Problem Set 8

What lies beyond what can be solved by a computer? What intuitions can we build about those sorts of problems? In this problem set, you will learn how to reason about the unsolvable.

As always, please feel free to drop by office hours or send us emails if you have any questions. We'd be happy to help out.

This problem set has 125 possible points. It is weighted at 7% of your total grade.

Good luck, and have fun!

Due Monday, December 2nd at 2:15 PM

Problem One: Understanding Mapping Reductions (20 Points)

We often use mapping reductions to determine how hard one problem is by relating it to some other problem we already know about. However, we have to be careful when doing so.

i. Below is an **incorrect** proof that EQ_{TM} (the language of pairs of TMs with the same language) is **RE**, even though in lecture we proved that it was neither **RE** or co-**RE**. Identify what is wrong with this proof. You should be sure that you are 100% positive what is wrong with this proof before you attempt any other problems on this problem set, because the mistake made here is extremely common!

Theorem: $EQ_{TM} \in$ **RE**.

Proof: We will prove $A_{TM} \leq_M EQ_{TM}$; since $A_{TM} \in$ **RE**, this proves that $EQ_{TM} \in$ **RE** as well. To prove this, we will give a mapping reduction from A_{TM} to EQ_{TM}.

Given a TM/string pair $\langle M, w \rangle$, define $f(\langle M, w \rangle) = \langle \text{Amp}(M, w), S \rangle$, where S is any TM whose language is Σ^* (for example, $S =$ "On input x, accept"). Note that f is a computable function.

To prove that f is a mapping reduction from A_{TM} to EQ_{TM}, we will prove that for any TM M and string w, that $\langle M, w \rangle \in A_{TM}$ iff $\langle \text{Amp}(M, w), S \rangle \in EQ_{TM}$. To see this, note that by the definition of A_{TM}, we have $\langle M, w \rangle \in A_{TM}$ iff M accepts w. By a theorem proven in lecture, we know that M accepts w iff $\mathscr{L}(\text{Amp}(M, w)) = \Sigma^*$. Since $\mathscr{L}(S) = \Sigma^*$, the previous statement is equivalent to the statement that M accepts w iff $\mathscr{L}(\text{Amp}(M, w)) = \mathscr{L}(S)$. Finally, by the definition of EQ_{TM}, we know that $\mathscr{L}(\text{Amp}(M, w)) = \mathscr{L}(S)$ iff $\langle \text{Amp}(M, w), S \rangle \in EQ_{TM}$. Combining these statements together, we see that $\langle M, w \rangle \in A_{TM}$ iff $\langle \text{Amp}(M, w), S \rangle \in EQ_{TM}$. Thus f is a mapping reduction from A_{TM} to EQ_{TM}, so $A_{TM} \leq_M EQ_{TM}$, as required. ■

We've used mapping reductions to relate the difficulty of **R**, **RE**, and co-**RE** languages to one another. Could we use them to relate problems from classes of languages as well? Usually, the answer is no.

ii. Find an pair of languages L_1 and L_2 where $L_1 \leq_M L_2$ and L_2 is regular, but L_1 is not regular. Briefly explain why L_1 is not regular and why L_2 is regular, then prove that $L_1 \leq_M L_2$ by exhibiting a mapping reduction from L_1 to L_2. This means that we cannot establish that a language is regular by finding a mapping reduction from it to a known regular language.

To motivate why it is that we've defined reductions as we have, suppose that we change the definition of a mapping reduction as follows: For languages A and B, we say that $A \leq_M B$ iff there exists a computable function f such that for any $w \in A$, we have $f(w) \in B$.

iii. Prove that under this modified definition, *every* language A is mapping reducible to *every* language $B \neq \varnothing$. This (hopefully!) explains why we didn't define reductions this way.

Problem Two: Disjoint Unions (16 Points)

Given two languages L_1 and L_2, their union $L_1 \cup L_2$ is the set of all strings in at least one of L_1 and L_2. However, there is usually no connection between the "hardness" of L_1 and L_2 and the "hardness" of $L_1 \cup L_2$. For example, L_D is unrecognizable and $\overline{L}_D$ is co-unrecognizable, but $L_D \cup \overline{L}_D = \Sigma^*$ is decidable (in fact, it's regular!). The reason for this is that to decide whether $w \in L_D \cup \overline{L}_D$, we don't actually need to check whether $w \in L_D$ or $w \in \overline{L}_D$.

