7 Undecidability Reductions

In the previous chapter we saw our first example of an undecidable language: HALTS. In this chapter we will discuss *reductions*, a powerful technique for relating a new problem to one that we've already solved which we will use throughout the rest of the course. In Section 7.1 we will define reductions and use them to show that a variety of languages about the encodings of algorithms are undecidable. In Section 7.2, we will introduce an undecidable problem which is *not* about the encodings of algorithms called the Post Correspondence Problem, and show how PCP can also be used as a basis for undecidability reductions. Finally in Section 7.3 we will extend our discussion of the limits of computation beyond algorithms that return a boolean by defining uncomputable functions and showing that a particular function on the integers is uncomputable.

7.1 Turing Reductions

One of the most powerful concepts in computer science is *reduction*. Reducing problem A to problem B means giving an algorithm to solve A, assuming that you already have an algorithm for B. Reductions are used in everyday life. For example, if I am lost in Texas I can reduce the problem of getting home to the problem of finding Highway 6 (since I already know how to navigate to my apartment from Highway 6). They are also commonly used in programming. For example, say that you write a function to find the minimum element of an array by calling a "sort" function from the standard library and then taking the first element of the array. You have reduced the problem of finding the minimum element to the problem of sorting an array. Reductions are more than just a technique for solving new problems, however. Reductions give us a way to compare the "difficulty" of problems, and as we'll see in this section, they are a useful tool for proving that a language is undecidable.

For two languages A and B we say that $A \leq_T B$ (read "A **Turing reduces to** B") if it is possible to construct a decider for A using a decider for B as a subroutine. We use a less than or equal symbol because Turing reductions allow us to compare the difficulty with respect to decidability. In particular, if $A \leq_T B$ then we can say the following:

1. If B is decidable, then so is A.
2. If A is undecidable, then so is B.

Conclusion 1 follows from the definition of a Turing reduction. If $A \leq_T B$, then we can construct a decider for A given a decider for B, and if B is decidable then we have a decider for B. Conclusion 2 is simply the contrapositive of conclusion 1.

Theorem 33. *The language*

$$\text{RET-TRUE} := \{\langle P, x\rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ returns True}\}$$

is undecidable.

Proof. We could use the same technique as in Theorem 32 to prove this language undecidable, but in the interest of demonstrating reductions, we will instead prove this by showing that

$$\text{HALTS} \leq_T \text{RET-TRUE}.$$

We need to construct a decider for HALTS, assuming that we have access to a decider for RET-TRUE. Let RTDecider be a decider for RET-TRUE. We construct a decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Simulate P(y)
            Return True
    Return RTDecider(Helper, x)
```

To show that HaltsDecider correctly decides HALTS we need to show that for every $\langle P, x\rangle \in$ HALTS, HaltsDecider(P, x) returns True and for every $\langle P, x\rangle \notin$ HALTS, HaltsDecider(P, x) returns False.

- Suppose that $\langle P, x\rangle \in$ HALTS. Then $P(x)$ halts, so Helper(x)'s simulation of $P(x)$ will finish and Helper(x) will return True. Therefore $\langle$Helper$, x\rangle \in$ RET-TRUE and thus RTDecider(Helper$, x)$ returns True, so HaltsDecider(P, x) also returns True.
- Now suppose that $\langle P, x\rangle \notin$ HALTS. Then $P(x)$ loops, so Helper(x)'s simulation of $P(x)$ will loop forever and Helper(x) will never return True. Therefore $\langle$Helper$, x\rangle \notin$ RET-TRUE and thus RTDecider(Helper$, x)$ returns False, so HaltsDecider(P, x) also returns False.

Thus, we have shown that HALTS $\leq_T$ RET-TRUE, so since HALTS is undecidable, RET-TRUE is also undecidable. □

This proof may seem a little counterintuitive. After all, we constructed a decider for HALTS which we just showed in the previous chapter is impossible! This reduction is essentially just another form of proof by contradiction. We showed that if RET-TRUE were decidable, then HALTS would be as well. Since we know HALTS is not decidable, neither is RET-TRUE.

It's also important to recognize that our decider for HaltsDecider constructed a helper algorithm Helper, *but never actually ran* Helper. If HaltsDecider simulated running Helper (or P) we couldn't say that HaltsDecider is a decider, because the simulation of P might cause HaltsDecider to loop forever. But since Helper is never run, just written as source code that gets passed into RTDecider (which always halts because we assumed it is a decider), HaltsDecider is guaranteed to always return an answer without infinite looping.

Theorem 34. *The language*

$$\text{EMPTY} := \{\langle P\rangle \mid P \text{ is an algorithm and } L(P) = \emptyset\}$$

is undecidable.

Proof. We prove this by showing that

$$\text{HALTS} \leq_T \text{EMPTY}.$$

Let EmptyDecider be a decider for EMPTY. We construct a decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Simulate running P(x)
            Return True
    Return ¬EmptyDecider(Helper)
```

