Problem Set 9

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 – the last one of the quarter! – 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.

Good luck, and have fun!

Due Friday, March 11th at the start of lecture.

Because this problem set is due on the last day of class, no late days may be used and no late submissions will be accepted. Sorry about that!

Problem One: Computable Bijections (4 Points)

A function $f : \Sigma^* \to \Sigma^*$ is called a ***computable function*** if it is possible to write a function with signature

```
string compute_f(string input)
```

with the following property: for any $w \in \Sigma^*$, `compute_f(w)` $= f(w)$. In other words, a function is computable if you can write a piece of code that, given any string w, returns the value of $f(w)$.

A ***computable bijection*** is a computable function f that's a bijection. If you remember from way, way back in the quarter, we mentioned that all bijections are invertible. Interestingly, the inverse of any computable bijection is itself a computable function. In other words, if you can write a function `compute_f` that computes the value of a bijection f, then you can write a function `compute_f_inverse` that computes the value of the bijection f^{-1}.

Prove that if f is a computable bijection, then f^{-1} is a computable function. To do so, given an arbitrary computable bijection f, give pseudocode for a function `compute_f_inverse` that, given a string w, returns a string x such that $f(x) = w$. Then, write a brief proof explaining why your code is correct.

As a hint, you don't need to write much code here. If you find yourself writing ten or more lines of pseudocode, you're probably missing something.

Problem Two: Double Verification (4 Points)

Here's an interesting fact: if L is a language where $L \in$ **RE** and $\overline{L} \in$ **RE**, then $L \in$ **R**. This problem will ask you to prove this.

Let L be a language where $L \in$ **RE** and $\overline{L} \in$ **RE**. This means that there's a verifier V_{yes} for L and a verifier V_{no} for $\overline{L}$. In software, you could imagine that V_{yes} and V_{no} correspond to methods with these signatures:

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

Your task is to prove that $L \in$ **R** by writing pseudocode for a function

```
bool isInL(string w)
```

that accepts as input a string w, then returns true if $w \in L$ and returns false if $w \notin L$. Then, write a brief proof explaining why your pseudocode meets these requirements.

As a hint, you don't need to write much code here. If you find yourself writing ten or more lines of pseudocode, you're probably missing something.

Problem Three: Why Loop in *LOOP*? (3 Points)

In lecture, we proved that the language

$$LOOP = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ loops on } w \}$$

is not an **RE** language. The proof worked like this:

1. Assume that *LOOP* ∈ **RE**.
2. That means there's a verifier for *LOOP*, which we can model in software as a method

   ```
   bool imConvincedWillLoop(string program, string input, string certificate);
   ```
3. Construct a self-referential program that uses this method to obtain a contradiction.

The self-referential program that we wrote to show that *LOOP* ∉ **RE** looks pretty different from the self-referential programs we've written in proofs of undecidability. It's natural, then, to wonder why this is.

Consider the following self-referential program that purports to prove that *LOOP* ∉ **RE**:

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

      if (imConvincedWillLoop(me, input, input)) {
            accept();
      } else {
            while (true) {
                  // loop infinitely
            }
      }
}
```

This program, when run on any input, either halts on that input or loops on that input.

i. Is the program above guaranteed to cause a contradiction if it halts on its input? If so, why? If not, why not?

ii. Is the program above guaranteed to cause a contradiction if it loops on its input? If so, why? If not, why not?

Hopefully, your answers to this problem will help you get a better sense for why the self-referential programs we write in proofs involving **RE** look so different from the self-referential programs we write for proofs involving **R**.

Problem Four: Password Checking (8 Points)

Consider the following language:

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

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 certificate)
```

with the following properties:

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

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.

