(I) Define a two-head deterministic finite automaton (2HDFA) $M = (Q, \Sigma, \delta, s, F)$ to be like a DFA except that its instructions have the form $(p, (c_1, c_2)/ (D_1, D_2), q)$ where $D_1$ and $D_2$ can be $R$ for move right or $S$ for stay. Without loss of generality we may suppose that each instruction moves at least one of the two heads right.

(a) Sketch how a 2HDFA can recognize a few of the languages we’ve proved to be nonregular. (Giving the full machine is optional; it’s not much more than a sketch anyway.)

(b) Can a 2HDFA recognize the language of palindromes? What do you think? (Proof not needed.)

(c) Explain how, for any fixed one-tape DTM $M$, a 2HDFA $H$ can decide the Kleene $T$-predicate for $M$ under a natural encoding scheme where $\vec{c}$ is a sequence of IDs. That is, show how $H$, given $x, I_0, I_1, \ldots, I_t$ as a string, can decide whether this is a valid accepting computation of $M$ on input $x$. (To get you started, note that $I_0$ is $sx_1x_2\cdots x_n$, or if you prefer using members of $Q \times \Gamma$ as single chars, $I_0 = (s) x_2 \cdots x_n$. You may suppose the comma is not a char in the work alphabet $\Gamma$ of $M$.)

Answer: (a) A 2HDFA $H$ can recognize $\{a^nb^n : n \geq 0\}$, which was our first example of a non-regular language, by this strategy: Head 1 advances to the first $b$ (if the tape is empty, then $n = 0$ and $H$ accepts; else if there is no $b$, it rejects). Then the two heads count off step-by-step, and if Head 1 keeps reading $b$’s and hits the blank at the same time Head 2 has been reading $a$’s and hits the first $b$, $H$ accepts. A 2HDFA $H'$ can recognize the language $\{x\#x : x \in \Sigma^*\}$ of double-words whose break is marked by the char $\#$ which is not in $\Sigma$ by a similar strategy after Head 1 marches alone to the $\#$ and then one cell further. (This is meaningful in the context of CSE396 because the double-word language is not context-free either.) However, (b) a 2HDFA $H$ cannot recognize the language of palindromes. (This can be proved by adapting the crossing-sequence framework of presentation option D on HW4 regarding the palindrome language, where you can show that $H$ gives a crossing sequence of length $k = 2$ with the same key property of determining that $w$ on the right side must equal the $v^R$ from the left side.)

(c) The first ID $I_0$ must be $sx$, so checking that the beginning $x, I_0$ is correct is much like checking the double-word language as above. The ID $I_1$ will differ from $I_0$ only in the first char or two, and that difference can be checked according to the $\delta_M$ of $M$. Because $\delta_M$ is fixed, checking the consistency of what happens at the beginning is programmable directly by giving our 2HDFA $H_M$ enough states to cover all state-char combinations of $M$. The rest of each ID must be identical, and that is like checking the double-word language again. To check that $I_2$ correctly follows from $I_1$ is similar, and $H_M$ is set up perfectly to do that right when the check of $I_0 \vdash_M I_1$ has completed. This knocks on all the way through. It is possible that $I_{j+1}$ can be longer or shorter than $I_j$ by one character, but that will happen at one end or the other according to $\delta_M$ and can be checked similarly. The tape heads basically march in lockstep (once head 1 has advanced into $I_0$ initially) and check the whole thing in one left-to-right pass. The larger significance is that checking a computation is the Turing-machine analogue of checking a proof—and the analogy is brought to fruition by the enxt problem.
(II) Deduce from (I) that the emptiness problem for 2HDFAs, whose language can be called $E_{2\text{HDFA}}$, is undecidable—indeed, mapping equivalent to $E_{TM}$. Conclude along lines of the presentation option (A) on HW4 (whose key is being given) that for any program verification logic $F$, there exist 2HDFAs $H$ such that $L(H) = \emptyset$ but $F$ cannot prove it. (IMHO, 2HDFAs are about the simplest kind of code whose behavior cannot be generally verified.)