Now we prove that HaltsDecider decides HALTS.

- Suppose that $\langle P, x\rangle \in$ HALTS. Then for every $y \in \Sigma^*$, Helper(y) finishes simulating $P(x)$ and therefore returns True. Thus $L($Helper$) = \Sigma^* \neq \emptyset$, so $\langle$Helper$\rangle \notin$ EMPTY, so EmptyDecider(Helper) returns False, so HaltsDecider(P, x) returns True.
- Suppose that $\langle P, x\rangle \notin$ HALTS. Then for every $y \in \Sigma^*$, Helper(y) never finishes simulating $P(x)$ and therefore loops forever. Thus $L($Helper$) = \emptyset$, so $\langle$Helper$\rangle \in$ EMPTY, so EmptyDecider(Helper) returns True, so HaltsDecider(P, x) returns False.

□

Notice that once again, our machine HaltsDecider created a helper machine Helper without actually running it. This time Helper ignores its own input string and runs $P(x)$ (rather than $P(y)$) so that the behavior of Helper on *all* strings depends on the behavior of M on the specific string x. The other major difference compared to the previous reduction is that HaltsDecider is doing the *opposite* of what EmptyDecider does, because $P(x)$ halting corresponds to $\langle \text{Helper} \rangle \notin \text{EMPTY}$.

Theorem 35. *The language*

$$\text{ACCEPTS-2} := \{\langle P \rangle \mid P \text{ is an algorithm and } |L(P)| = 2\}$$

is undecidable.

Proof. We prove this by showing that

$$\text{HALTS} \leq_T \text{ACCEPTS-2}.$$

Let A2Decider be a decider for ACCEPTS-2. We construct a decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Simulate running P(x)
            If y = ε or y = "411":
                Return True
            Return False
    Return A2Decider(Helper)
```

Now we prove that HaltsDecider decides HALTS.

- Suppose that $\langle P, x \rangle \in \text{HALTS}$. Then for every $y \in \Sigma^*$, Helper(y) finishes simulating $P(x)$. Therefore Helper(y) returns True ε and "411" and returns False on all other inputs. Thus $L(\text{Helper}) = \{\varepsilon, \text{"411"}\}$, so $|L(\text{Helper})| = 2$, so $\langle \text{Helper} \rangle \in \text{ACCEPTS-2}$, so A2Decider(Helper) returns True, so HaltsDecider(P, x) returns True.
- Suppose that $\langle P, x \rangle \notin \text{HALTS}$. Then for every $y \in \Sigma^*$, Helper(y) never finishes simulating $P(x)$ and therefore loops forever. Thus $L(\text{Helper}) = \emptyset$, so $\langle \text{Helper} \rangle \notin \text{ACCEPTS-2}$, so A2Decider(Helper) returns False, so HaltsDecider(P, x) returns False.

□

At this point you may be seeing the general pattern. To prove that some language L containing the encodings of algorithms is undecidable, we show that $\text{HALTS} \leq_T L$. To prove this we construct a decider for HALTS. This decider for HALTS constructs a helper machine which runs $P(x)$ and then returns True if its input y is in some set $S \subseteq \Sigma^*$ (where the choice of S depends on L). If $\langle P, x \rangle \in \text{HALTS}$, then $L(\text{Helper}) = S$, otherwise $L(\text{Helper}) = \emptyset$. If machines with language S are in L and machines with language $\emptyset$ are not, or vice-versa, then we can decide HALTS by running our decider for L on Helper and returning either the same thing (if $\langle \text{Helper} \rangle$ is in L when $L(\text{Helper}) = S$) or the opposite (if $\langle \text{Helper} \rangle$ is in L when $L(\text{Helper}) = \emptyset$).

This general pattern can work even for some languages that look more complicated, like the following one which involves pairs of algorithms:

Theorem 36. *The language*

$$\text{UNIQUE-INTERSECT} := \{\langle P_1, P_2 \rangle \mid P_1 \text{ and } P_2 \text{ are algorithms with } |L(P_1) \cap L(P_2)| = 1\}$$

is undecidable.

Proof. To prove that UNIQUE-INTERSECT is undecidable, we show that

$$\text{HALTS} \leq_T \text{UNIQUE-INTERSECT}.$$

Let UIDecider be a decider for UNIQUE-INTERSECT. We create the following decider HaltsDecider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of the following algorithm Helper:
        Helper(y)
            Run P(x)
            If y = ε:
                Return True
            Else:
                Return False
    Return UIDecider(Helper, Helper)
```

