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!

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, November 18th at 2:15 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.

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.

Unless stated otherwise, any TM you design should be a **deterministic** TM.

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

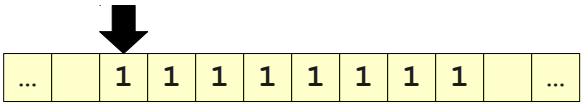
Problem One: The Collatz Conjecture (24 Points)

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

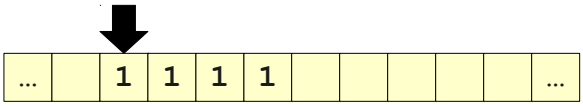
- If $n = 1$, stop.
- If n is even, set $n = n / 2$.
- If n is odd, set $n = 3n + 1$.
- Repeat.

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:

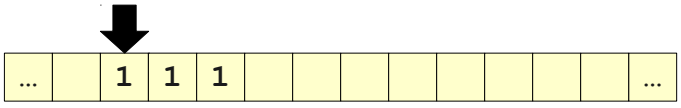


The TM would end with this configuration:

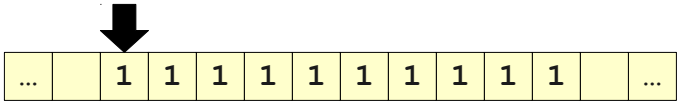


You can assume that there are an even number of **1**s on the tape at startup and can have your TM behave however you'd like if this isn't the case. Please provide a description of your TM as discussed at the start of this problem set. *(For reference, our solution has 7 states. If you have significantly more than this, you might want to change your approach.)*

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. *(For reference, our solution has 9 states. If you have significantly more than this, you might want to change your approach.)*

(continued on the next page)

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. *(For reference, our solution has 8 states. If you have significantly more than this, you might want to change your approach.)*

Problem Two: Manipulating Encodings (16 Points)

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

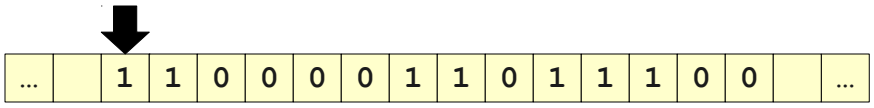
- doubling each character in $\langle O_1 \rangle$ (i.e. `0` becomes `00` and `1` becomes `11`),
- then writing out the string `01` as a delimiter, and finally
- writing out the description of $\langle O_2 \rangle$ unchanged.

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

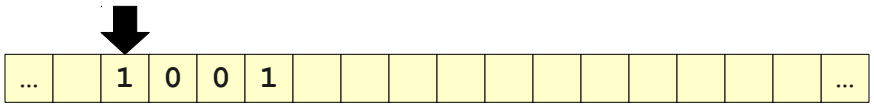
In order for this representation to be useful, Turing machines need to be to extract the first and second part of an encoded pair. This problem asks you to design TMs that do precisely these tasks.

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:

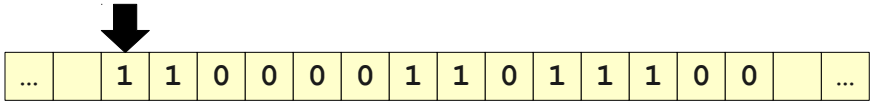


   You can assume that the encoding is properly formatted and can have your TM behave however you'd like if this isn't the case. Please provide a description of your TM as discussed at the start of this problem set. *(For reference, our solution has 9 states. If you have significantly more than this, you might want to change your approach.)*

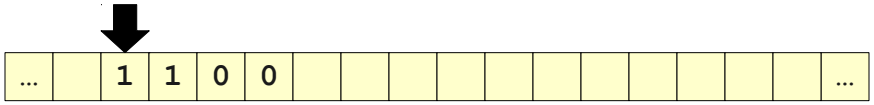
(continued on the next page)

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. *(For reference, our solution has 2 states.)*

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 TMs 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 TM 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 we see $\mathscr{L}(M) = L_1 \cup L_2$, 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." Prove that there is no string w where every language containing w is undecidable and no string w where every language containing w is unrecognizable. 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. Prove that for every 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 every language is in **R**.

iv. Give a high-level description of 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 the **R** languages, it would be 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 **R** 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 **hypothetical machine** (or HM for short) that has the computational power to decide precisely the **R** languages. That is, $L \in \mathbf{R}$ iff there is a HM that decides L. We can't assume much about how HMs work – perhaps they use an infinite tape and a finite-state control, or maybe they use a combination of stacks and queues, or perhaps they just use magic widgets – so we can't design concrete HMs. However, we will make the following (reasonable) assumptions about HMs:

- Any **R** language is accepted by some HM, and each HM accepts an **R** language.
- Since HMs accept precisely the **R** languages, all HMs halt on all inputs. That is, once started, an HM will always eventually terminate with an answer.
- Since HMs are a type of automaton, each HM is finite and can be encoded as a string. For any HM H, we will let the encoding of H be represented by $\langle H \rangle$.
- HMs are an effective model of computation. Thus the Church-Turing thesis says that the Turing machine is at least as powerful as a HM. Thus there is some Turing machine U_{HM} that takes as input a description of a HM H and some string w, then accepts if H accepts w and rejects if H rejects w. Note that U_{HM} can never loop infinitely, because H is a hypothetical machine and always eventually accepts or rejects. More specifically, U_{HM} is the decider "On input $\langle H, w \rangle$, where H is an HM and w is a string, simulate the execution of H on w. If H accepts w, accept. If H rejects w, reject."

Unfortunately, these four properties are impossible to satisfy simultaneously.

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

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

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

- HMs do not accept precisely the **R** languages, or
- HMs do not halt on all inputs, or
- HMs cannot be encoded as strings (meaning they lack finite descriptions), or
- HMs 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 **R** 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 fill out the feedback questions for this problem set, available at https://docs.google.com/forms/d/1mKJMlID3WHfD9tq-N0R41iOf3oDdyZ4amXq0Hke4T4k/viewform.We will give you full credit for any answers you write (as long as you answer all the parts of the question!), though we'd appreciate it if you were honest in your feedback.

Extra Credit Problem: TMs and Regular Languages (5 Points Extra Credit)

We can measure the amount of time that a Turing machine takes to run on some string w by counting the total number of transitions the Turing machine makes when running on w. Let's denote the number of transitions that M takes when run on string w by $T(M, w)$

Let M be a Turing machine with input alphabet Σ. Prove that if there is some fixed constant n such that $T(M, w) \leq n$ for any $w \in \Sigma^*$, then $\mathscr{L}(M)$ is regular. (Intuitively, this means that if there's some fixed upper bound to the amount of time that a TM takes to run, then its language must be regular.)