**Answer:** Given a 2DFA $H$, if $L(H) \neq \emptyset$ then we can prove it just as we can for a general TM in place of $H$. So $E_{2\text{HDFA}}$ belongs to co-RE. To show it is complete, we show $E_{TM} \leq_m E_{2\text{HDFA}}$. This reduction is computed by the mapping $f((M)) = \langle H_M \rangle$ in (I) above, because: $L(H_M) = \emptyset \iff M$ has no accepting computations $\iff L(M) = \emptyset$.

It follows that the language $B = E_{2\text{HDFA}}$ is not c.e. For any adequate and effective logic $F$, however, the language $A$ of 2HDFAs $H$ such that $F$ proves $L(H) = \emptyset$ is c.e. This is because (by the adequacy of $F$ for computation) we can formalize in $F$ a statement $S_H$ expressing “$L(H) = \emptyset$” and then define

$$A = \{(H) : (\exists \pi)R_F(S_H, \pi)\},$$

using the decidable proof predicate $R_F$ from the HW4 presentation options (A,B). Thus $A \neq B$. By the presumed soundness of $F$, $A$ is a subset of $B$. So $B \setminus A$ is nonempty. Any $H$ in the difference is a 2DFA whose language is empty, but $F$ cannot prove it.

(III) In his original 1936 paper, Alan Turing thought in terms of “computable numbers” $z_L$ rather than decidable languages $L$. To define what he meant by example, the language $P = \{2, 3, 5, 7, 11, 13 \ldots \}$ of prime numbers becomes the binary irrational number

$$z_P = 0.0110101000101 \ldots$$

That is, we identify binary strings and positive integers under the correspondence $1 \leftrightarrow \epsilon$, $2 \leftrightarrow 0$, $3 \leftrightarrow 1$, $4 \leftrightarrow 00$, and so on. Then the $i$-th bit of $z_L$ is a 1 if $i$ is in $L$, and is 0 otherwise. (If we want to allow zero as a number we can use the bit before the “binary decimal point” for it.)

Prove that a language $L$ is decidable if and only if there is a total TM $T$ that on any input $n$ outputs the first $n$ bits of $z_L$, which is what Turing meant by an irrational number being “computable.” What happens for rational numbers whose denominator is a power of 2?

**Answer:** First suppose $L$ is decidable. We can take a total TM $M$ such that $L(M) = L$ and need to build a total transducer $T_M$ such that for all $n$, $T_M(n)$ computes $z_L$ to $n$ places. Here is C++ style pseudocode:

```c++
string out = "";
for (int i = 1; i <= n; i++) {
    out += (M accepts i ? '1' : '0');
}
return out;
```

Note that the last bit is the run of $M$ on the $n$th string itself. Conversely, if $z_L$ is computable in Turing’s sense, then we have a total transducer $T_L$ such that for any input $n$, $T_L(n)$ computes $z_L$ to $n$ places. Build $M(n)$ to compute $T_L(n)$ and accept if the $n$th bit is a ‘1,’ rejecting otherwise. Then $M$ is total and $L(M) = L$, so $L$ is decidable.
A rational number \( r \) whose denominator is a power \( 2^k \) will need at most \( k \) “binary decimal” places; everything after that will be 0. It follows that \( r = z_L \) for a language \( L \) that is a subset of \( \{1, \ldots, k\} \), so the language \( L \) is finite. And vice-versa: if \( L \) is finite then we can let \( k \) be its biggest number; then \( z_L \) is a fraction over \( 2^k \). There is, however, a binary expansion \( z'_L \) for \( L \) that gives the same number \( r \) but a different binary expansion ending in infinitely many 1s. For example, \( \frac{1}{2} = 0.1000 \cdots = 0.01111 \cdots \) and \( \frac{25}{32} = 0.1100100 \cdots = 0.11000111 \cdots \). The languages corresponding to these expansions are exactly the complements of finite sets, called cofinite sets. Except for them, there is a 1-to-1 correspondence between languages \( L \) and the value of \( z_L \) as a real number between zero and one.