Correctness:

- Suppose that $\langle P, x\rangle \in$ HALTS. Then $L(\text{Helper}) = \{\varepsilon\}$, so $|L(\text{Helper}) \cap L(\text{Helper})| = |\{\varepsilon\}| = 1$. Thus $\langle \text{Helper}, \text{Helper}\rangle \in$ UNIQUE-INTERSECT, so UIDecider(Helper, Helper) will return True, so HaltsDecider(P, x) will return True.
- Suppose that $\langle P, x\rangle \notin$ HALTS. Then $L(\text{Helper}) = \emptyset$, so $|L(\text{Helper}) \cap L(\text{Helper})| = |\emptyset| = 0$. Thus $\langle \text{Helper}, \text{Helper}\rangle \notin$ UNIQUE-INTERSECT, so UIDecider(Helper, Helper) will return False, so HaltsDecider(P, x) will return False.

□

So far all of the reductions we have seen have been from HALTS. This isn't the only language we can reduce from, however. To prove that a language is undecidable, we can reduce from any language that already know is undecidable (so far we've added RET-TRUE, EMPTY, ACCEPTS-2, and UNIQUE-INTERSECT to that list).

Theorem 37. *The language*

$$\text{FINITE} := \{\langle P\rangle \mid P \text{ is an algorithm and } L(P) \text{ is finite}\}$$

is undecidable.

Proof. To prove that FINITE is undecidable, we will show that

$$\text{EMPTY} \leq_T \text{FINITE}.$$

Let FiniteDecider be a decider for FINITE. We create the following decider EmptyDecider for EMPTY:

```
EmptyDecider(P)
    Σ ← the alphabet of P
    Write out the source code of the following algorithm Helper:
        Helper(x)
            For num_steps from 1 to ∞:
                For n from 0 to num_steps:
                    For each s ∈ Σ^n:
                        Simulate P(s) for num_steps steps
                        If the simulation returned True:
                            Return True
    Return FiniteDecider(Helper)
```

Correctness:

- Suppose that $\langle P \rangle \in$ EMPTY. Then P never returns True for any input, so no matter what string s HELPER simulates P on, $P(s)$ will never return True, and therefore HELPER(x) will never return True for any x. Thus $L(\text{HELPER}) = \emptyset$. Therefore $|L(\text{HELPER})| = |\emptyset| = 0$, which is clearly finite, so $\langle \text{HELPER} \rangle \in$ FINITE. Therefore FINITEDECIDER(HELPER) will return True, so EMPTYDECIDER(P) will return True.
- Suppose that $\langle P \rangle \notin$ EMPTY. Then there exists some $y \in \Sigma^*$ such that $P(y)$ returns True. Let t be the number of steps that it takes for $P(y)$ to return True. We note that each iteration of the outer for loop of HELPER eventually finishes (i.e. runs in finite time) because there are only a finite number of strings of length at most num_steps, and we run P for only a finite number of steps on each of them. Thus, if it has not already returned True, HELPER will eventually reach an iteration of the outer for loop where num_steps is $\max(|y|, t)$. In this iteration, it will simulate $P(y)$ for at least t steps, and therefore the simulation will return True and so HELPER will return True.

