# Problem Set 7

This problem explores Turing machines, nondeterministic computation, properties of the **RE** and **R** languages, and the limits of **RE** and **R** languages. This will be your first experience exploring the limits of computation, and I hope that you find it exciting!

In any question that asks for a proof, you **must** provide a rigorous mathematical proof. You cannot draw a picture or argue by intuition. You should, at the very least, state what type of proof you are using, and (if proceeding by contradiction, contrapositive, or induction) state exactly what it is that you are trying to show. If we specify that a proof must be done a certain way, you must use that particular proof technique; otherwise you may prove the result however you wish.

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. The earlier questions serve as a warm-up for the later problems, so do be aware that the difficulty of the problems does increase over the course of this problem set.

Good luck, and have fun!

**Due Monday, March 4th at 12:50 PM**

## A Note on Turing Machine Design

Many questions in this problem set will ask you to design Turing machines that solve various problems. In some cases, we will want you to write out the states and transitions within the Turing machine, and in other cases you will only need to provide a high-level description.

If a problem asks you to **draw the state-transition diagram for a Turing machine**, we expect you to draw out a concrete Turing machine by showing the states in that Turing machine and the individual transitions between them. If a question asks you to do this, as a courtesy to your TAs, please include with your Turing machines the following information:

- A short, one-paragraph description of the high-level operation of the machine.
- A brief description of any subroutines in the Turing machine or any groups of states in the Turing machine that represent storing a constant in the TM's finite-state control.
- A one-sentence description of each state in the machine.

For simplicity, you may assume that all missing transitions implicitly cause the TM to reject.

If a problem asks you to **give a high-level description of a Turing machine**, you can just provide a high-level description of the machine along the lines of what we did in lecture. More generally, unless you are specifically asked to give a state-transition diagram, any time that you are asked to design a Turing machine, you are encouraged to do so by giving a high-level description.

If you have any questions about this, please feel free to ask!

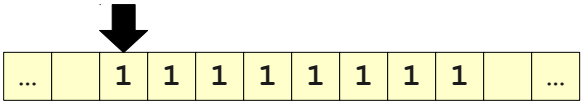
## Problem One: The Collatz Conjecture (24 Points)

In last Friday's lecture, we discussed the *Collatz conjecture*, which claims that the following procedure (called the *hailstone sequence*) terminates for all positive natural numbers $n$:

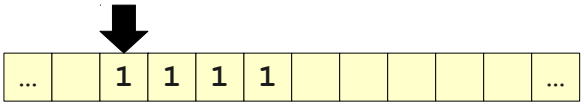
- If $n = 1$, stop.
- If $n$ is even, set $n = n / 2$.
- If $n$ is odd, set $n = 3n + 1$.
- Repeat.

In lecture, we claimed that it was possible to build a TM for the language $L = \{ \texttt{1}^n \mid$ the hailstone sequence terminates for $n \}$ over the alphabet $\Sigma = \{\texttt{1}\}$. In this problem, you will do exactly that. The first two parts to this question ask you to design key subroutines for the TM, and the final piece asks you to put everything together to assemble the final machine.

i. Draw the state transition diagram for a Turing machine that, when given a tape holding $\texttt{1}^{2n}$ surrounded by infinitely many blanks, ends with $\texttt{1}^n$ written on its tape, surrounded by infinitely many blanks. You can assume the tape head begins reading the first `1`, and your TM should end with the tape head reading the first `1` of the result. For example, given this initial configuration:

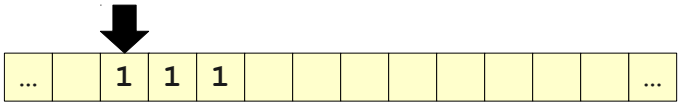



The TM would end with this configuration:

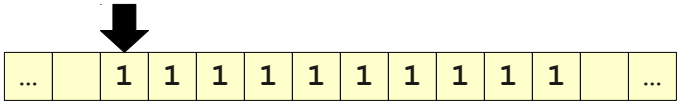



You can assume that there are an even number of `1`s on the tape at startup and can have your TM behave however you'd like if this isn't the case. Please provide a description of your TM as discussed at the start of this problem set.

ii. Draw the state transition diagram for a Turing machine that, when given a tape holding $\texttt{1}^n$ surrounded by infinitely many blanks, ends with $\texttt{1}^{3n+1}$ written on its tape, surrounded by infinitely many blanks. You can assume that the tape head begins reading the first `1`, and your TM should end with the tape head reading the first `1` of the result. For example, given this configuration:




The TM would end with this configuration:




Please provide a description of your TM as discussed at the start of this problem set.

iii. Using your TMs from parts (i) and (ii) as subroutines, draw the state transition diagram for a Turing machine $M$ that recognizes $L$. You do not need to copy your machines from part (i) and (ii) into the resulting machine. Instead, you can introduce "phantom states" that stand for the entry or exit states of those subroutines and then add transitions into or out of those states. Please provide a description of your TM as discussed at the start of this problem set.

