Problem Set 7

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.

This problem set has 34 possible points. It is weighted at 6% of your total grade.

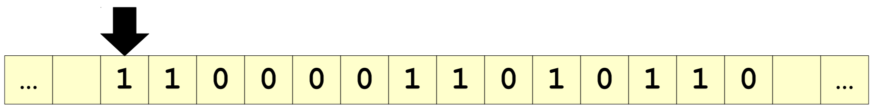
Good luck, and have fun!

Due Wednesday, May 27th at the start of lecture

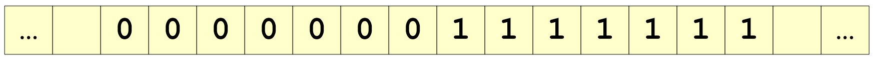
Problem One: Binary Sorting (3 Points)

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.)*

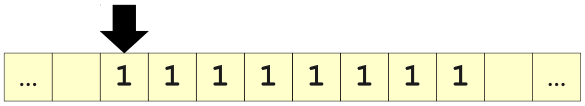
Problem Two: The Collatz Conjecture (6 Points)

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:

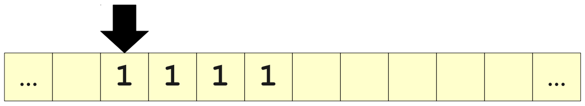
- If $n = 1$, stop.
- If n is even, set $n = n / 2$.
- If n is odd, set $n = 3n + 1$.
- Repeat.

In lecture, we claimed that it was possible to build a TM for the language $L = \{ \mathtt{1}^n \mid \text{the hailstone sequence terminates for } n \}$ over the alphabet $\Sigma = \{\mathtt{1}\}$. 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 $\mathtt{1}^{2n}$ surrounded by infinitely many blanks, ends with $\mathtt{1}^n$ 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:

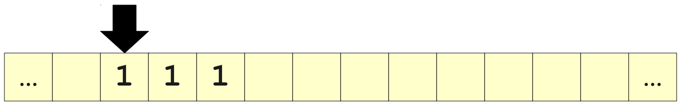


The TM would end with this configuration:

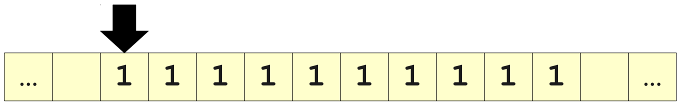


You can assume that there are an even number of `1`s on the tape at startup and can have your TM behave however you'd like if this isn't the case. Please use our provided TM editor to design, develop, test, and submit your answer to this question. *(For reference, our solution has fewer than 10 states.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

ii. Design a TM subroutine that, given a tape holding $\mathtt{1}^n$ surrounded by infinitely many blanks, ends with $\mathtt{1}^{3n+1}$ 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. *(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. 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. (Check our TM for checking whether a number is composite as a reference.) 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 Three: What Does it Mean to Solve a Problem? (6 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. 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.

Problem Five: Password Checking (5 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 $p \in \Sigma^*$ be your password. A TM that would work as a valid password checker would be a TM M where $\mathscr{L}(M) = \{p\}$; 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? Let $p \in \Sigma^*$ be your password and consider the following language:

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

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.

i. Suppose there is a function

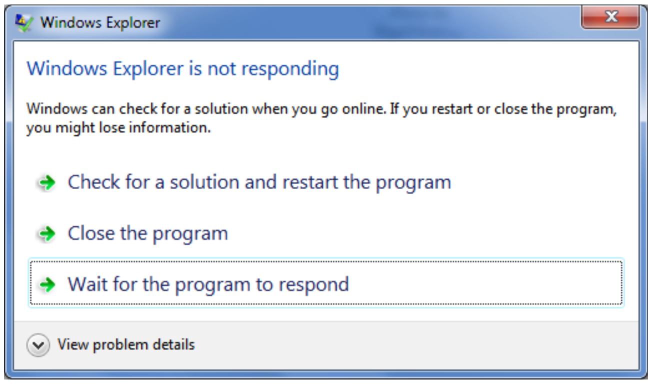
```
bool isPasswordChecker(string program)
```

that accepts as input a program and returns whether or not that program only accepts the string p. Using the programs from lecture as a template, write the pseudocode for a self-referential program that uses the `isPasswordChecker` method to obtain a contradiction. No justification is necessary; you'll do that in the next step.

ii. Now, write a formal proof that L is not decidable. Use the proofs from lecture as a template – use your pseudocode from above to write the high-level description of a self-referential Turing machine that uses a decider for L as a subroutine, then obtain a contradiction from that machine.

Problem Six: 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:



Windows Explorer

Windows Explorer is not responding

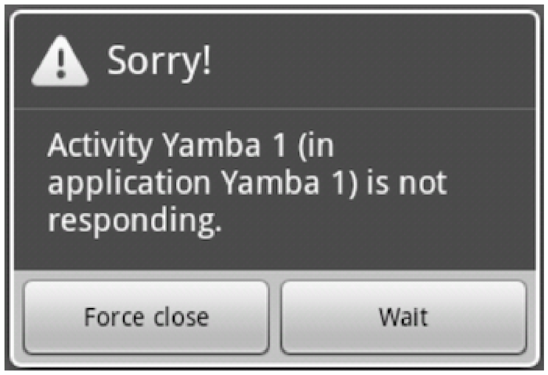
Windows can check for a solution when you go online. If you restart or close the program, you might lose information.

Check for a solution and restart the program

Close the program

Wait for the program to respond

View problem details



Sorry!

Activity Yamba 1 (in application Yamba 1) is not responding.

Force close

Wait

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 Seven: Self-Reference and RE (4 Points)

In lecture, we saw that A_{TM} is undecidable, but 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.)*

Problem Eight: Closure Properties of R (5 Points)

This question explores various closure properties of **R**. Because **R** corresponds to decidable problems, languages in **R** are precisely the languages for which you can write a method

```
bool inL(string w)
```

such that

- for any string $w \in L$, calling `inL(`w`)` returns true.
- for any string $w \notin L$, calling `inL(`w`)` returns false.

This means that we can reason about closure properties of the decidable languages by writing actual pieces of code.

i. Let L_1 and L_2 be decidable languages over the same alphabet Σ. Prove that $L_1 \cup L_2$ is also decidable. To do so, suppose that you have methods `inL1` and `inL2` matching the above conditions, then show how to write a method `inL1uL2` with the appropriate properties. Then, write a short proof explaining why your method has the required properties.

ii. Using a proof along the lines of part (i) of this problem, prove that **R** is closed under concatenation.

iii. Prove that **R** is closed under symmetric difference. This might seem like a weird thing to prove, but you'll need this result on the final problem set.

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

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

- Running the first program prints the second program.
- Running the second program prints the third program.
- Running the third program prints the fourth program.
- Running the fourth program prints 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 next one in the sequence, wrapping back around at the end. Please submit your programs by emailing the staff list (cs103-spr1415-staff@lists.stanford.edu) with the subject "PS7 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.)