  Therefore, HELPER will return True for every input string (since HELPER does the same thing for every input string x), and thus $L(\text{HELPER}) = \Sigma^*$, which is infinite. Therefore $\langle \text{HELPER} \rangle \notin$ FINITE, so FINITEDECIDER(HELPER) will return False, and therefore EMPTYDECIDER(P) will return False.

□

In this case, reducing from EMPTY did not make it easier to show that FINITE was undecidable (see if you can find an alternative proof using a reduction from HALTS). In the next section, we'll see an example of a problem where HALTS is actually not the best problem to reduce from to prove undecidability.

However, this proof did allow us to show off a technique for running a program on infinitely many strings in parallel (despite the fact that our model of computation is inherently sequential!). The approach that we used in this proof's helper machine of running P on increasingly large sets of strings for increasingly large numbers of steps (which guarantees that if there is any string for which P returns True, we will find it in some finite amount of time) is called **dovetailing**, and it occasionally comes up in other problems in complexity theory.

Note that our reduction would not have worked if we had tried to use a simpler helper machine like the one below which runs P on strings one at a time:

```
HELPER(x)
  For n from 0 to ∞:
      For each s ∈ Σ^n:
          Simulate P(s)
          If the simulation returned True:
              Return True
```

If, for example, $P(\varepsilon)$ loops forever, then this version of the helper algorithm will get stuck simulating $P(\varepsilon)$ forever, and never move on to longer strings. Thus, even if P returns True on some other string, this version of the helper machine would never find it. That's why we needed the dovetailing technique.

7.2 The Post Correspondence Problem

Not all undecidable languages are about encodings of algorithms. In 1946, Emil Post described the following problem, which became known as the Post Correspondence problem, and showed that it cannot be solved by any algorithm.

In the **Post correspondence problem**, we are given some finite set of dominoes where each domino has a string written on the top and the bottom. For example, we might be given the set:

$$\left\{ \left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right] \right\}.$$

The goal of the problem is to find a sequence of one or more copies of dominoes from this set such that the concatenation of the strings on the top of the dominoes is the same as the concatenation of the strings on the bottom of the dominoes. We call such a sequence a **match**. For example, the following arrangement would be a match for the set of dominoes above:

$$\left[\frac{a}{ab}\right]\left[\frac{b}{ca}\right]\left[\frac{ca}{a}\right]\left[\frac{a}{ab}\right]\left[\frac{abc}{c}\right].$$

Both the top and the bottom strings are *abcaaabc*. On the other hand the set of dominoes below has no match:

$$\left\{\left[\frac{ab}{a}\right], \left[\frac{cab}{bc}\right], \left[\frac{a}{\varepsilon}\right], \left[\frac{aaa}{b}\right], \left[\frac{dcb}{bca}\right]\right\}.$$

To see this we note that we cannot use the fifth domino, because it has a d on top and none of the dominoes have a d on the bottom, and we cannot form a solution out of the first four dominoes, because on each of them the top string is longer than the bottom string, so we cannot have a sequence of dominoes where the top and bottom strings are even the same length, let alone equal to each other.

Theorem 38. *The language*

$$\text{PCP} := \{\langle P\rangle \mid P \text{ is an instance of the Post correspondence problem with a solution}\}$$

is undecidable.

We will not prove this theorem, because a full proof would take several pages and get deep into the nitty-gritty instruction sets of RAM model programs. The very high level idea is to prove that HALTS $\leq_T$ PCP by constructing a set of dominoes for which any solution to the problem would have the matching top and bottom string represent the sequence of instructions and configurations that P goes through when run on x. This sequence of configurations only terminates (and thus corresponds to a finite sequence of dominoes) if $P(x)$ eventually halts.