## Problem Two: Manipulating Encodings (16 Points)

In what follows, you can assume that $\Sigma = \{\texttt{0}, \texttt{1}\}$.

In Monday's lecture, we discussed string encodings of objects and ways in which TMs could manipulate those encodings. To help give you a better feeling for why this is possible, this question asks you to design two TM subroutines to perform common manipulations on encodings.

When discussing encodings, we saw that it was possible to take two encodings of objects $\langle O_1 \rangle$ and $\langle O_2 \rangle$ and combine them together to form a single string $\langle O_1, O_2 \rangle$ that encodes both of those objects. The specific encoding scheme we suggested was the following: the string $\langle O_1, O_2 \rangle$ is the string formed by

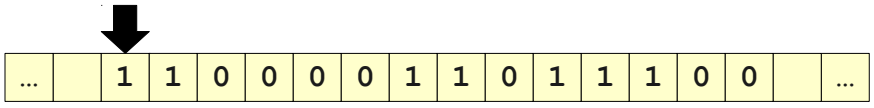
- Doubling each character in $\langle O_1 \rangle$ (i.e. `0` becomes `00` and `1` becomes `11`),
- then writing out the string `01` as a delimiter, and finally
- writing out the description of $\langle O_2 \rangle$ unchanged.

For example, suppose that $\langle O_1 \rangle$ = `1010` and $\langle O_2 \rangle$ = `11111`. Then the encoding $\langle O_1, O_2 \rangle$ would be the string **`11001100`**`01`**`11111`** (here, I've underlined the parts of the encoding corresponding to the original encodings.

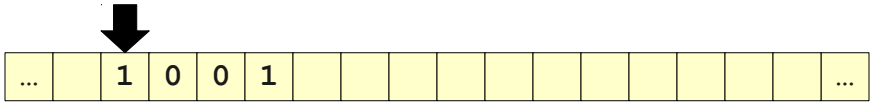
In order for this representation to be useful, Turing machines need to be to extract the first and second part of an encoded pair. This problem asks you to design TMs that do precisely these tasks.

i. Draw the state transition diagram for a Turing machine that, given an encoding $\langle O_1, O_2 \rangle$ of two objects, ends with the string $\langle O_1 \rangle$ written on its tape, surrounded by infinitely many blanks. You can assume that the tape head begins reading the first character of $\langle O_1, O_2 \rangle$, and should design the TM so it ends with its tape head reading the first character of $\langle O_1 \rangle$. The input will be surrounded by infinitely many blanks.

   For example, given this initial configuration:




   The TM should end in this configuration:



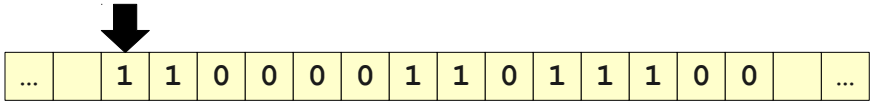

   You can assume that the encoding is properly formatted and can have your TM behave however you'd like if this isn't the case.

   Please provide a description of your TM as discussed at the start of this problem set.

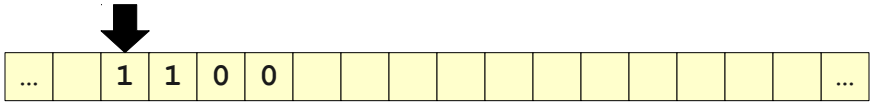
*(continued on the next page)*

ii. Draw the state transition diagram for a Turing machine that, given an encoding $\langle O_1, O_2 \rangle$ of two objects, ends with the string $\langle O_2 \rangle$ written on its tape, surrounded by infinitely many blanks. You can assume that the tape head begins reading the first character of $\langle O_1, O_2 \rangle$, and should design the TM so it ends with its tape head reading the first character of $\langle O_2 \rangle$. The input will be surrounded by infinitely many blanks.

For example, given this initial configuration:




The TM should end in this configuration:




You can assume that the encoding is properly formatted and can have your TM behave however you'd like if this isn't the case.

