# Problem Set 8 Solutions

## 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!

The problem with this proof is that showing that $A_{TM} \leq_M EQ_{TM}$ does not show that $EQ_{TM}$ is **RE**. The reduction done in the proof is a valid reduction, but the conclusion being drawn from it is incorrect. To show that $EQ_{TM}$ is **RE**, we'd have to show that $EQ_{TM} \leq_M A_{TM}$. On an unrelated note, showing that $A_{TM} \leq_M EQ_{TM}$ does prove that $EQ_{TM}$ is not co-**RE**.

ii. Find an pair of languages $L_1$ and $L_2$ where $L_1 \leq_M L_2$, $L_2$ is regular, but $L_1$ is not regular, then prove that this is the case 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.

One option is the following: Choose $L_1 = \{ \texttt{0}^n\texttt{1}^n \mid n \in \mathbb{N} \}$ and $L_2 = \texttt{0}^*$. Then $L_1$ is not regular (we have proven this in lecture), while $L_2$ is because there is a regular expression for it. However:

*Theorem:* $L_1 \leq_M L_2$.

*Proof:* We exhibit a mapping reduction from $L_1$ to $L_2$. Let $f(w)$ be defined as follows:

$$f(w) = \texttt{0} \quad \text{if } w = \texttt{0}^n\texttt{1}^n \text{ for some } n \in \mathbb{N}$$

$$f(w) = \texttt{1} \quad \text{otherwise}$$

This reduction is computable, because a TM can decide whether or not $w$ has the form $\texttt{0}^n\texttt{1}^n$ for some $n \in \mathbb{N}$ (we could, for example, run the decider for this language that we built in lecture, clear off the tape when it's done running, and then write either `0` or `1`). Moreover, we claim that $w \in L_1$ iff $f(w) \in L_2$. To see this, note that if $w \in L_1$, then $w$ has the form $\texttt{0}^n\texttt{1}^n$ for some $n \in \mathbb{N}$, and therefore $f(w) = \texttt{0} \in L_2$. If $w \notin L_1$, then $w$ does not have the form $\texttt{0}^n\texttt{1}^n$ for any $n \in \mathbb{N}$, and so $f(w) = \texttt{1} \notin L_2$. Consequently, $f$ is a mapping reduction from $L_1$ to $L_2$, and so $L_1 \leq_M L_2$. ■

The reason that this reduction is possible is that mapping reductions have enormous computational power – they can solve any problem in **R** as a subroutine – and so can easily reduce recursive but nonregular languages to regular languages.

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

The intuition behind what's wrong here is that since there is no restriction on what happens to strings not contained in $A$, we can build up a completely trivial mapping from $A$ to any nonempty language $B$ by just mapping everything to some fixed string that happens to be in $B$. The proof is shown here:

*Theorem:* Under the modified definition of mapping reductions, $A \leq_M B$ for any language $A$ and any language $B \neq \emptyset$.

*Proof:* Let $A$ be any language and $B$ be any language other than $\emptyset$. Since $B \neq \emptyset$, there exists some string $w_{yes} \in B$. Then define $f(w) = w_{yes}$ for any $w \in \Sigma^*$. $f$ is computable, since we can build a TM that blanks out the tape and then writes $w_{yes}$ onto it. Moreover, for any $w \in A$, $f(w) = w_{yes} \in B$. Thus, under the new definition of mapping reductions, $A \leq_M B$. ■

A subtle point not captured in this proof is that we don't need to have a way to actually *find* the string $w_{yes}$ that we are using in this construction. Mathematically speaking, for any nonempty language $B$ there has to be some string $w_{yes} \in B$, and therefore mathematically there is a Turing machine that blanks out its tape on startup and then writes $w_{yes}$. We might not know whether or not we had found such a Turing machine when it was given to us (since $B$ might be an unrecognizable language), but we can guarantee that it exists.

**Why we asked this question:** This question was all about sharpening your intuition for what mapping reducibility actually means. Part (i) of this question was designed to make sure you understood the directionality behind reductions: they transmit easiness in one direction and hardness in another.

Part (ii) of this question was interesting. Our hope was that this problem would help you better understand exactly what a computable function is and why mapping reductions can't be used to talk about other complexity classes. In solving this problem, we hoped that you would recognize that because the reduction just has to be a computable function, it's possible to do an *enormous* amount of work when computing a reduction. This large amount of work is precisely what makes mapping reductions not preserve regularity.

