e., M does not accept $\langle M \rangle$. This means that $M(\langle M \rangle)$ either rejects or loops. If it loops, then $M_{\text{SELF-LOOP}}(\langle M \rangle)$ accepts, and so by our

definition of M_{BARBER}, we have that $M_{\text{BARBER}}(\langle M \rangle)$ accepts, as desired. On the other hand, if $M(\langle M \rangle)$ rejects, then because $M_{\text{BARBER}}(\langle M \rangle)$ runs this and returns the opposite answer, it accepts, again as desired.

Now suppose $\langle M \rangle \notin L_{\text{BARBER}}$, i.e., $M(\langle M \rangle)$ accepts. So, $M_{\text{SELF-LOOP}}(\langle M \rangle)$ rejects, so $M_{\text{BARBER}}(\langle M \rangle)$ proceeds to run $M(\langle M \rangle)$, which accepts, and then returns the opposite answer, which is to reject, as desired.

(5 EC pts) 7. **Optional extra credit: An interesting language.**

Recall that a language L is *recognizable* if there is a Turing machine that accepts every string in L, and either rejects *or loops* on every string not in L.

Prove that there exists a language L where neither L nor its complement $\overline{L}$ is recognizable.

Solution: There are several approaches to this problem. First, we describe a counting argument, and then we describe a diagonalization argument.

Recall that the language of a Turing machine M, denoted $L(M)$, is the set of strings that M accepts. Consider a function that maps each Turing machine M to the pair $(L(M), \overline{L(M)})$. The set of all languages that appear in some such tuple is countable, because the set of all TMs is countable: enumerating the TMs yields an enumeration of the languages, two at a time. Since the set of all languages is uncountable, there is a language L that does not appear in the enumeration. Since L does not appear as the first element of any of the pairs, L is not recognizable. And since L does not appear as the second element of any of the pairs, $\overline{L}$ is not recognizable.

Now we describe a diagonalization argument. Consider an enumeration $M_1, M_2, \ldots$ of all TMs, and an enumeration $w_1, w_2, \ldots$ of all strings. We exhibit a language L for which neither L nor $\overline{L}$ is recognizable. We include w_1 in L if M_1 *does not* accept w_1, and we include w_2 in L if M_1 *does* accept w_2. Similarly, we include w_3 in L if M_2 *does not* accept w_3, and include w_4 in L if M_2 *does* accept w_4. In general, for all $i \geq 1$, we include w_{2i-1} in L if M_i *does not* accept w_{2i-1}, and we include w_{2i} in L if M_i *does* accept w_{2i}.

Due to the design of how the w_i for *odd* i are included/excluded, L is not recognized by any machine in the list, and is thus not recognized by any TM. And due to the design of how the w_i for *even* i are included/excluded, $\overline{L}$ not recognized by any machine in the list, and is thus not recognized by any TM.