Please provide a description of your TM as discussed at the start of this problem set.

## Problem Three: Finding Flaws in Proofs (12 Points)

The **RE** languages are closed under union: if $L_1 \in$ **RE** and $L_2 \in$ **RE**, then $L_1 \cup L_2 \in$ **RE** as well. Below is an attempted proof that this is true:

*Theorem*: If $L_1 \in$ **RE** and $L_2 \in$ **RE**, then $L_1 \cup L_2 \in$ **RE**.

*Proof:* Consider any **RE** languages $L_1$ and $L_2$. Since $L_1 \in$ **RE** and $L_2 \in$ **RE**, we know that that there must exist recognizers $M_1$ and $M_2$ such that $\mathscr{L}(M_1) = L_1$ and $\mathscr{L}(M_2) = L_2$. Now, let $M$ be the following Turing machine, which we claim is a recognizer for $L_1 \cup L_2$:

> $M$ = "On input $w$:
> Run $M_1$ on $w$.
> If $M_1$ accepts $w$, accept.
> If $M_1$ rejects $w$:
> Run $M_2$ on $w$.
> If $M_2$ accepts $w$, accept.
> If $M_2$ rejects $w$, reject."

We claim that $\mathscr{L}(M) = L_1 \cup L_2$. To see this, note that by construction, $M$ accepts $w$ iff $M_1$ accepts $w$ or $M_1$ rejects $w$ and $M_2$ accepts $w$. Furthermore, note that $M_1$ accepts $w$ iff $w \in L_1$ and $M_2$ accepts $w$ iff $w \in L_2$. Therefore, $M$ accepts $w$ iff $w \in L_1$ or $w \in L_2$. Since $w \in L_1 \cup L_2$ iff $w \in L_1$ or $w \in L_2$, this means that $M$ accepts $w$ iff $w \in L_1 \cup L_2$. Thus $\mathscr{L}(M) = L_1 \cup L_2$, so $L_1 \cup L_2 \in$ **RE**, as required. ■

Although the theorem being proven is correct, the purported proof is incorrect because the constructed machine $M$ does not necessarily have language $L_1 \cup L_2$.

Give concrete examples of languages $L_1$ and $L_2$ and give high-level descriptions of machines $M_1$ and $M_2$ such that $\mathscr{L}(M_1) = L_1$ and $\mathscr{L}(M_2) = L_2$, but $\mathscr{L}(M) \neq L_1 \cup L_2$. You should explain why your languages and machines satisfy these properties, but you don't need to prove it. Then, determine the exact error in the proof that lets it justify (incorrectly) that $\mathscr{L}(M) = L_1 \cup L_2$.

## Problem Four: Nondeterministic Turing Machines (20 points)

Prove each of the following by giving a ***high-level description*** of an appropriate nondeterministic Turing machine and proving that the machine you describe has the appropriate language. Remember that for any NTM $M$, to prove that $\mathscr{L}(M) = L$, you should prove the following:

*For any string $w \in \Sigma^*$, $w \in L$ iff there is some series of choices $M$ can make such that $M$ accepts $w$.*

Notice that this statement is a biconditional.

In order to receive credit for your answers, your solutions *must* use nondeterminism as a key part of their operation, and you must write a proof of correctness.

i. Prove that the **RE** languages are closed under union. That is, if $L_1 \in$ **RE** and $L_2 \in$ **RE**, then $L_1 \cup L_2 \in$ **RE**.

ii. Prove that the **RE** languages are closed under concatenation. That is, if $L_1 \in$ **RE** and $L_2 \in$ **RE**, then $L_1L_2 \in$ **RE** as well.

***We will cover the material necessary to solve the remaining problems in Wednesday's lecture.***

## Problem Five: R and RE Languages (24 Points)

We have covered a lot of terminology and concepts in the past few days pertaining to Turing machines and **R** and **RE** languages. These problems are designed to explore some of the nuances of how Turing machines, languages, decidability, and recognizability all relate to one another. Please don't hesitate to ask if you're having trouble answering these questions – we hope that by working through them, you'll get a much better understanding of key computability concepts.

i. Give a high-level description of a TM $M$ such that $\mathscr{L}(M) \in$ **R**, but $M$ is not a decider. This shows that just because a TM's language is decidable, it's not necessarily the case that the TM itself must be a decider.