A common mistake we saw on part (ii) of this problem was to define a function $f$ only on strings in the nonregular language without specifying how it behaves on strings not in the nonregular language. Remember that mapping reductions are defined on all strings, not just strings in the source language.

Finally, part (iii) of this question was designed to make sure you understood why the biconditional was necessary in the definition of mapping reductions. Reductions need to work on all strings, not just strings in the language. If reductions don't always preserve the correct answer, then the reduction might not actually tell you anything about the two languages connected by the reduction.

## Problem Two: Disjoint Unions (16 Points)

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.

*Proof:* We exhibit a mapping reduction from $L_1$ to $L_1 \uplus L_2$. Let $f(w) = \texttt{0}w$. We claim that $w \in L_1$ iff $f(w) \in L_1 \uplus L_2$. To see this, note that by construction $w \in L_1$ iff $f(w) = \texttt{0}w \in L_1 \uplus L_2$. Moreover, $f(w)$ is computable, since we can design a TM that simply shifts $w$ over by one step on the tape and writes a `0` to the front. Since there is a mapping reduction from $L_1$ to $L_1 \uplus L_2$, we have that $L_1 \leq_M L_1 \uplus L_2$. ■

The proof that $L_2 \leq_M L_1 \uplus L_2$ is analogous; just substitute `1` for `0` and $L_2$ for $L_1$.

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

*Proof*: Since $L$ is recognizable but undecidable, we know that $\overline{L} \notin$ **RE**, since otherwise we would have that $L \in$ **R**, contradicting the fact that $L$ is undecidable. Since $\overline{L} \notin$ **RE**, we know that $L \notin$ co-**RE**. Using our results from (i), because $L \notin$ co-**RE** and $L \leq_M L \uplus \overline{L}$, this means that $L \uplus \overline{L} \notin$ co-**RE** either. Also, since $\overline{L} \notin$ **RE** and $\overline{L} \leq_M L \uplus \overline{L}$, this means that $L \uplus \overline{L} \notin$ **RE** either. Consequently, $L \uplus \overline{L}$ is neither **RE** nor co-**RE**. ■

**Why we asked this question:** Almost all of the reductions we've done so far have involved TMs, but there's no fundamental reason why this must be the case. Part (i) of this problem boils down to a very simple reduction (namely, prepend a `0` to the input string), and part (ii) uses the fact that this reduction works in order to show that the disjoint union of a language and its complement can be ferociously hard to solve.

Part (ii) of this problem was also designed to help you navigate the space of **RE** and co-**RE**. The key insight necessary to solve this problem is that any language that is in **RE** – **R** cannot be co-**RE**, and its complement must be co-**RE** but not **RE**.

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

*Proof:* We first show that there is no mapping reduction from $A_{TM}$ to $L_D$, then the converse.

To see that there is no mapping reduction from $A_{TM}$ to $L_D$, we proceed by contradiction; assume that $A_{TM} \leq_M L_D$. Because $L_D \in$ co-**RE**, we see that $A_{TM} \in$ co-**RE**, which we know is false. We have reached a contradiction, so our assumption must have been wrong and there is no mapping reduction from $A_{TM}$ to $L_D$.

To see that there is no mapping reduction from $L_D$ to $A_{TM}$, we proceed by contradiction; assume that $L_D \leq_M A_{TM}$. Then since $A_{TM} \in$ **RE**, this means that $L_D \in$ **RE**, which we know is false. We have reached a contradiction, so our assumption must have been wrong. Thus there is no mapping reduction from $L_D$ to $A_{TM}$. ■

**Why we asked this question:** We use reducibility to connect together the relative difficulties of problems. By showing that $A_{TM}$ and $L_D$ aren't reducible to one another in either direction, we hoped that you would see that **RE** – **R** and co-**RE** – **R** are incomparably difficult classes of problems.

Interestingly, this proof can easily be generalized to show that for any undecidable languages $L_1$ and $L_2$ where $L_1 \notin$ **RE** and $L_2 \notin$ co-**RE**, that $L_1$ and $L_2$ cannot be reduced to one another.

