Problem Set 8

What problems are beyond our capacity to solve? Why are they so hard? And why is anything that we've discussed this quarter at all practically relevant? In this problem set, you'll explore the absolute limits of computing power.

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 43 possible points. It is weighted at 6% of your total grade. This is slightly higher than usual because this problem set also contains a quick cumulative review problem at the end.

Good luck, and have fun!

Due Wednesday, June 3 at the start of lecture.

No late submissions accepted. No late days may be used.

Problem One: Closure Properties of RE (5 Points)

This question explores various closure properties of **RE**. Because **RE** corresponds to recognizable problems, languages in **RE** 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`)` either returns false or loops infinitely without returning.

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 recognizable languages over the same alphabet Σ. Prove that $L_1 \cap L_2$ is also recognizable. To do so, suppose that you have methods `inL1` and `inL2` matching the above conditions, then show how to write a method `inL1nL2` with the appropriate properties. Then, write a short proof explaining why your method has the required properties.

The **RE** languages are also closed under union. Let's imagine that we have two recognizable languages L_1 and L_2 that are in **RE** and that we have methods `inL1` and `inL2` matching the above properties. Below is an ***incorrect*** construction that purportedly is a recognizer for $L_1 \cup L_2$:

```
bool inL1uL2(string w) {
      return inL1(w) || inL2(w);
}
```

ii. Give concrete examples of languages L_1 and L_2 and implementations of methods `inL1` and `inL2` such that the above piece of code is not a recognizer for $L_1 \cup L_2$. Justify your answer.

To show closure under union, it's easier to use the fact that the **RE** languages are precisely the verifiable languages. A language L is an **RE** language precisely if it's possible to write a method named

```
bool imConvincedIsInL(string w, string c)
```

with the following properties:

- This function always returns a value.
- If $w \in L$, then there is some choice of c where calling `imConvincedIsInL(w, c)` returns true.
- If $w \notin L$, then calling `imConvincedIsInL(w, c)` returns false for all choices of c.

Take a few minutes to make sure you understand why these properties mean that the method is a verifier for the language L.

iii. Let L_1 and L_2 be recognizable languages over the same alphabet Σ. Using the verifier definition of **RE**, prove that $L_1 \cup L_2$ is also recognizable. To do so, suppose that you have methods `imConvincedIsInL1` and `imConvincedIsInL2` matching the above conditions, then show how to write a method `imConvincedIsInL1uL2` with the appropriate properties. Then, write a short proof explaining why your method has the required properties.

Problem Two: Password Checking (6 Points)

Let $p \in \Sigma^*$ be a string and consider the following language:

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

In the previous problem set, you proved that $L \notin \mathbf{R}$. In this problem, you'll prove that $L \notin \mathbf{RE}$.

Let's suppose for the sake of contradiction that $L \in \mathbf{RE}$. This means that there must be some verifier for the language L. In software, we can express that verifier as a function

```
bool imConvincedIsPasswordChecker(string program, string c)
```

with the following properties:

- This function always returns a value.
- If `program` is a password checker, then there is some choice of c where calling `imConvincedIsPasswordChecker(program, c)` returns true.
- If `program` is not a password checker, then calling `imConvincedIsPasswordChecker(program, c)` returns false for all choices of c.

We can now try to write a self-referential program that uses the above function to cause a contradiction. Here's a first attempt:

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

      for (int i = 0 to infinity) {
            for (each string c of length i) {
                  if (imConvincedIsPasswordChecker(me, c)) {
                        accept();
                  }
            }
      }
}
```

This code is, essentially, a minimally-modified version of the self-referential program we used to get a contradiction for the language *LOOP*.

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.

iii. Modify the code above to address the deficiency you identified in part (ii). Then, briefly explain why your modified program leads to a contradiction regardless of whether it's a valid password checker.

iv. Formalize your argument in part (iii) by proving that $L \notin \mathbf{RE}$. Use the proof that *LOOP* is not an **RE** language as a template.

Problem Three: Equivalent TMs (4 Points)

If you've taken CS106A, CS106B, or CS107, you've probably noticed that we have a lot of section leaders and TAs on staff. This is partially so that we can provide lots of one-on-one support and assistance in those courses, but part of it is also due to the fact that it's really hard to grade programming assignments. This question explores why.

When teaching a programming class, it would be really nice if we could fully autograde student programming submissions. Ideally, we'd like to be able to write our own reference solution to one of the programming problems, then check, for each student submission, whether that submission is in some way "equivalent" to our reference solution. If it is, then the submission must be correct, and if it isn't, then the submission must be incorrect.

Let's reformulate this as an equivalent problem about Turing machines. Suppose we have a student-submitted TM M_1 and a reference TM M_2. We'd like to be able to check whether these TMs have the same languages, that is, whether $\mathscr{L}(M_1) = \mathscr{L}(M_2)$. (This isn't perfectly analogous to our original problem, but it's a close enough match.) We'd like to see whether we can write a TM that can check whether these two TMs have the same language, and, if not, at least whether we can write a TM that checks whether these two TMs have different languages.

Consider the following language EQ_{TM}:

$$EQ_{TM} = \{ \langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } \mathscr{L}(M_1) = \mathscr{L}(M_2) \}$$

In other words, EQ_{TM} is the set of all pairs of TMs that have the same language.

It turns out that this is a *frighteningly* hard problem to solve, and in this question you'll see why.

i. Suppose there is a function

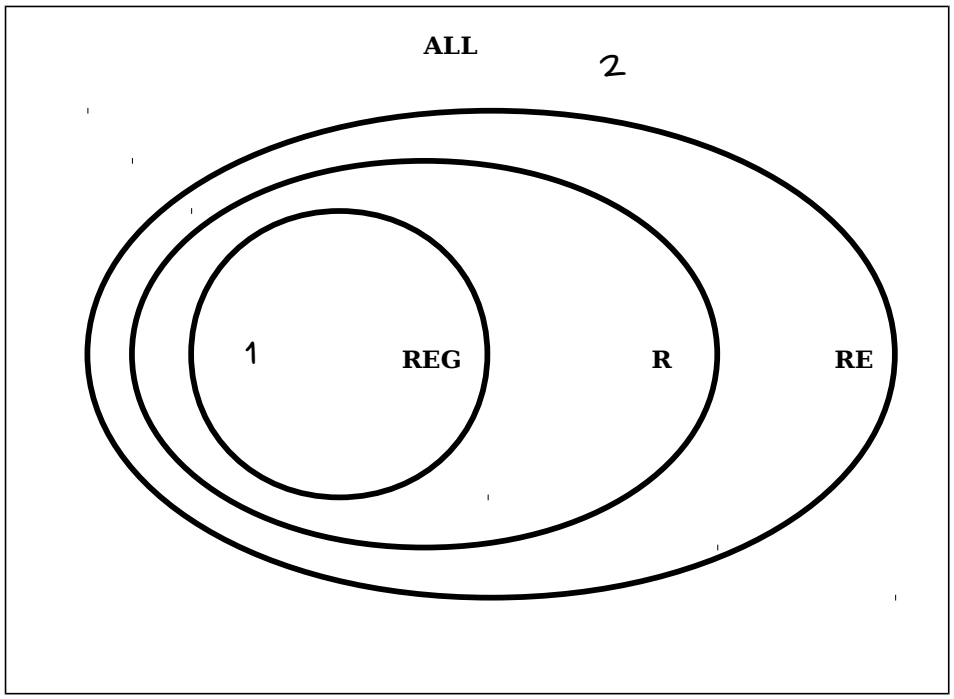
```
bool imConvincedAreEqual(string p1, string p2, string certificate)
```

that acts as a verifier for EQ_{TM}. Design a self-referential program that uses this function to obtain a contradiction. As a hint, you might want to hard-code one of the arguments to the function.

ii. Using your program from part (i) as a template, write a formal proof that $EQ_{TM} \notin \mathbf{RE}$. (*A note:* if you have a copy of the Sipser textbook, in Chapter Five, Sipser proves that this language is not **RE** using mapping reductions. You're welcome to read over this proof if you'd like, but your answer to this question should use self-reference rather than mapping reductions.)

Problem Four: The Big Picture (10 Points)

Below is a Venn diagram showing the overlap of different classes of languages we've studied so far. We have also provided you a list of 12 numbered languages. For each of those languages, draw where in the Venn diagram that language belongs. As an example, we've indicated where Language 1 and Language 2 should go. No proofs or justifications are necessary – the purpose of this problem is to help you build a better intuition for what makes a language regular, **R**, **RE**, or none of these.



