What problems are beyond our capacity to solve? In this last assignment, you’ll explore the limits of computation.

Exercise 1

In class, we saw that $A_{TM}$ is undecidable, but it is Turing-recognizable. From a programming perspective, this means that it’s possible to write a procedure `will_accept(program, input)` that takes as its arguments string representations of a program and that program’s input and has the following return values:

- If the program would accept that input, `will_accept` must return `true`.
- If the program would not accept the input, `will_accept` may return `false` – or it may go into an infinite loop.

File `ow_my_brain.py`:

```python
def will_accept(program, input):
    # An implementation of a recognizer for ATM:
    # Does the program accept the input?
    ...

def main(my_input):
    # Read ow_my_brain.py
    my_code = my_source()

    if will_accept(my_code, my_input):
        return False  # i.e., reject
    else:
        return True   # i.e., accept
```

Use proof by contradiction to show that this program must loop infinitely on all inputs.

It's fine for this to be a casual “proof sketch” as long as your thinking is clear.
Exercise 2

When you log onto a website with a password, you hope that your password is the only possible input that will log into your account. If you have the source code for the password checking system, could you tell whether your password was the only one it would accept?

Let’s frame the question using 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. Suppose your password is the string love. A TM that would work as a valid password checker would be a TM M where \( L(M) = \{ \text{love} \} \). That is, M accepts your password, love, and it doesn’t accept anything else.

Given a TM, is there some way you could tell if the TM is a valid password checker? This question corresponds to deciding if the encoding \( \langle M \rangle \) of a TM M is in this language:

\[
\text{CHECKER} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \{ \text{love} \} \}
\]

We’ll prove the language CHECKER is undecidable and, thus, there’s no algorithm that can check whether a TM is suitable as a password checker.

Suppose for the sake of contradiction that CHECKER is decidable. This means that we could write a function \text{is\_password\_checker} with the following properties:

- If \( p \) is the source of a program that accepts just the string love, then calling \text{is\_password\_checker}(p) will return True.
- If \( p \) is not the source of a program that accepts just the string love, then calling \text{is\_password\_checker}(p) will return False.

We can prove that CHECKER is undecidable by building a self-referential program that uses \text{is\_password\_checker} to obtain a contradiction. Here’s a first try:

---

1 For instance, a system that allows a “master key” password for logging in to any account has a major security vulnerability!

“All right now, what are the three most common[ly] used passwords?”
“‘love’, ‘secret’, and, uh, ‘sex’, but not in that order necessarily, right?”
Hackers, 1995
File c.py:

```python
def is_password_checker(p):
    ...some implementation...

def main(my_input):
    # Read c.py
    checker = my_source()

    if is_password_checker(checker):
        # Reject the password my_input
        return False
    else:
        # Accept the password my_input
        return True
```

a. Suppose this program (c.py) is a password checker. Briefly explain why running this program leads to a contradiction.

b. Suppose this program is not a password checker. Briefly explain why running this program does not lead to a contradiction.  

*Hint: What language does it recognize in this case?*
c. Our goal in building a self-referential program is for the program to cause a contradiction regardless of whether it's a password checker. As you saw in (b), this program does not cause a contradiction if it is not a password checker. Consider the following modified self-referential program:

```python
def main(my_input):
    checker = my_source()
    if is_password_checker(checker):
        return False
    else:
        if my_input == "love":
            return True
        else:
            return False
```

Briefly explain why this program does cause a contradiction in both cases.

You've now (informally) proven that CHECKER is undecidable. Thus there's no way for a computer to decide if an arbitrary program is a password checker or if it contains a backdoor! :-/
Exercise 3

Prove that each of the following languages is undecidable by reducing $A_{TM}$ to it.

a. $ENTERS = \{(M, w, q) \mid q$ is a state in Turing machine $M,$
and $M$ enters $q$ when run on $w}\$

**Proof** By contradiction. Assume that $ENTERS$ is decidable. This means there is a TM $D$ that decides it. We can use $D$ to construct a TM $A$ that decides $A_{TM}$ ...

---

Take a look at the handout on reductions for examples of this kind of proof.

*Hint:* Keep this one simple! You don’t need to construct an additional Turing machine to give to $D$; just use $M$. Since we have the encoding of $M$, we can tell what its accept state is.
b. \( \text{INFINITE} = \{ (M) \mid L(M) \text{ is infinite} \} \)

\[ \text{Hint: This is very similar to the second reduction from class, for \( \text{REGULAR}_{\text{TM}} \).} \]

For that proof, we made a decider for \( \text{HALT}_{\text{TM}} \) by constructing a special TM whose language was regular if \( M \) would halt on \( w \) and non-regular if \( M \) would loop on \( w \).

For this problem, you can make a decider for \( \text{A}_{\text{TM}} \) by constructing a special TM whose language is infinite when \( M \) accepts \( w \) and finite when \( M \) doesn't accept \( w \)!