Even though we did not prove it ourselves, we can use Theorem 38 to prove the undecidability of other languages.

Theorem 39. *The language*

$$\begin{aligned}\text{MPCP} := \{\langle P, d\rangle \mid\ & P \text{ is an instance of the Post correspondence problem,}\\ & d \text{ is a domino in } P \text{ and } P \text{ has a solution beginning with } d\}\end{aligned}$$

is undecidable.

Proof. To prove that MPCP is undecidable, we will show that PCP $\leq_T$ MPCP. Let MPCPDecider be a decider for MPCP. We create the following decider PCPDecider for PCP:

```
PCPDecider(P)
    For each domino d in P:
        If MPCPDecider(P, d) returns True:
            Return True
    Return False
```

□

Correctness:

- Suppose $\langle P\rangle \in$ PCP. Then P has a match, and this match must begin with some domino x from P. Since there are only finitely many dominoes in P and MPCPDecider(P, d) runs in finite time for each of them (since MPCPDecider is a decider), the for loop in PCPDecider will eventually reach an iteration where $d = x$ and it will run MPCPDecider(P, x). By our choice of x, MPCPDecider(P, x) returns True, so PCPDecider(P) will return True.

- Suppose $\langle P \rangle \notin$ PCP. Then P has no solution, so it does not have a solution starting with d for any domino d in P. Therefore all of the calls that PCPDecider makes to MPCPDecider will return False, so PCPDecider(P) will return False.

PCP has been used as a starting point to prove several other more interesting languages undecidable, although the proofs are beyond the scope of this course.

7.3 Uncomputable Functions

So far in this chapter and the previous one, we have focused on programs that only return True or False (or loop forever) in order to prove that languages are decidable or undecidable. The idea of decidability can naturally be extended to arbitrary functions, however. We say that a function $f : X \to Y$ is **computable** if there is some algorithm that computes it, i.e. if there exists an algorithm P such that for every $x \in X$, $P(\langle x \rangle)$ outputs $\langle f(x) \rangle$ (We do not care what P does when its input string does not encode an element of f's domain X. We can generally just assume that the algorithm returns ε in this case). We say that a function is **uncomputable** if it is not computable.

While Turing reductions are only defined between languages, we can prove that a function is uncomputable using a similar technique: assume for sake of contradiction that the function is computable, then use the algorithm which computes the function to create a decider for HALTS (or some other known undecidable language). Since HALTS is undecidable, this is a contradiction, and so our assumption that the function was computable must have been false.

Let's see an example of this technique. We define a **binary RAM model program** as a RAM model program with alphabet $\Sigma = \{0, 1\}$ which does not use any integer constants other than 0 and 1. We define the **busy beaver function**, $BB : \mathbb{Z}^+ \to \mathbb{Z}^+$ as follows: $BB(n)$ is the maximum number of steps that an n-instruction binary RAM model program can take on input ε before halting (here the max is across all n-instruction binary RAM model programs which halt on input ε).

Theorem 40. *BB is uncomputable.*

Proof. AFSOC that BB is computable. Let Beaver be an algorithm that computes BB.

We construct the following decider for HALTS:

```
HaltsDecider(P, x)
    Write out the source code of a binary RAM model program Helper:
        Helper(y)
            Return P(x)
    n ← the number of instructions in Helper
    t ← Beaver(n)
    Simulate Helper(ε) for t steps
    If the simulation of Helper halted:
        Return True
    Return False
```

We note that we have assumed here that we can simulate P using a helper algorithm which is a *binary* RAM model program. i.e. we have assumed that there exists a universal machine which uses $\Sigma = \{0, 1\}$ and uses no integer constants besides $\{0, 1\}$. This is true, and you may assume this fact in your proofs, but we will not prove it, since we did not even formally prove that universal machines exist in the first place.

We will now prove that HaltsDecider correctly decides HALTS:

- Suppose that $\langle P, x \rangle \in$ HALTS. Then Helper(ε) will halt. Since Helper is a binary RAM model program with n instructions, we know that it halts in at most $BB(n)$ steps. Since Beaver(n) computes $BB(n)$, this means that the simulation of Helper(ε) will halt within the t steps that we run it for, so HaltsDecider(P, x) will return True.