ii. Only *languages* can be decidable or recognizable; there's no such thing as an "undecidable string" or "unrecognizable string." Explain why there is no string $w$ such that any language that contains $w$ is undecidable and no string $w$ such that any language that contains $w$ is unrecognizable (a short paragraph should be sufficient). This result is important – the reason that languages become undecidable or unrecognizable is that there is no TM that can always give back the correct answer for *every* string in the language, not because there is some "bad string" that makes the language undecidable or unrecognizable.

iii. There is exactly one language $L \in$ **RE** such that every TM $M$ with $\mathscr{L}(M) = L$ must be a decider. What language is it? Briefly justify why your choice of $L$ has the property that every TM for $L$ is a decider, then prove that any other **RE** language has at least one TM for it that isn't a decider.

iv. Prove that for every **RE** language $L$, there is a decider $M^+$ that accepts every string in $L$ and a decider $M^-$ that rejects every string not in $L$. Explain why this result doesn't prove that **R** = **RE**.

v. Show that there is a TM $M$ with the following properties: $\mathscr{L}(M)$ is an infinite subset of $A_{TM}$, but $M$ is a decider. That is: $M$ accepts infinitely many strings of the form $\langle N, w \rangle$, where $N$ is a TM that accepts string $w$, yet the machine $M$ is a decider. Prove that your machine has the required properties. This shows that even though $A_{TM}$ is undecidable, it is still possible to build a TM that will decide $A_{TM}$ for infinitely many inputs.

## Problem Six: Why Decidability and Recognizability? (24 Points)

There are two classes of languages associated with Turing machines – the **RE** languages, which can be recognized by a Turing machine, and the **R** languages, which can be decided by a Turing machine.

Why didn't we talk about a model of computation that accepted just the **R** languages and nothing else? After all, having such a model of computation would be useful – if we could reason about automata that just accept recursive languages, it would be much easier to see what problems are and are not decidable.

It turns out, interestingly, that there is no class of automata with this property, and in fact the only way to build automata that can decide all recursive languages is to also have those automata also accept some languages that are **RE** but not **R**. This problem explores why.

Suppose, for the sake of contradiction, that there is a type of automaton called a **deciding machine** (or DM for short) that has the computational power to decide precisely the **R** languages. That is, $L \in \mathbf{R}$ iff there is a DM that decides $L$.

We will make the following (reasonable) assumptions about deciding machines:

- Any recursive language is accepted by some DM, and each DM accepts a recursive language.
- Since DMs accept precisely the recursive languages, all DMs halt on all inputs. That is, all DMs are deciders.
- Since deciding machines are a type of automaton, each DM is finite and can be encoded as a string. For any DM $D$, we will let the encoding of $D$ be represented by $\langle D \rangle$.
- DMs are an effective model of computation. Thus the Church-Turing thesis says that the Turing machine is at least as powerful as a DM. Thus there is some Turing machine $U_D$ that takes as input a description of a DM $D$ and some string $w$, then accepts if $D$ accepts $w$ and rejects if $D$ rejects $w$. Note that $U_D$ can never loop infinitely, because $D$ is a deciding machine and always eventually accepts or rejects. More specifically, $U_D$ is the decider "On input $\langle D, w \rangle$, simulate the execution of $D$ on $w$. If $D$ accepts $w$, accept. If $D$ rejects $w$, reject."

Unfortunately, these four properties are impossible to satisfy simultaneously.

i. Consider the language $REJECT_{DM} = \{ \langle D \rangle \mid D \text{ is a DM that rejects } \langle D \rangle \}$. Prove that $REJECT_{DM}$ is decidable.

ii. Prove that there is no DM that decides $REJECT_{DM}$.

Your result from (ii) allows us to prove that there is no class of automaton like the DM that decides precisely the **R** languages. If one were to exist, then it should be able to decide all of the **R** languages, including $REJECT_{DM}$. However, there is no DM that accepts the decidable language $REJECT_{DM}$. This means that one of our assumptions must have been violated, so at least one of the following must be true:

- DMs do not accept precisely the **R** languages, or
- DMs are not deciders, or
- DMs cannot be encoded as strings (meaning they lack finite descriptions), or
- DMs cannot be simulated by a TM (they are not effective models of computation)

Thus there is no effective model of computation that decides just the recursive languages.

## Problem Seven: 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 following questions. 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.

i. How hard did you find this problem set? How long did it take you to finish? Does that seem unreasonably difficult or time-consuming for a five-unit class?

ii. Did you attend a recitation section this week? If so, did you find it useful?

iii. How is the pace of this course so far? Too slow? Too fast? Just right?

iv. Is there anything in particular we could do better? Is there anything in particular that you think we're doing well?