(IV) Show that the following decision problem is not only undecidable, its language is not c.e.:

**Instance:** A deterministic Turing machine \( M \).

**Question:** Does \( M \) run in polynomial time?

Deduce, again along lines of the presentation option (A) on HW4, that for any reasonable system of logic \( F \), there are algorithms that run in polynomial time, but \( F \) cannot prove that they do so.

**Answer:** This time we do a delay-switch reduction from the \( D_{TM} \) language, mapping a given TM \( M \) to the following TM \( M' \). In pseudocode, it goes:

```
input x;
int n = |x|;
Simulate M(M) for up to n steps /by this simulator/
if (M did not accept M within that time) {
    halt; //note this all took only O(n) time
} else { //M accepted M, so we "panic" by running for more than polynomial time
    do something that takes \( 2^n \) steps before halting
}
```

The main point is that if \( M \) accepts its own code, then it does so in some finite number \( t \) of steps by \( M \), but the further point is that this simulation takes only some finite number \( n_0 \) of steps by \( M' \) simulating \( M \). Then for all inputs \( x \) of lengths \( n \geq n_0 \), \( M'(x) \) sees the acceptance and “panics” by not running in polynomial time—since this happens for almost-all \( x \) it definitely does not run in asymptotic polynomial time. Whereas if \( M \in D_{TM} \), then \( M' \) runs in linear time, hence in polynomial time. So the language \( PT \) of this problem is not c.e.

But we can formalize \( M \) running in polynomial time via the formal statement

\[
P_M = (\exists k)(\forall x)(\exists \vec{c}) : T(M, x, \vec{c}) \land |\vec{c}| \leq |x|^k.
\]

The set \( PT_F \) of provable cases of the sentence \( P_M \) (as a set of machine codes \( M \)) is c.e., since defined by the existence of a proof \( \pi \) such that \( R_F(P_M, \pi) \) holds. By the soundness of \( F \), \( PT_F \subseteq PT \); by \( PT_F \) being c.e. whereas \( PT \) is not, there must exist programs \( M \in PT \setminus PT_F \). Such programs run in polynomial time, but \( F \) cannot prove that they do so. [As I have mentioned in lecture, this kind of result created some ideas 45 years ago that \( P \neq NP \) might
be undecidable in natural systems $F$ such as first-order arithmetic (called PA for Peano Arithmetic), but those ideas were shown to lack power even before my doctoral thesis cleaned up this area in 1986.]

(V) Gödel’s Second Incompleteness Theorem in full says that an adequate and effective system of logic $F$ cannot prove the statement $(\forall \pi) \neg R_F(0 = 1, \pi)$, which expresses that $F$ is consistent, unless $F$ actually is inconsistent (and hence proves all statements it can formulate). Use this idea to create a Turing machine $M_F$ that runs in linear time if $F$ is consistent, but not otherwise. You can do this using the “delay switch” framework, but with “Search For Pie” in place of the “Simulate $M$ on its own code” as the action to do for $n$ steps, and with a time-wasting action in the “else:panic” branch.

Answer: The pseudocode for a new machine $M''$ is “similar but different” with regard to the answer to (IV):

```
input x;
int n = |x|;
Spend n steps /by this simulator/ exhaustively looking for a proof
    string "pie" such that $R_F(0=1',\text{pie})$ holds
if (no such proof is found) {
    halt;  //note this all took only $O(n)$ time
} else {
    //F got "pie" in its face, so we "panic" by running for more
    //than polynomial time
    do something that takes $2^n$ steps before halting
}
```

Now the reasoning is that by its adequacy for computation, $F$ can prove that: if $M''$ runs in linear time (or in polynomial time, or anything less than exponential time), then no proof $\pi$ in $F$ of a contradiction can exist. Hence, if $F$ can prove that $M''$ runs in linear time, then $F$ can prove that there is no such $\pi$—i.e., $F$ would be able to prove its own consistency. But this is precisely what Gödel proved that an adequate, effective, and consistent formal system cannot do. The difference from (IV) is that instead of just showing that unprovable cases of machines $M$ exist, we’ve actually coded such a machine directly.

