Problem Set 8

This problem explores Turing machines, properties of the **RE** and **R** languages, and the limits of decidability. 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, ask questions on Piazza, or send us emails if you have any questions. We'd be happy to help out.

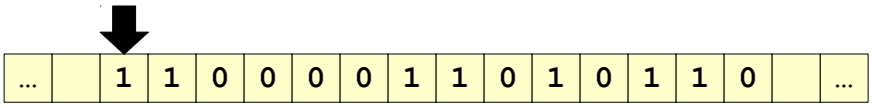
Good luck, and have fun!

Due Friday, November 20th at the start of lecture

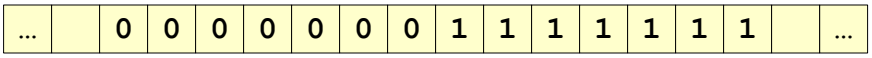
Problem One: Binary Sorting (3 Points)

Your task in this question is to design a TM that can sort a sequence of `0`s and `1`s into ascending order. Specifically, your TM will be given as input a string of `0`s and `1`s surrounded by infinitely many blanks and should sort them so that all `0`s come before all `1`s. Your TM should end by entering an accepting state after rewriting the contents of the tape so that the `0`s and `1`s are sorted and the string is surrounded by infinitely many blanks. The tape head can end anywhere on the tape.

For example, given this initial configuration:



The TM should end in an accepting state with these tape contents:



Please use our provided TM editor to design, develop, test, and submit your answer to this question. *(For reference, our solution has fewer than 10 states. If you have significantly more than this, you might want to change your approach.)*

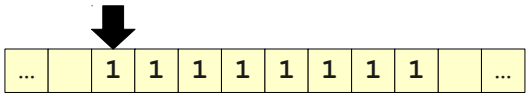
Problem Two: The Collatz Conjecture (6 Points)

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:

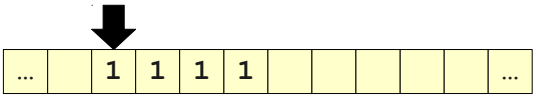
- If $n = 1$, stop.
- If n is even, set $n = n / 2$.
- If n is odd, set $n = 3n + 1$.
- Repeat.

In lecture, we claimed that it was possible to build a TM for the language $L = \{ 1^n \mid n \geq 1$ and the hailstone sequence terminates for $n \}$ over the alphabet $\Sigma = \{1\}$. 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. Design a TM subroutine that, given a tape holding 1^{2n} surrounded by infinitely many blanks, ends with 1^n written on the 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:

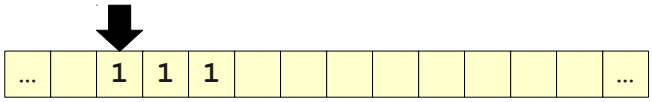


The TM would end with this configuration:

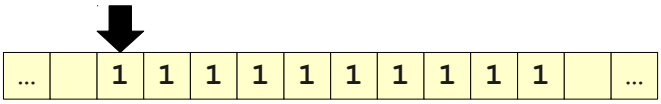


You can assume that there are an even number of 1s on the tape at startup and can have your TM behave however you'd like if this isn't the case. Please use our provided TM editor to design, develop, test, and submit your answer to this question. Since our TM tool doesn't directly support subroutines, just have your machine accept when it's done. *(For reference, our solution has fewer than 10 states. If you have significantly more than this, you might want to change your approach.)*

ii. Design a TM subroutine that, given a tape holding 1^n surrounded by infinitely many blanks, ends with 1^{3n+1} written on the 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 use our provided TM editor to design, develop, test, and submit your answer to this question. Since our TM tool doesn't directly support subroutines, just have your machine accept when it's done. *(For reference, our solution has fewer than 10 states. If you have significantly more than this, you might want to change your approach.)*

iii. Using your TMs from parts (i) and (ii) as subroutines, draw the state transition diagram for a Turing machine M that recognizes L. Our TM tool is configured for this problem so that you can use our reference solutions for parts (i) and (ii) as subroutines in your solution. To do so, follow these directions:

1. Create states named `half`, `half_`, `trip`, and `trip_`.
2. To execute the subroutine that converts $\texttt{1}^{2n}$ into $\texttt{1}^{n}$, have your machine transition into the state named `half`. When that subroutine finishes, the TM will automagically transition into the state labeled `half_`. You do not need to – and should not – define any transitions into `half_` or out of `half`.
3. To execute the subroutine that converts $\texttt{1}^{n}$ into $\texttt{1}^{3n+1}$, have your machine transition into the state named `trip`. When that subroutine finishes, the TM will automagically transition into the state labeled `trip_`. You do not need to – and should not – define any transitions into `trip_` or out of `trip`.