A more interesting construction is the *disjoint union* of two languages, a way of combining together strings from two different languages that tags each string with information about which language it came from. In what follows, let's assume all languages are over the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$. Formally, the disjoint union of two languages L_1 and L_2 is the language

$$L_1 \uplus L_2 = \{ \texttt{0}w \mid w \in L_1 \} \cup \{ \texttt{1}w \mid w \in L_2 \}$$

For example, if $L_1 = \{ \texttt{1}, \texttt{10}, \texttt{100}, \texttt{1000} \}$ and $L_2 = \{ \varepsilon, \texttt{0}, \texttt{1}, \texttt{00}, \texttt{01}, \texttt{10}, \texttt{11} \}$, then $L_1 \uplus L_2$ is the set

$$L_1 \uplus L_2 = \{ \texttt{01}, \texttt{010}, \texttt{0100}, \texttt{01000}, \texttt{1}, \texttt{10}, \texttt{11}, \texttt{100}, \texttt{101}, \texttt{110}, \texttt{111} \}$$

Notice how each string in $L_1 \uplus L_2$ is tagged with which language it originated in. Any string that starts with `0` came from L_1, and any string that starts with `1` came from L_2. Because of this tagging, the disjoint union of two languages produces a new language that is at least as "hard" as either of the input languages. In this problem, you will prove various important properties about the disjoint union.

i. Prove that if L_1 and L_2 are any languages, then $L_1 \leq_M L_1 \uplus L_2$. A similar proof can be used to show that $L_2 \leq_M L_1 \uplus L_2$, but you don't need to do that here.

ii. Prove that if L is recognizable but undecidable, then $L \uplus \overline{L}$ is neither **RE** nor co-**RE**.

Your result from (ii) shows how to construct extraordinarily hard problems out of problems that are already known to be hard. For example, $A_{TM} \uplus \overline{A}_{TM}$ and $L_D \uplus \overline{L}_D$ are neither **RE** nor co-**RE**.

Problem Three: A_{TM} and L_D (12 Points)

When we first saw the language L_D, we described it as a problem that was "harder" than A_{TM} because A_{TM} is **RE** while L_D is not. However, this might not have been fair. Both L_D and A_{TM} are hard problems, but neither one is "harder" than the other in the sense that neither one is mapping reducible to the other. Prove that $A_{TM} \not\leq_M L_D$ and that $L_D \not\leq_M A_{TM}$.

Problem Four: RE-Completeness (20 Points)

A language L is called ***RE-hard*** iff for any language $L' \in$ **RE**, we have $L' \leq_M L$. In other words, L is **RE**-hard iff every language in **RE** reduces to L. Intuitively, **RE**-hard problems are "at least as hard" as all the problems in **RE**, since a TM for an **RE**-hard problem could be used to solve every problem in **RE**.

i. Prove that all **RE**-hard problems are undecidable.

ii. Prove that if L is **RE**-hard and $L \leq_M L'$, then L' is **RE**-hard. *(Hint: Prove $\leq_M$ is transitive.)*

A language L is called ***RE-complete*** iff $L \in$ **RE** and L is **RE**-hard. Intuitively, an **RE**-complete language is one of the hardest problems in **RE**, since it's in **RE** but "at least as hard" as any **RE** language.

iii. Prove that A_{TM} is **RE**-hard. Since $A_{TM} \in$ **RE**, this proves that A_{TM} is **RE**-complete.

Problem Five: Accept all the Strings! (20 Points)*

Consider the language

$$A_{ALL} = \{ \langle M \rangle \mid \mathscr{L}(M) = \Sigma^* \}$$

This language is neither **RE** nor co-**RE**, and in this problem you will see why.

i. Prove that $A_{TM} \leq_M A_{ALL}$. Since $A_{TM} \notin$ co-**RE**, this proves that $A_{ALL} \notin$ co-**RE** either.

The trickier part of the proof is proving that A_{ALL} is not **RE**. To do this, we will reduce $\overline{A}_{TM} \leq_M A_{ALL}$. Since $\overline{A}_{TM} \notin$ **RE**, this proves that $A_{ALL} \notin$ **RE** either.

Suppose that you are given a Turing machine M and a string w. We can construct a new TM N as follows:

> N = "On input x:
> Run M on w for $|x|$ steps.
> If M accepted within $|x|$ steps, reject.
> Otherwise, accept."

For example, on input `000`, N would run M on w for 3 steps, rejecting if M accepted w within that time and accepting otherwise. Similarly, if N were run on `010101`, N would accept if M did not accept w within 6 steps and would reject otherwise.

ii. Prove that M does not accept w iff $\mathscr{L}(N) = \Sigma^*$.

iii. Using your answer from (ii), prove that $\overline{A}_{TM} \leq_M A_{ALL}$. Since $\overline{A}_{TM}$ is not **RE**, this proves that A_{ALL} is not **RE** either.

Problem Six: Complementary Turing Machines (12 Points)

Consider the language *COMPLEMENT* defined as follows:

$$COMPLEMENT = \{ \langle M, N \rangle \mid M \text{ and } N \text{ are TMs and } \mathscr{L}(M) = \overline{\mathscr{L}(N)} \}$$

Prove that *COMPLEMENT* is neither **RE** nor co-**RE** by showing that $A_{ALL} \leq_M COMPLEMENT$. Make sure you justify how this reduction proves that *COMPLEMENT* is neither **RE** nor co-**RE**.

---

* Original image by Allie Brosh. This image courtesy of quickmeme.com.