- Suppose that $\langle P, x \rangle \notin$ HALTS. Then Helper(ε) loops forever, so when we simulate it for t steps it will not halt. Thus HaltsDecider(P, x) will return False.

□

The busy beaver function is traditionally defined in terms of Turing machines. Since we have avoided describing the technical details of Turing machines, in this course we will use our version based on binary RAM model programs. Note that the restriction to programs that only use the constants 0 and 1 is necessary for the function to be well-defined, because if we allow arbitrarily large constants then we can make small programs that run arbitrarily long. For example, in the following program we can set c to any positive constant to obtain a 5-line program that runs for more than $3c$ steps.

1: i $\leftarrow$ 0
2: Malloc('0')
3: i $\leftarrow$ i + 1
4: If i < c: goto 2
5: Output(1,1)

We can create a binary RAM model program with the same functionality as the one above, but only by letting the number of instructions grow in proportion to c (i.e. by replacing the loop with c copies of lines 2 and 3).

7.4 Comprehension Questions

1. Suppose that A and B are languages and $A \leq_T B$. Which of the following can we conclude? (select all that apply)

   (a) If A is decidable, then B is decidable.

   (b) If B is decidable, then A is decidable.

   (c) If A is undecidable, then B is undecidable.

   (d) If B is undecidable, then A is undecidable.

2. Let IS-EVEN $:= \{\langle n \rangle \mid n \text{ is an even integer}\}$. Let SUBSET-SUM $:= \{\langle S, k \rangle \mid S \subseteq \mathbb{Z}, k \in \mathbb{Z}, \exists T \subseteq S, \sum_{x \in T} x = k\}$. Which of the following are true?

   (a) IS-EVEN $\leq_T$ HALTS

   (b) IS-EVEN $\leq_T$ SUBSET-SUM

   (c) SUBSET-SUM $\leq_T$ HALTS

   (d) SUBSET-SUM $\leq_T$ IS-EVEN

   (e) HALTS $\leq_T$ IS-EVEN

   (f) HALTS $\leq_T$ SUBSET-SUM

3. Define a language RET-FALSE $:= \{\langle P, x \rangle \mid P \text{ is an algorithm, } x \in \Sigma^* \text{ and } P(x) \text{ returns False}\}$. Suppose we wish to show that HALTS $\leq_T$ RET-FALSE. We assume that we have a decider RFDecider for RET-FALSE and give the following decider for HALTS:

   ```
   HaltsDecider(P, x)
     Write down the source code of an algorithm Helper.
     Return RFDecider(Helper, x)
   ```

   To finish this proof we need to describe how to design Helper. Which of the following choices for Helper would allow HaltsDecider to correctly decide HALTS?

   (a)
   ```
   Helper(y)
     Run P(x)
     Return True
   ```

(b)
```
HELPER(y)
  Run P(x)
  Return False
```

(c)
```
HELPER(y)
  If y = x:
      Return False
  Run P(x)
  Return True
```

(d)
```
HELPER(y)
  If y = x:
      Return True
  Run P(x)
  Return False
```

(e)
```
HELPER(y)
  Run P(x)
  Return y = x
```

4. Suppose that we have an instance of the Post correspondence problem with the following 5 dominoes:

   (a) $\left[\frac{a}{bc}\right]$
   (b) $\left[\frac{b}{ab}\right]$
   (c) $\left[\frac{ca}{a}\right]$
   (d) $\left[\frac{cba}{cb}\right]$
   (e) $\left[\frac{acb}{ab}\right]$

   Which of these dominoes must come first in any match for this instance of the problem?

5. Let P be an instance of PCP and let d be a domino from P. Which of the following statements are guaranteed to be true? (select all that apply)

   (a) If $\langle P \rangle \in$ PCP then $\langle P, d \rangle \in$ MPCP.
   (b) If $\langle P, d \rangle \in$ MPCP then $\langle P \rangle \in$ PCP.