Please use our provided TM editor to design, develop, test, and submit your answer to this question. *(For reference, our solution has fewer than 15 states. If you have significantly more than this, you might want to change your approach.)*

Problem Three: What Does it Mean to Solve a Problem? (5 Points)

Let L be a language over Σ and M be a TM with input alphabet Σ. Below are three properties that may hold for M:

1. M is a decider (that is, M halts on all inputs.)
2. For any string $w \in \Sigma^*$, if M accepts w, then $w \in L$.
3. For any string $w \in \Sigma^*$, if M rejects w, then $w \notin L$.

At some level, for a TM to claim to solve a problem, it should have at least some of these properties. Interestingly, though, just having two of these properties doesn't say much.

i. Prove that if L is any language over Σ, then there is a TM M that satisfies properties (1) and (2) with respect to L.

ii. Prove that if L is any language over Σ, then there is a TM M that satisfies properties (1) and (3) with respect to L.

iii. Prove that if L is any language over Σ, then there is a TM M that satisfies properties (2) and (3) with respect to L.

iv. Suppose that L is a language over Σ for which there is a TM M that satisfies properties (1), (2), and (3). What can you say about L?

Problem Four: R and RE Languages (4 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.

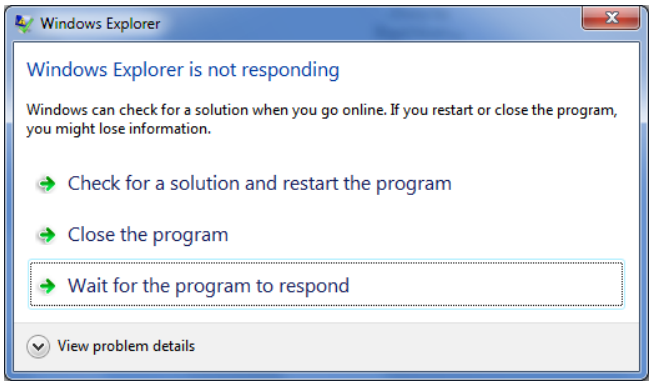
i. Give a high-level description of a TM M such that $\mathscr{L}(M) \in \mathbf{R}$, but M is not a decider (you can draw a concrete example of TM or give pseudocode for a program). Briefly justify your answer. This shows that just because a TM's language is decidable, it's not necessarily the case that the TM itself must be a decider.

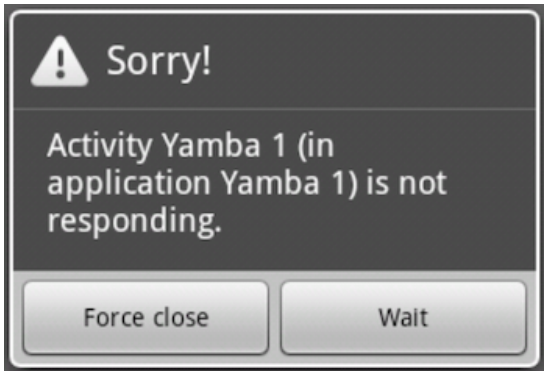
ii. Only *languages* can be decidable or recognizable; there's no such thing as an "undecidable string" or "unrecognizable string." Prove that for every string w, there's an **R** language containing w and an **RE** language containing w.

Depending on how quickly we cover the material in Friday's lecture, you may need material from Monday's lecture in order to solve the remaining problems on this problem set.

Problem Five: This Program is Not Responding (1 Point)

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.

Why does the operating system have to display a message like this? Briefly justify your answer.

Problem Six: Password Checking (8 Points)

If you're an undergraduate here, you've probably noticed that the dorm staff have master keys they can use to unlock any of the doors in the residences. That way, if you ever lock yourself out of your room, you can, sheepishly, ask for help back in. (Not that I've ever done that or anything.)