(VI) Show that a language $L$ is c.e. if and only if $L$ is the range of a computable function $f$. Then show that $L$ is in NP if and only if $L$ is the range of a polynomial-time computable function $g$, where also there is a polynomial $q$ such that for all $x$ and $y$, if $g(x) = y$ then $q(|y|) \geq |x|$. This latter condition, called “polynomial honesty,” prevents $g$ from making strings exponentially shorter. For a topical example, consider the following function that takes a 3CNF formula $\phi$ and an assignment to it:

$$g(\langle \phi, a \rangle) = \begin{cases} 
\phi & \text{if } a \text{ satisfies } \phi \\
\phi_n & \text{otherwise}
\end{cases}$$

where $\phi_n$ is a fixed 3CNF formula of the same size as $\phi$ that is trivially satisfiable. Because $a$ cannot be longer than the encoding of $\phi$, $g$ gives linear honesty, and its range is 3SAT.
**Answer.** First, if $L = \text{Ran}(f)$ for some function $f$ computed by a Turing “transducer” $T$, then we can design an ordinary TM “acceptor” $M$ that on any input $y$ executes a “dovetail” loop:

```plaintext
define (t = 1,2,3,...) {
    foreach (x of length up to t) {
        if (T(x) == y) { accept y; }
    }
}
```

Then $L(M) = \text{Ran}(f) = L$, so $L$ is c.e. Conversely, if $L$ is c.e. and nonempty (a condition that is often omitted, as above), so that it contains a string $x_0$, then we can take a (not necessarily total) TM $M$ and make the following dovetail loop:

```plaintext
string last = x_0;
define (i = 1,2,3,...) {
    foreach (x of length up to i) {
        if (M accepts x within i steps) { last = x; }
    }
}
```

Then define $T(r)$ to run the dovetail loop for $r$ steps (not up to $i = r$ but meaning overall steps of the loop) and output the value of $\text{last}$ at that step. This recursively enumerates all the strings in $L(M)$ and creates a function $f$ computed by $T$ whose range is $L$. (If $x_0$ is the only string that $M$ accepts then $\text{last}$ stays equal to $x_0$ throughout and $f$ becomes a constant function, which is always computable.)

Now if $f$ is polynomial time computable and also meets the honesty condition with a polynomial $q$, then for all $y$,

$$y \in \text{Ran}(f) \iff (\exists x : x \leq q(|y|))[f(x) = y].$$

This is a valid NP-definition of $\text{Ran}(f)$. Conversely, if $A$ is a language in NP then we can take some verifying predicate $R(x,y)$ and polynomial $q$ such that for all $x, x \in A \iff (\exists y : |y| \leq q(|x|))R(x,y)$. Then define

$$f(z) = \begin{cases} x' & \text{if } z = \langle x', y' \rangle \text{ such that } R(x', y') \text{ holds} \\
? & \text{otherwise.} \end{cases}$$

Because witnesses $y'$ can be of length at most $q(|x'|)$, this $f$ meets the honesty condition with basically the same polynomial $q$, because with $y = x'$ in the above definition, $x = z$ has length $|\langle x, y \rangle| = q'(n)$ where $q'(n) \approx n + 1 + q(n)$ is likewise a polynomial.

The one hitch is what to put in for the ‘?’ in case of strings $z$ that don’t package an $x'$ and a witness $y'$. We can’t choose a single $x_0 \in A$ like before because as $z$ gets longer this will violate the honesty condition. Hence $A$ needs to include “easy strings $x_n$” at each length $n$ that we can know in advance. Or—and this is neater for both parts—we can interpret “computable” as meaning “partial computable” and then we may just not worry about this case.