6. True or false? $\forall\, n \in \mathbb{Z}^+, \exists\, t \in \mathbb{Z}^+$, every n instruction binary RAM model program which halts on input ε halts on ε after at most t steps.

7.5 Exercises

1. Prove that the following language is undecidable:

$$L := \{\langle P \rangle \mid P \text{ is an algorithm and there exist at least two distinct input strings } x, y \in \Sigma^* \text{ such that } P(x) \text{ and } P(y) \text{ loop forever.}\}$$

2. Define

   REJECTS-3 := $\{\langle P \rangle \mid P$ is an algorithm which returns False for exactly 3 different input strings$\}$

   Prove that REJECTS-3 is undecidable.

3. Define

$$\text{EQ} := \{\langle P_1, P_2 \rangle \mid P_1 \text{ and } P_2 \text{ are algorithms with } L(P_1) = L(P_2)\}.$$

   Prove that EQ is undecidable.

4. Define

$$\text{DISJOINT} := \{\langle P_1, P_2\rangle \mid P_1 \text{ and } P_2 \text{ are algorithms with } L(P_1) \cap L(P_2) = \emptyset\}.$$

Prove that DISJOINT is undecidable.

5. Prove that RET-TRUE $\leq_T$ HALTS
6. Prove that EMPTY $\leq_T$ HALTS
7. Prove that the following language is undecidable

$$\begin{aligned}\text{ODD-PCP} :=\{\langle P\rangle \mid\ & P \text{ is an instance of the Post correspondence problem,}\\ & \text{which has a solution using an odd number of total dominoes}\}\end{aligned}$$

8. Prove that the following language is undecidable:

$$\begin{aligned}L :=\{\langle P\rangle \mid\ & P \text{ is an instance of the Post correspondence problem where}\\ & \text{every non-empty top/bottom string on a domino begins}\\ & \text{with the symbol 'a' and } P \text{ has a match}\}\end{aligned}$$

9. Show that the function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ defined by
$f(n) :=$ the number of n-instruction binary RAM model programs which halt on ε
is uncomputable.
10. Show that the function $g : \{P \mid P \text{ is an algorithm}\} \times \mathbb{Z}^+ \to \mathbb{Z}^+$ defined by
$g(P, n) :=$ the number of strings $x \in \Sigma^n$ for which $P(x)$ returns True
is uncomputable.
11. Show that the function $h : \{P \mid P \text{ is an binary RAM model program}\} \to \mathbb{Z}^+$ defined by
$h(P) :=$ The number of instructions in the shortest binary RAM model program with the same language as P (i.e. the P' with the fewest instructions for which $L(P') = L(P)$)
is uncomputable.
12. We define the following languages:

$$\begin{aligned}\text{MM3} :=\{\langle S\rangle : & S \text{ is a set of } 3\times 3 \text{ matrices with integer entries such that it is possible}\\ & \text{to obtain the zero matrix by multiplying a finite sequence of matrices from } S\}\\ \text{MM4} :=\{\langle S\rangle : & S \text{ is a set of } 4\times 4 \text{ matrices with integer entries such that it is possible}\\ & \text{to obtain the zero matrix by multiplying a finite sequence of matrices from } S\}\end{aligned}$$

For example, suppose S is a set containing the following matrices:

$$\begin{bmatrix}1&0&0\\0&1&0\\0&0&0\end{bmatrix}, \begin{bmatrix}-1&-2&-3\\4&5&6\\-7&-8&-9\end{bmatrix}, \begin{bmatrix}1&1&0\\0&0&1\\0&0&1\end{bmatrix}, \begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}$$

Then $\langle S\rangle \in$ MM3 because

$$\begin{bmatrix}1&0&0\\0&1&0\\0&0&0\end{bmatrix}\begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix}\begin{bmatrix}1&1&0\\0&0&1\\0&0&1\end{bmatrix}\begin{bmatrix}1&0&0\\0&1&0\\0&0&0\end{bmatrix} = \begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}$$

MM3 is known to be undecidable (you may assume this). Prove that MM4 is undecidable.