Solutions for Week Ten

Self-Reference

i. What does the following program do?

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

    if (input == me) accept();
    else reject();
}
```

This program takes as input a string, then accepts it if the string is the program's own source code and rejects it otherwise. In other words, it's a program that only accepts itself.

ii. Consider the following program:

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

    if (input == "") accept();
    else if (input[0] == input[input.length() - 1]) accept();
    else reject();
}
```

What happens if we run the above program with input *abba*? Why?

This program will accept *abba* – it's not empty, but its first and last characters are the same.

iii. Let *p1* be a string containing the source of the above program. What will happen if we call *willAccept(p1, "abba")*? Why?

The call will return true, since the program accepts the input.

iv. Consider this program:

```
int main() {
    string input = getInput();
    string target = "";

    while (target != input) target += "a";
    accept();
}
```

What happens if we run the above program with input *abba*? Why?

The program will loop infinitely, since it will compare the input string *abba* against the strings ε, *a*, *aa*, *aaa*, *aaaa*, etc. and never find a match.

v. Let *p2* be a string containing the source of the above program. What will happen if we call *willAccept(p2, "abba")*? Why?

The call will return false, since the input program does not accept the specified string.

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

    if (willAccept(me, input)) {
        reject();
    } else {
        accept();
    }
}
```

vi. Suppose we feed the string *abba* as input to the above program. Explain why if *willAccept* says that the program accepts *abba*, then the program does not accept *abba*.

Suppose that *willAccept* says that this program accepts *abba*. In the first line, this program gets its own source code. In the second, it gets the input (*abba*). In the if statement, the condition will evaluate to true, so it then proceeds to reject *abba*. Since the program rejects *abba*, it does not accept *abba*.

vii. Suppose we feed the string *abba* as input to the above program. Explain why if *willAccept* says that the does not accept *abba*, then it does accept *abba*.

Suppose that *willAccept* says that this program does not accept *abba*. In the first line, this program gets its own source code. In the second, it gets the input (*abba*). In the if statement, the condition will evaluate to false, so the program then proceeds to accept *abba*.

viii. Explain why your answers to parts (vi) and (vii) collectively result in a contradiction that shows that A_{TM} is undecidable.

By our assumption that *willAccept* is a decider for A_{TM}, the results returned by *willAccept* is always accurate, which means that this program accepts *abba* if and only if it does not accept *abba*. This is impossible. The only way to resolve this issue is to notice that the *willAccept* method we assumed existed must not actually exist, so A_{TM} must be undecidable.

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

    if (willAccept(me, input)) {
        while (true) {
            // Do nothing
        }
    } else {
        accept();
    }
}
```

ix. Suppose we feed the string *abba* as input to the above program. Explain why if *willAccept* says that the program accepts *abba*, then it does not accept *abba*.

Suppose that *willAccept* says that this program accepts *abba*. In the first line, this program gets its own source code. In the second, it gets the input (*abba*). In the if statement, the condition will evaluate to true, so the program then proceeds to loop infinitely. Since the program loops on *abba*, it does not accept *abba*.

x. Suppose we feed the string *abba* as input to the above program. Explain why if *willAccept* says that the program does not accept *abba*, then it does accept *abba*.

Suppose that *willAccept* says that this program does not accept *abba*. In the first line, this program gets its own source code. In the second, it gets the input (*abba*). In the if statement, the condition will evaluate to false, so the program then proceeds to accept *abba*.

xi. Explain why your answers to parts (ix) and (x) collectively result in a contradiction that shows that A_{TM} is undecidable.

By our assumption that *willAccept* is a decider for A_{TM}, the results returned by *willAccept* is always accurate, which means that this program accepts *abba* if and only if it does not accept *abba*. This is impossible. The only way to resolve this issue is to notice that the *willAccept* method we assumed existed must not actually exist, so A_{TM} must be undecidable.

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

    if (willAccept(me, input)) {
        accept();
    } else {
        reject();
    }
}
```

xii. Suppose we feed the string *abba* as input to the above program. Explain why if *willAccept* says that the program accepts *abba*, then it does accept *abba*.

Suppose that *willAccept* says that program accepts *abba*. In the first line, this program gets its own source code. In the second, it gets the input (*abba*). In the if statement, the condition will evaluate to true, so the program then proceeds to accept *abba*.

xiii. Suppose we feed the string *abba* as input to the above program. Explain why if *willAccept* says that the program does not accept *abba*, then it does not accept *abba*.

Suppose that *willAccept* says that this program does not accept *abba* In the first line, this program gets its own source code. In the second, it gets the input (*abba*). In the if statement, the condition will evaluate to false, so the program then proceeds to reject *abba*.

xiv. Explain why your answers to parts (xii) and (xiii) do ***not*** prove that A_{TM} is decidable.

The fact that we didn't get a contradiction here doesn't mean that our assumption was correct! As shown by the other examples here, we can still use a decider for A_{TM} to cause all sorts of problem, so the fact that we didn't do this here doesn't say anything.

Self-Reference and Decidability

i. Suppose for the sake of contradiction that $L \in \mathbf{R}$. This means that we could write a function