Problem Seven: Why All This Matters (20 Points)

Why are we studying A_{TM}, L_D, and these other "unsolvable" problems? It turns out that the fact that these problems are "unsolvable" has enormous practical implications for real computing systems.

Since their memory is finite, computers are not as powerful as Turing machines. However, as computers get more and more memory, we can think of them as getting progressively closer and closer in power to Turing machines. For the purposes of this question, we'll assume that a standard computer is equivalent in power to a Turing machine.

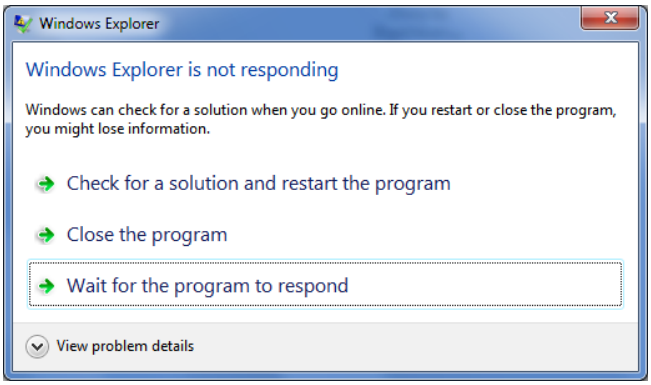
Because Turing machines and equivalently powerful models of computation can simulate one another, it is possible in any reasonable programming language to write a function like this one:

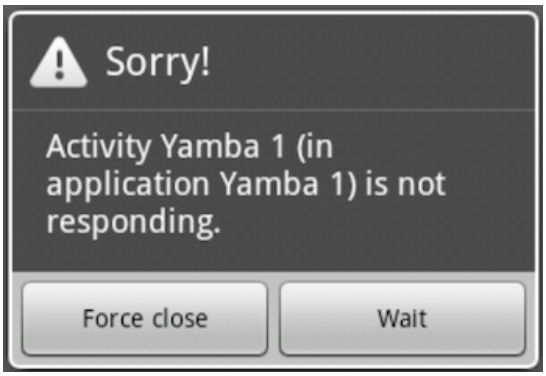
```
boolean simulateTuringMachine(TM M, string w)
```

This function takes in a suitably-encoded Turing machine M and a string w, then runs M on w. If M accepts w, then this function returns `true`. If M rejects w, then this function returns `false`. If M loops on w, then this function loops forever and never returns.

Using the existence of the above function, and the fact that a TM can simulate a computer, answer the following questions.

i. Most operating systems provide some functionality to detect programs that are looping infinitely. Typically, they display a dialog box containing a message like these shown below:





These messages give the user the option to terminate the program or to let the program keep running in the hopes that it stops looping.

An ideal OS would shut down any program that had gone into an infinite loop, since these programs just waste system resources (processor time, battery power, etc.) that could be better spent by other programs. It makes more sense for the OS to automatically detect programs that have gone into an infinite loop.

Show that no OS can detect all programs that have gone into an infinite loop. Specifically, prove the following: if there were an OS that could always detect programs that have entered an infinite loop, then $\overline{HALT} \in$ **RE**. *(Hint: you will need to use the fact that computers can simulate TMs and TMs can simulate computers. Try embedding a Turing machine into a computer program and tricking an OS that can detect infinite loops into accidentally recognizing $\overline{HALT}$.)*

(Continued on the next page.)

ii. Suppose that you want to create a website that teaches people how to program. On this site, you give a set of programming problems and invite users to submit programs that solve those problems. For each programming problem, you write a small set of test cases that submitted programs should be able to pass. Each test cases consists of a set of inputs to the user's program, along with the expected outputs. You can assume that all that matters is whether the program eventually outputs the right answer, not how long it takes to do so.

Show that it is impossible to automatically decide whether an arbitrary submitted program passes these tests. Specifically, prove the following: if a program could take in an input program and an arbitrary set of test cases and could decide whether the program passes the tests, then $A_{TM} \in \mathbf{R}$. *(Hint: Again, use the fact that TMs can simulate computers and computers simulate TMs. Try embedding a TM within a program so that the behavior of the program depends on what the TM does on the string. Then, specify some test cases to check the behavior of the program.)*

Problem Eight: Course Feedback (5 Points)

We want this course to be as good as it can be, and we'd really appreciate your feedback on how we're doing. For a free five points, please answer the feedback questions for this problem set online at https://docs.google.com/forms/d/1_-nMscC_9Cr3XaL755ltiFqXw-rhsi4j2GlyjuqCNeo/viewform. We'll give you full credit no matter what you write (as long as you write something!), but we'd appreciate it if you're honest about how we're doing.

Extra Credit Problem: A Smaller Version of A_{TM} (5 Points)

For any natural number k, define the language $A_{TM}^{(k)}$ as

$$A_{TM}^{(k)} = \{ \langle M, w \rangle \mid M \text{ is a TM with at most } k \text{ states and } M \text{ accepts } w \}$$

Prove that there exists a constant $k \in \mathbb{N}$ such that $A_{TM}^{(k)}$ is undecidable.