1. Σ^*
2. EQ_{TM} (defined earlier in this problem set.)
3. $\{ \mathtt{a}^n \mid n \in \mathbb{N} \}$
4. $\{ \mathtt{a}^n \mid n \in \mathbb{N}$ and is a multiple of 137 $\}$
5. $\{ \mathtt{a}^n \mid n \in \mathbb{N} \} \cup \{ \mathtt{a}^n \mid n \in \mathbb{N}$ and is a multiple of 137 $\}$
6. $\{ \mathtt{1}^n\mathtt{+}\mathtt{1}^m \stackrel{?}{=} \mathtt{1}^{n+m} \mid m, n \in \mathbb{N} \}$
7. $\{ \langle M \rangle \mid M$ is a Turing machine and $\mathscr{L}(M) = L_D \}$
8. $\{ \langle M \rangle \mid M$ is a Turing machine and $\mathscr{L}(M) = \varnothing \}$
9. $\{ \langle M, n \rangle \mid M$ is a TM, $n \in \mathbb{N}$, and M ***accepts*** all strings of length at most $n \}$
10. $\{ \langle M, n \rangle \mid M$ is a TM, $n \in \mathbb{N}$, and M ***rejects*** all strings of length at most $n \}$
11. $\{ \langle M, n \rangle \mid M$ is a TM, $n \in \mathbb{N}$, and M ***loops*** on all strings of length at most $n \}$
12. $\{ \langle M_1, M_2, M_3, w \rangle \mid M_1, M_2$, and M_3 are TMs, w is a string, and at least two of M_1, M_2, and M_3 accept $w. \}$

(We will cover the material necessary to solve the remaining problems on Friday.)

Problem Five: 4-Colorability (4 Points)

An undirected graph G is called 3-colorable if there exists a way to color each the nodes in G one of three colors such that no two nodes of the same color are connected by an edge. The following language consists of all graphs that are 3-colorable.

$$3COLOR = \{ \langle G \rangle \mid G \text{ is an undirected, 3-colorable graph} \}$$

This language is **NP**-complete. If we have time, we'll sketch a proof of this in Wednesday's lecture.

An undirected graph is called 4-colorable if there is a way to color each of the nodes in G one of four colors so that no two nodes of the same color are connected by an edge. We can formalize the 4-coloring problem as a language as follows:

$$4COLOR = \{ \langle G \rangle \mid G \text{ is an undirected, 4-colorable graph} \}$$

Note that every 3-colorable graph is also 4-colorable, but not all 4-colorable graphs are 3-colorable. In other words, 3-colorability is a stricter requirement than 4-colorability. However, it is still the case that *4COLOR* is **NP**-complete, and in this problem you will prove this result.

i. Briefly justify why $4COLOR \in \mathbf{NP}$. No formal proof is necessary.

ii. Prove that *4COLOR* is **NP**-hard by proving $3COLOR \leq_p 4COLOR$. That is, show how to take an arbitrary graph G and construct (in polynomial time) a graph G' such that graph G is 3-colorable if and only if graph G' is 4-colorable.

For simplicity, you do not need to formally prove that your reduction is correct and runs in polynomial time. Instead, briefly answer each of the following questions about your reduction (two or three sentences apiece should be sufficient):

1. If the original graph G is 3-colorable, why is your new graph G' 4-colorable?
2. If your new graph G' is 4-colorable, why is the original graph G 3-colorable?
3. Why can your reduction be computed in polynomial time?

As a reminder, the job of a reduction isn't to solve the problem – it's to *transform* the problem. Therefore, you don't need to worry about how exactly you'd determine whether G' is 4-colorable or not. You just need G' to be built so that G' is 4-colorable if and only if G is 3-colorable.

Also, remember that since *4COLOR* is a language of undirected graphs, the output of your reduction should just be a graph, not a graph with any nodes colored. You need to make the graph G' such that if there is *any* way to 4-color it, the original graph G is 3-colorable. You can't, for example, specify that certain nodes must be certain colors.