## Problem Four: RE-Completeness (20 Points)

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

*Proof:* Let $L$ be an **RE**-hard problem. Since $A_{TM} \in \mathbf{RE}$, we know that $A_{TM} \leq_M L$. Because $A_{TM} \notin \mathbf{R}$, this means that $L \notin \mathbf{R}$. Since our choice of $L$ was arbitrary, this proves that any **RE**-hard problem is 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.)*

*Proof:* Let $L$ be an **RE**-hard language and $L'$ a language where $L \leq_M L'$. Now, consider any **RE** language $L''$. We will prove that $L'' \leq_M L'$, from which we can conclude that $L'$ is **RE**-hard.

Since $L$ is **RE**-hard, we know that $L'' \leq_M L$. Thus there is a computable function $f$ where $w \in L''$ iff $f(w) \in L$. Since $L \leq_M L'$, there is a computable function $g$ such that $w \in L$ iff $g(w) \in L'$. Now consider the function $h$ defined as $h(w) = g(f(w))$. First, we claim that $h$ is computable. To see this, note that we can build a TM to evaluate $h$ by first running the TM that computes $f$ to compute $f(w)$ and then run the TM that computes $g$ on that result to get $g(f(w)) = h(w)$. Since both $f$ and $g$ are computable, each intermediate computation terminates, so the overall computation terminates. Next, we prove that $w \in L''$ iff $h(w) \in L'$. Then $w \in L''$ iff $f(w) \in L$ iff $g(f(w)) \in L''$. Since $h(w) = g(f(w))$, this means $w \in L''$ iff $h(w) \in L''$. Thus $h$ is a mapping reduction from $L''$ to $L'$, so $L'' \leq_M L'$, as required. ■

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

*Proof:* Let $L$ be any **RE** language. We will prove that $L \leq_M A_{TM}$, from which we get that $A_{TM}$ is **RE**-hard.

Since $L \in \mathbf{RE}$, there is some TM $M$ where $\mathscr{L}(M) = L$. Define $f(w) = \langle M, w \rangle$, which is a computable function because $\langle M \rangle$is a fixed string. We claim that $w \in L$ iff $\langle M, w \rangle \in A_{TM}$. To see this, note that $w \in L$ iff $w \in \mathscr{L}(M)$. By definition of $\mathscr{L}(M)$, we know that $w \in \mathscr{L}(M)$ iff $M$ accepts $w$. Finally, by definition of $A_{TM}$, we know that $M$ accepts $w$ iff $\langle M, w \rangle \in A_{TM}$. Combining these steps together, we see that $w \in L$ iff $\langle M, w \rangle \in A_{TM}$. Therefore, $f$ is a mapping reduction from $L$ to $A_{TM}$, so $L \leq_M A_{TM}$, as required. ■

**Why we asked this question:** We asked this question for several reasons. First, this question motivates the related concepts of **NP**-hardness and **NP**-completeness that we started to explore on Monday. We hoped that by asking you to work through this problem in the context of **RE**, you would have a better intuition for the analogous results for **NP**.

Part (i) of this problem was designed to help you understand just how hard **RE**-hard languages are: they're at least undecidable, and possibly a lot harder. Part (ii) of this problem shows how reducibility transmits **RE**-hardness and is analogous to the result about **NP**-hardness. Finally, part (iii) of this problem shows that **RE**-hard problems actually do exist and that **RE**-completeness is actually possible.

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

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

*Theorem:* $A_{ALL} \notin$ co-**RE**.

*Proof:* We will prove that $A_{TM} \leq_M A_{ALL}$. Since $A_{TM} \notin$ co-**RE**, this proves that $A_{ALL} \notin$ co-**RE**, as required.

To prove $A_{TM} \leq A_{ALL}$, we exhibit a mapping reduction from $A_{TM}$ to $A_{ALL}$. For any TM/string pair $\langle M, w \rangle$, let $f(\langle M, w \rangle) = \langle \text{Amp}(M, w) \rangle$. From lecture, we know that $f$ is computable.

We further claim that $\langle M, w \rangle \in A_{TM}$ iff $\langle \text{Amp}(M, w) \rangle \in A_{ALL}$. To see this, note that by the definition of $A_{TM}$ we have $\langle M, w \rangle \in A_{TM}$ iff $M$ accepts $w$. As proven in lecture, $M$ accepts $w$ iff $\mathscr{L}(\text{Amp}(M, w)) = \Sigma^*$. Finally, by the definition of $A_{ALL}$, we know $\mathscr{L}(\text{Amp}(M, w)) = \Sigma^*$ iff $\langle \text{Amp}(M, w) \rangle \in A_{ALL}$. Combining all of these steps together, we see that $\langle M, w \rangle \in A_{TM}$ iff $\langle \text{Amp}(M, w) \rangle \in A_{ALL}$. Thus $f$ is a mapping reduction from $A_{TM}$ to $A_{ALL}$, so $A_{TM} \leq_M A_{ALL}$, as required. ■