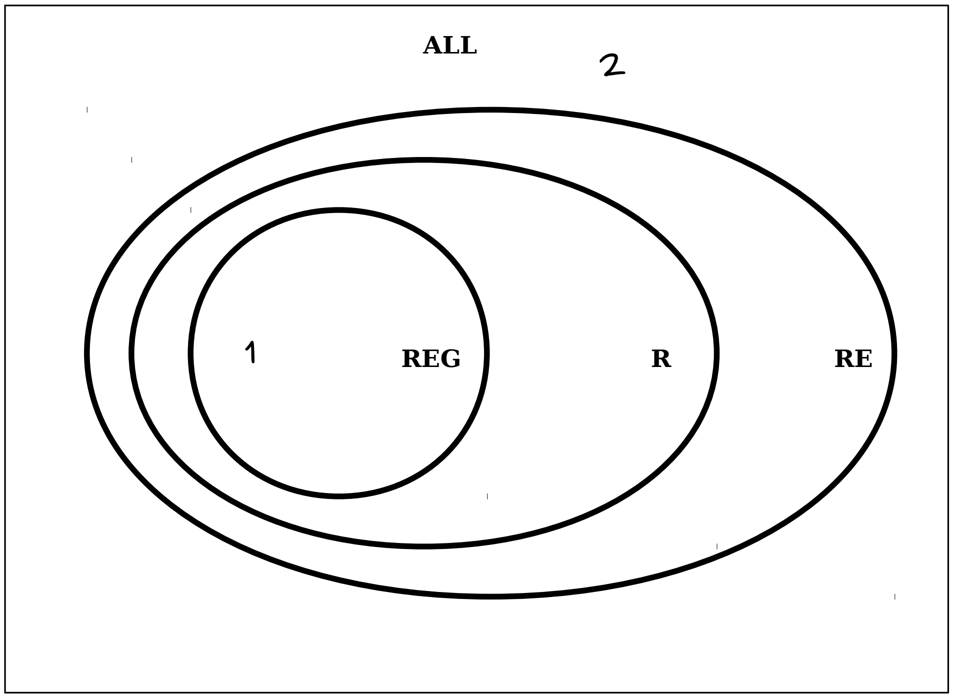
ii. Suppose that this program is ***not*** a valid password checker. Briefly explain why running this program does ***not*** lead to a contradiction.

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. (No formal proof is necessary; you'll prove this in part (iv)).

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 Five: The Lava Diagram (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. L_D
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. $\{\ \mathtt{1}^n\mathtt{+}\mathtt{1}^m\overset{?}{=}\mathtt{1}^{n+m} \mid m, n \in \mathbb{N}\ \}$
6. $\{\ \langle M \rangle \mid M$ is a Turing machine and $\mathscr{L}(M) \neq \varnothing\ \}$
7. $\{\ \langle M \rangle \mid M$ is a Turing machine and $\mathscr{L}(M) = \varnothing\ \}$
8. $\{\ \langle M \rangle \mid M$ is a Turing machine and $\mathscr{L}(M) = L_D\ \}$
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 Monday.)

Problem Six: Resolving P $\overset{?}{=}$ 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 polynomial time.
2. There is an **NP** language that can be decided in polynomial time.
3. There is an **NP-*complete*** language that can be decided in polynomial time.
4. There is an **NP**-***hard*** language that can be decided in polynomial time.
5. There is an **NP** language that ***cannot*** be decided in polynomial time.
6. There is an **NP**-***complete*** language that ***cannot*** be decided in polynomial time.
7. There is a polynomial-time ***verifier*** for every language in **NP**.
8. There is a polynomial-time ***decider*** for every language in **NP**.
9. There is a language $L \in$ **P** where $L \leq_p$ SAT.
10. There is a language $L \in$ **NP** where $L \leq_p$ SAT.
11. There is a language $L \in$ **NPC** where $L \leq_p$ SAT.
12. There is a language $L \in$ **P** where SAT $\leq_p L$.
13. There is a language $L \in$ **NP** where SAT $\leq_p L$.
14. There is a language $L \in$ **NPC** where SAT $\leq_p L$.
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:

$$\textbf{REG} \subset \textbf{CFL} \subset \textbf{P} \overset{?}{=} \textbf{NP} \subset \textbf{R} \subset \textbf{RE} \subset \textbf{ALL}$$

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 $\overset{?}{=}$ NP (Worth an A+, $1,000,000, and a Stanford Ph.D)

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