Compare this to a password system. When you log onto a website with a password, you have the presumption that your password is the only possible password that will log you in. There shouldn't be a "master key" password that can unlock any account, since that would be a huge security vulnerability. But how could you tell? If you had the source code to the password checking system, could you figure out whether your password was the only password that would grant you access to the system?

Let's frame this question in terms of Turing machines. If we wanted to build a TM password checker, "entering your password" would correspond to starting up the TM on some string, and "gaining access" would mean that the TM accepts your string.

Let's suppose that your password is the string "iheartquokkas." A TM that would work as a valid password checker would be a TM M where $\mathscr{L}(M) = \{\text{iheartquokkas}\}$: the TM accepts your string, and it doesn't accept anything else. Given a TM, is there some way you could tell whether the TM was a valid password checker?

Connsider the following language L:

$$L = \{ \langle M \rangle \mid M \text{ is a TM and } \mathscr{L}(M) = \{\text{iheartquokkas}\} \}$$

Your task in this problem is to prove that L is undecidable (that is, $L \notin \mathbf{R}$). This means that there's no algorithm that can mechanically check whether a TM is suitable as a password checker. Rather than dropping you headfirst into this problem, we've split this problem apart into a few smaller pieces.

Let's suppose for the sake of contradiction that $L \in \mathbf{R}$. That means that there is some function

```
bool isPasswordChecker(string program)
```

with the following properties:

- If `program` is the source of a program that accepts just the string "iheartquokkas," then calling `isPasswordChecker(program)` will return `true`.
- If `program` is not the source of a program that accepts just the string "iheartquokkas," then calling `isPasswordChecker(program)` will return `false`.

We can try to build a self-referential program that uses the `isPasswordChecker` function to obtain a contradiction. Here's a first try:

```
bool isPasswordChecker(string program) {
      /* … some implementation … */
}
int main() {
      string me = mySource();
      string input = getInput();

      if (isPasswordChecker(me)) {
            reject();
      } else {
            accept();
      }
}
```

This code is, essentially, a minimally-modified version of the self-referential program we used to get a contradiction for the language A_{TM}.

i. Suppose that this program is a valid password checker. Briefly explain why running this program leads to a contradiction.

ii. Suppose that this program is ***not*** a valid password checker. Briefly explain why running this program does ***not*** lead to a contradiction.

Ultimately, the goal of building a self-referential program here is to have the program cause a contradiction regardless of whether or not it's a password checker. As you've seen in part (ii), this particular program does not cause a contradiction if it isn't a password checker. Consequently, if we want to prove that $L \notin \mathbf{R}$, we need to modify it so that it leads to a contradiction in the case where it is a password checker.

iii. Modify the above code so that it causes a contradiction regardless of whether it's a password checker. Then, briefly explain why your modified program is correct. (No formal proof is necessary here; you're going to do that in the next step.)

iv. Formalize your argument in part (iii) by proving that $L \notin \mathbf{R}$. Use the proof that $A_{TM} \notin \mathbf{R}$ as a template for your proof.

Problem Seven: Self-Reference and RE (4 Points)

In lecture, we saw that A_{TM} is undecidable, but it is recognizable. From a programming perspective, this means that it's possible to write a method

```
bool willAccept(string program, string input)
```

that accepts as input a program and an input. The method then has the following guarantees:

- If the program accepts the input, this method *must* return true.
- If the program does not accept the input, this method *may* return false, or it may go into an infinite loop and never return.

Now, consider the following program:

```
int main() {
      string me = mySource();
      string input = getInput();

      if (willAccept(me, input)) {
            reject();
      } else {
            accept();
      }
}
```

Prove that this program must loop infinitely on all inputs. *(Hint: proceed by contradiction.)*

Extra Credit Problem: Quine Relays (1 Point Extra Credit)

In either C, C++, Python, or Java, write four different programs with the following properties:

- Running the first program prints the complete source code of the second program.
- Running the second program prints the complete source code of the third program.
- Running the third program prints the complete source code of the fourth program.
- Running the fourth program prints the complete source code of the first program.
- None of the programs perform any kind of file reading.

In other words, we'd like a collection of four different programs, each of which prints the complete source of the next one in the sequence, wrapping back around at the end.

Please submit your programs by emailing the staff list (cs103-aut1516-staff@lists.stanford.edu) with the subject "PS8 EC" and attaching your source files as a .zip archive. (We ask that you submit this way because we need to be able to independently verify that your programs work as expected.)