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

*Proof:* We prove both directions of implication. First, assume that $M$ does not accept $w$. In that case, if we run $M$ on $w$ for any number of steps, it will not accept. Thus if we run $N$ on any string $x$, $M$ will not accept $w$ within $|x|$ steps, and so $N$ will accept $x$. Since our choice of $x$ was arbitrary, this shows that if $M$ accepts $w$, then $N$ accepts all strings. Thus $\mathscr{L}(N) = \Sigma^*$.

Now, assume that $M$ accepts $w$. In that case, it does so in a finite number of steps; let that number of steps be $n$. Then if we run $N$ on any string $x$ of length at least $n$, $M$ will accept $w$ within $|x|$ steps. Consequently, $N$ will reject $x$, so $x \notin \mathscr{L}(N)$. Since there is some string $x$ that is not contained in $\mathscr{L}(N)$, we know that $\mathscr{L}(N) \neq \Sigma^*$, as required. ■

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.

*Proof:* To prove $\overline{A}_{TM} \leq A_{ALL}$, we exhibit a mapping reduction from $\overline{A}_{TM}$ to $A_{ALL}$. For any TM/string pair $\langle M, w \rangle$, let $f(\langle M, w \rangle) = \langle N \rangle$, where $N$ is the machine defined in part (ii). We state without proof that $f$ is computable. We further claim that $\langle M, w \rangle \in \overline{A}_{TM}$ iff $\langle N \rangle \in A_{ALL}$. To see this, note that $f(\langle M, w \rangle) = \langle N \rangle \in A_{ALL}$ iff $\mathscr{L}(N) = \Sigma^*$. As we proved in part (ii), $\mathscr{L}(N) = \Sigma^*$ iff $M$ does not accept $w$. Finally, note that $M$ does not accept $w$ iff $\langle M, w \rangle \in \overline{A}_{TM}$. Thus we have $\langle M, w \rangle \in \overline{A}_{TM}$ iff $\langle N \rangle \in A_{ALL}$, so $f$ is a mapping reduction from $\overline{A}_{TM}$ to $A_{ALL}$. Therefore, $\overline{A}_{TM} \leq_M A_{ALL}$. ■

**Why we asked this question:** This question gave a different example of a language that's neither **RE** nor co-**RE**: the language of all TMs that accept everything.

Part (i) of this problem was designed to make sure you were comfortable writing reductions that involved putting machines inside other machines (here, by using the amplifier machine). Parts (ii) and (iii) were designed to show you another example of a reduction that works by putting TMs inside other TMs, here, using a different TM from the amplifier. This "nega-amplifier" TM is remarkably useful in other reductions and acts as a complement to the amplifier.

## Problem Six: Complementary Turing Machines (12 Points)

*Proof:* We will prove that $A_{ALL} \leq_M$ *COMPLEMENT*. Since $A_{ALL}$ is neither **RE** nor co-**RE**, this proves that *COMPLEMENT* is neither **RE** nor co-**RE**.

Let's define the TM $E$ as $E$ = "On input $w$, reject." This means that $\mathscr{L}(E) = \varnothing$. Now, consider the function $f(\langle M \rangle) = \langle M, E \rangle$, which is computable. We further claim for all TMs $M$ that $\langle M \rangle \in A_{ALL}$ iff $\langle M, E \rangle \in$ *COMPLEMENT*. To see this, note that by definition of $A_{ALL}$, $\langle M \rangle \in A_{ALL}$ iff $\mathscr{L}(M) = \Sigma^*$. Since $\Sigma^* = \overline{\varnothing}$, we have that $\mathscr{L}(M) = \Sigma^*$ iff $\mathscr{L}(M) = \overline{\varnothing}$. Since $\mathscr{L}(E) = \varnothing$, we have that $\mathscr{L}(M) = \overline{\varnothing}$ iff $\mathscr{L}(M) = \overline{\mathscr{L}(E)}$. Finally, by definition of *COMPLEMENT*, we see that $\mathscr{L}(M) = \overline{\mathscr{L}(E)}$ iff $\langle M, E \rangle \in$ *COMPLEMENT*. Combining these steps together, we get that $\langle M \rangle \in A_{ALL}$ iff $\langle M, E \rangle \in$ *COMPLEMENT*. Thus $f$ is a mapping reduction from $A_{ALL}$ to *COMPLEMENT*, so $A_{ALL} \leq_M$ *COMPLEMENT*, as required. ■

**Why we asked this question:** Although reducibility among problems involving TMs often involves putting TMs inside other TMs, this isn't always the case. We hoped that this problem would be a simple example of a reduction between TMs that doesn't involve any crazy constructions with amplifier machines.