Problem Six: Resolving P $\stackrel{?}{=}$ NP (8 Points)

This problem explores the question

What would it take to prove whether P = NP?

For each statement below, decide whether the statement would definitely prove **P = NP**, definitely prove **P ≠ NP**, or would do neither. Write "**P = NP**," "**P ≠ NP**," or "neither" as your answer to each question – *we will not award any credit if you write "true" or "false,"* since there are three possibilities for each statement. No justification is necessary.

1. There is a **P** language that can be decided in deterministic polynomial time.
2. There is an **NP** language that can be decided in deterministic polynomial time.
3. There is an **NP-*complete*** language that can be decided in deterministic polynomial time.
4. There is an **NP**-***hard*** language that can be decided in deterministic polynomial time.
5. There is an **NP** language that ***cannot*** be decided in deterministic polynomial time.
6. There is an **NP**-***complete*** language that ***cannot*** be decided in deterministic polynomial time.
7. There is an **NP**-***hard*** language that ***cannot*** be decided in deterministic polynomial time.
8. There is a deterministic, polynomial-time ***verifier*** for every language in **NP**.
9. There is a deterministic, polynomial-time ***decider*** for every language in **NP**.
10. There is a language $L \in$ **P** where $L \leq_p$ 3SAT.
11. There is a language $L \in$ **NP** where $L \leq_p$ 3SAT.
12. There is a language $L \in$ **NPC** where $L \leq_p$ 3SAT.
13. There is a language $L \in$ **P** where 3SAT $\leq_p L$.
14. There is a regular language L where 3SAT $\leq_p L$.
15. All languages in **P** are decidable.
16. All languages in **NP** are decidable.

Problem Seven: The Big Picture (6 Points)

We have covered a *lot* of ground in this course throughout our whirlwind tour of computability and complexity theory. This last question surveys what we have covered so far by asking you to see how everything we have covered relates.

Take a minute to review the hierarchy of languages we explored:

$$\mathbf{REG} \subset \mathbf{CFL} \subset \mathbf{P} \stackrel{?}{=} \mathbf{NP} \subset \mathbf{R} \subset \mathbf{RE} \subset \mathbf{ALL}$$

The following questions ask you to provide examples of languages at different spots within this hierarchy. In each case, you should provide an example of a language, but you don't need to formally prove that it has the properties required. Instead, describe a proof technique you could use to show that the language has the required properties. There are many correct answers to these problems, and we'll accept any of them.

i. Give an example of a regular language. How might you prove that it is regular?

ii. Give an example of a context-free language is not regular. How might you prove that it is context-free? How might you prove that it is not regular?

iii. Give an example of a language in **P**. How might you prove it is in **P**?

iv. Give an example of a language in **NP** that is not known to be in **P**. How might you prove that it is in **NP**? Why don't we know whether it's in **P**?

v. Give an example of a language in **RE** not contained in **R**. How might you prove that it is **RE**? How might you prove that it is not contained in **R**?

vi. Give an example of a language that is not in **RE**. How might you prove it is not contained in **RE**?

Extra Credit Problem: P $\stackrel{?}{=}$ NP (Worth an A+, $1,000,000, and a Stanford Ph.D)

Prove or disprove: **P** = **NP**.