## Submission instructions

There are three ways to submit this assignment:

1. Hand in a physical copy of your answers at the start of class. This is probably the easiest way to submit if you are on campus.
2. Submit a physical copy of your answers in the filing cabinet in the open space near the handout hangout in the Gates building. If you haven't been there before, it's right inside the entrance labeled "Stanford Engineering Venture Fund Laboratories." There will be a clearly-labeled filing cabinet where you can submit your solutions.
3. Send an email with an electronic copy of your answers to the submission mailing list (cs103-win1213-submissions@lists.stanford.edu) with the string "[PS7]" somewhere in the subject line. If you do submit electronically, please submit your assignment as a single PDF if at all possible. Sending multiple image files makes it much harder to print out and grade your submission.

## Extra Credit Problem: Unrestricted Grammars (5 Points)

An *unrestricted grammar* is a substantial generalization of context-free grammars in which arbitrary strings can appear on the left-hand side of a production, not just nonterminals. This allows unrestricted grammars to replace arbitrary substrings of a sentential form with new substrings.

For example, the following unrestricted grammar generates the language $\{ \mathtt{0}^n\mathtt{1}^n\mathtt{2}^n \mid n \in \mathbb{N} \}$:

$$\begin{aligned}
T &\to S \mid \varepsilon \\
S &\to \mathtt{0}SX \\
X &\to A\mathtt{2} \\
\mathtt{2}A &\to A\mathtt{2} \\
SA &\to \mathtt{1} \\
\mathtt{1}A &\to \mathtt{11}
\end{aligned}$$

One possible derivation of `000111222` is shown here:

$$\begin{aligned}
&\underline{T} && \text{(Apply } T \to S\text{)} \\
\Rightarrow\ &\underline{S} && \text{(Apply } S \to \mathtt{0}SX\text{)} \\
\Rightarrow\ &\mathtt{0}\underline{S}X && \text{(Apply } S \to \mathtt{0}SX\text{)} \\
\Rightarrow\ &\mathtt{00}\underline{S}XX && \text{(Apply } S \to \mathtt{0}SX\text{)} \\
\Rightarrow\ &\mathtt{000}S\underline{X}XX && \text{(Apply } X \to A\mathtt{2}\text{)} \\
\Rightarrow\ &\mathtt{000}SA\mathtt{2}\underline{X}X && \text{(Apply } X \to A\mathtt{2}\text{)} \\
\Rightarrow\ &\mathtt{000}SA\mathtt{2}A\mathtt{2}\underline{X} && \text{(Apply } X \to A\mathtt{2}\text{)} \\
\Rightarrow\ &\mathtt{000}SA\mathtt{2}A\underline{\mathtt{2}A}\mathtt{2} && \text{(Apply } \mathtt{2}A \to A\mathtt{2}\text{)} \\
\Rightarrow\ &\mathtt{000}SA\underline{\mathtt{2}A}A\mathtt{22} && \text{(Apply } \mathtt{2}A \to A\mathtt{2}\text{)} \\
\Rightarrow\ &\mathtt{000}SAA\underline{\mathtt{2}A}\mathtt{22} && \text{(Apply } \mathtt{2}A \to A\mathtt{2}\text{)} \\
\Rightarrow\ &\mathtt{000}\underline{SA}AA\mathtt{222} && \text{(Apply } SA \to \mathtt{1}\text{)} \\
\Rightarrow\ &\mathtt{000}\underline{\mathtt{1}A}A\mathtt{222} && \text{(Apply } \mathtt{1}A \to \mathtt{11}\text{)} \\
\Rightarrow\ &\mathtt{00011}\underline{\mathtt{1}A}\mathtt{222} && \text{(Apply } \mathtt{1}A \to \mathtt{11}\text{)} \\
\Rightarrow\ &\mathtt{000111222}
\end{aligned}$$

Given an unrestricted grammar $G$, we define $\mathscr{L}(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \}$.

Prove that a language $L$ is **RE** iff there is an unrestricted grammar $G$ where $\mathscr{L}(G) = L$. This gives a formalism for describing the **RE** languages through generation, just as the Turing machine is a formalism for describing the **RE** languages through recognition.

You might find the following fact useful: for any TM $M$, there is a TM $M'$ such that $\mathscr{L}(M) = \mathscr{L}(M')$, but any time $M'$ accepts, it always accepts with the tape containing just a copy of the input string surrounded by infinitely many blanks.

***You are strongly encouraged to work on this problem – it has a beautiful solution and can be completed using only concepts from last Friday's lecture.***