## Problem Seven: Why All This Matters (20 Points)

i. 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* $\overline{HALT}$*.)*

*Proof:* By contradiction; assume that there is an OS $O$ that can detect programs that loop infinitely. Now consider the follow TM $H$:

> $H$ = "On input $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string:
>
> Construct a computer program $P$ that simulates $M$ on $w$, then terminates.
>
> Simulate OS $O$ running on program $P$.
>
> If $O$ reports that $P$ loops infinitely, accept.
>
> If program $P$ terminates, reject."

We claim that $\mathscr{L}(H) = \overline{HALT}$, which gives a contradiction because it implies that $\overline{HALT} \in$ **RE**, which we know is false. From this, we can conclude our initial assumption was incorrect, so there is no OS that can detect programs that loop infinitely.

To see that $\mathscr{L}(H) = \overline{HALT}$, note that $H$ accepts $\langle M, w \rangle$ iff the OS $O$ detects that program $P$ loops infinitely. Since by construction program $P$ runs $M$ on $w$ and terminates when $M$ halts, program $P$ loops infinitely iff $M$ loops infinitely on $w$. Thus $H$ accepts $\langle M, w \rangle$ iff $M$ loops infinitely on $w$ iff $\langle M, w \rangle \in \overline{HALT}$. Thus we have $\mathscr{L}(H) = \overline{HALT}$, as required. ■

ii. 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.)*

*Proof:* By contradiction; assume there is a program $V$ that can accept a program and test cases and decide whether the program passes the test cases. Then consider the following TM $H$:

> $H$ = "On input $\langle M, w \rangle$, where $M$ is a TM and $w$ is a string:
>
> Construct a program $P$ that runs $M$ on $w$, then prints "Accept" if $M$ accepts $w$ and prints "Reject" if $M$ rejects $w$.
>
> Simulate the execution of $V$ on program $P$, given the test case "$P$ prints 'Accept.'"
>
> If $V$ says $P$ passes the tests, accept.
>
> If $V$ says $P$ fails the tests, reject."

We claim that $H$ is a decider for $A_{TM}$, from which we get a contradiction since $A_{TM}$ is undecidable. This would mean our assumption is wrong, so no program $V$ exists.

To show this, we prove first that $H$ is a decider, then that $\mathscr{L}(H) = A_{TM}$. To see that $H$ is a decider, note that on any input $\langle M, w \rangle$, $H$ constructs the program $P$ (which can be done in finite time by just hardcoding $M$ and $w$ into $P$), then simulates $V$ on $P$ and a test case. Since $V$ always terminates, it either reports that $V$ passes or fails the tests. In either case, $H$ immediately terminates. Thus $H$ is a decider.

To show that $\mathscr{L}(H) = A_{TM}$, note that $H$ accepts $\langle M, w \rangle$ iff $V$ reports that $P$ passes the test case "$P$ prints 'Accept.'" This happens iff $P$ actually prints "Accept," which by construction happens iff $M$ accepts $w$. Finally, $M$ accepts $w$ iff $\langle M, w \rangle \in A_{TM}$. Thus $\mathscr{L}(H) = A_{TM}$, as required. ■

**Why we asked this question:** This question, probably the trickiest on the set, was designed to make sure you understood exactly why all of the results we've talked about so far have any practical relevance. It strayed away from mapping reducibility to hit at the core idea behind reducibility – it's possible to turn machines that solve one problem into machines that solve another, and in doing so we can prove that certain machines can't exist.

Part (i) of this problem was designed as a warm-up. To solve it, you need to realize that you could run a program inside the OS that simulates a TM. You can then simulate this entire process to build a TM that can recognize an unrecognizable language (namely, the set of all TM/string pairs where the TM loops on its input.) Part (ii) of this problem was designed as a more elaborate example that involves putting a TM inside a program, but then having that program base its behavior on what the TM actually does.