```
bool acceptsAtLeastOneString(string program)
```

that accepts as input the source code of a program, then returns true if the program accepts at least one string and returns false otherwise. Write a self-referential program that uses this function to obtain a contradiction.

Here is one possible self-referential program we could write:

```
int main() {
    string me = mySource();
    if (acceptsAtLeastOneString(me)) reject();
    else accept();
}
```

This program asks whether it will accept at least one string. If so, it then rejects the input regardless of what it is, so it never accepts anything. Otherwise, the program accepts all inputs. Both cases contradict what the behavior of the program is supposed to be.

ii. Formalize your reasoning from part (ii) by writing a formal proof that $L \notin \mathbf{R}$. To do so, follow the proof template from lecture: assume that $L \in \mathbf{R}$, describe what that assumption entails, write a program that causes a contradiction, then explain why you get a contradiction in all cases.

Proof: Suppose for the sake of contradiction that $L \in \mathbf{R}$. This means that there is a decider D for L, which we can represent in software as a method `acceptsAtLeastOneString` that takes as input the source code of a program, then returns true if that program accepts at least one string and returns false otherwise. Given this, we could then construct the above self-referential program, which we'll call P.

Let's now trace through the execution of program P, focusing in particular on the answer given back by the call to `acceptsAtLeastOneString`. If P will accept at least one string, then `acceptsAtLeastOneString(me)`, will return true, at which point P rejects its input. Since the behavior of P in this case is independent of what string is provided as input, we see that P therefore rejects all strings, meaning that it doesn't accept at least one string.

Otherwise, P never accepts any strings. Therefore, the call to `acceptsAtLeastOneString(me)` will return false, at which point P accepts its input. Since the behavior of P in this case is independent of what string is provided as input, we see that P therefore accepts all strings, meaning that it accepts at least one string

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, we conclude that $L \notin \mathbf{R}$, as required. ■

iii. Prove that $L \in$ **RE** by writing pseudocode for either a recognizer or a verifier for L.

This is a little bit tricky, but the key insight is that if a machine M accepts at least one string, then that string has finite length *and* the machine accepts it after a finite number of steps. Based on that information, here is a possible verifier for L:

```
bool checkAcceptsAtLeastOneString(tm M, string w, int s) {
        simulate M running on w for s steps;
        return whether M is in an accepting state;
}
```

For clarity, we've written the program with three inputs, but in fact both w and the number of steps could be encoded together into a single certificate, i.e., the verifier is designed to accept inputs of the form $\langle M, \langle w, s \rangle \rangle$.

Observe that for any Turing machine M, $M \in L$ if and only if there is some certificate $\langle w, s \rangle$ such that our verifier accepts $\langle M, \langle w, s \rangle \rangle$: If $M \in L$, then our verifier will accept M if given a certificate containing one of the strings M accepts, and how many steps it takes to accept that string. Conversely, if there exists some certificate such that our verifier accepts $\langle M, \langle w, s \rangle \rangle$, then clearly M accepts w, so $M \in L$.

Furthermore, our verifier always halts, because all it does is run a simulation of a Turing machine for a finite number of steps. Thus, since we have exhibited a valid verifier for L, $L \in$ **RE**, as desired.

There are plenty of other solutions that could work. For instance, the verifier could just take a number of steps s as the certificate. Since a Turing machine run for s steps can read at most s characters of input, the verifier could just simulate M for s steps on *every* string of length at most s (of which there are a finite number), and accept $\langle M, s \rangle$ if M accepts any of these strings.

Notice however that it would not work for the verifier to just take a string w and check whether M accepts it, because then the language of the verifier would be A_{TM}, which we know to be undecidable.

Writing a recognizer for L is also tricky. The important thing is to make sure that you only simulate the machine M for a finite number of steps at a time. Otherwise, M might loop on some input, and you'll never get to find out if it would have accepted some other string you haven't tried yet. Here's one solution:

```
bool acceptsAtLeastOneString(tm M) {
        int s=0;
        while (true) {
                for (each string w of length at most s) {
                        simulate M running on w for s steps;
                        if (M is in an accepting state) return true;
                }
                s++;
        }
}
```

Review/Potpourri

Graphs

i. A *complete graph* is a graph G = (V, E) such that $\forall x \in V.\ \forall y \in V.\ (x \neq y \rightarrow \{x,y\} \in E)$. Prove by induction that a complete graph on *n* vertices has $n(n-1)/2$ edges, for any $n \in \mathbb{N}$.

Proof: Let P(n) be the statement "a complete graph on n vertices has n(n-1)/2 edges." We will prove, by induction, that P(n) is true for all $n \in \mathbb{N}$, from which the theorem follows.

For the base case, we need to show that P(0) is true, meaning that a complete graph on 0 vertices has 0(-1)/2 edges. Observe that any graph on 0 vertices has 0 edges. Since $0 = 0(-1)/2$, we see that P(0) is true.

For the inductive step, assume for some arbitrary $k \in \mathbb{N}$ that P(k) holds, meaning that a complete graph on k vertices has k(k-1)/2 edges. We need to show that P(k+1) holds, meaning that a complete graph on k+1 vertices has (k+1)k/2 edges. Let G = (V, E) be a complete graph on k+1 vertices, and consider an arbitrary $v \in V$. Define G' to be the graph obtained by removing v and all its incident edges from the graph G. Observe that G' is a complete graph on k vertices, so by our inductive hypothesis, we know that G' has k(k-1)/2 edges. The total number of edges in G is equal to the number of edges in G', plus the number of edges incident to v. Therefore, we see that

$$\begin{aligned} |E| &= k(k-1)/2 + k \\ &= k(k+1)/2 \\ &= (k+1)k/2, \end{aligned}$$

which is what we needed to show. Therefore, P(k+1) is true, completing the induction. ■

ii. Prove by induction that for any graph G = (V, E) and vertices $x,y \in V$, if there is a path from x to y then there is a *simple* path from x to y. (Hint: try inducting on some property of the path, rather than the graph itself.)

Proof: Consider an arbitrary graph G = (V, E). Let P(n) be the statement "for any vertices $x,y \in V$, if there is a path from x to y of length n, then there is a simple path from x to y." We will prove, by complete induction, that P(n) is true for all $n \in \mathbb{N}$, from which the theorem follows.

For the base case, we need to show that P(0) is true. Suppose that for some vertices $x,y \in V$, there exists a path p from x to y of length 0. A path of length 0 consists of a single vertex, so p is itself a simple path from x to y.

For the inductive step, assume for some arbitrary $k \in \mathbb{N}$ that P(0), P(1), …, P(k) hold. We need to show that P(k+1) holds. Suppose that for some vertices $x,y \in V$, there exists a path p from x to y of length k+1. We will consider two cases:

1. p is a simple path. In this case, the existence of a simple path from x to y holds trivially.

2. p is not a simple path, meaning that it contains repeated vertices. Let $p = v_0, v_1, \ldots, v_{k+1}$ and choose $i,j \in \mathbb{N}$ with $0 \leq i < j \leq k+1$ such that $v_i = v_j$. Now, consider the path $p' = v_0, v_1, \ldots, v_{i-1}, v_j, \ldots, v_{k+1}$. p' is a path from x to y with length k+1-(j-i), which is between 0 and k. Thus by our inductive hypothesis, there exists a simple path from x to y, which is what we needed to show.

Therefore in either case P(k+1) is true, completing the induction. ■

Regular Languages

Prove that each of the following languages is regular.

i. $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ and $L = \{ w \in \Sigma^* \mid$ all of the **a**'s in w appear in groups of 2 or more $\}$.

L is given by the following regular expression: $\mathbf{b}^*(\mathbf{aa}^+\mathbf{b}^*)^*$.

ii. $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ and $L = \{ w \in \Sigma^* \mid w$ does not contain the substring **ab** $\}$.

L is given by the following regular expression: $\mathbf{b}^*\mathbf{a}^*$.

iii. $\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ and $L = \{ w \in \Sigma^* \mid w$ does not contain the substring **ab** $\}$.

L is given by the following regular expression: $((\mathbf{b} \cup \mathbf{c})^*(\mathbf{a}^*\mathbf{c})^*)^*\mathbf{a}?$.

Context-Free Languages

Consider the alphabet $\Sigma = \{$ **a**, **b**, **ε**, **∅**, **∪**, *****, **(**, **)** $\}$ and the following language, which we'll call REGEXP:

$$\{ w \in \Sigma^* \mid w \text{ is a valid regular expression over } \{\mathbf{a},\mathbf{b}\} \}.$$

You may wish to refer back to the formal definition of regular expressions from lecture 17.

i. Give three examples of strings in REGEXP, and three examples of strings *not* in REGEXP.

((a*))**, **ε**, **∅∪∅∪∅** ∈ REGEXP, and **((a**, **∪***, **()** ∉ REGEXP.

ii. Prove or disprove: the language REGEXP defined above is regular.

REGEXP is not regular. The key insight is that the definition of REGEXP requires balancing parentheses, which automata and regular expressions cannot do.

Proof: Consider the set $S = \{ (^n \mid n \in \mathbb{N} \}$. This set is infinite because it contains one string for each natural number. Now, consider any strings $(^n$, $(^m \in S$ where $n \neq m$. Then $(^n\mathbf{a})^n \in$ REGEXP but $(^m\mathbf{a})^n \notin$ REGEXP. Consequently, $(^n \not\equiv_{\text{REGEXP}} (^m$. Therefore, by the Myhill-Nerode Theorem, REGEXP is not regular. ■

iii. Write a context-free grammar for REGEXP.

$S \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{ε} \mid \mathbf{∅} \mid S \cup S \mid S^* \mid (S)$

Consider the alphabet $\Sigma = \{ \mathbf{a}, \mathbf{\in}, \mathbf{,}, \mathbf{\{}, \mathbf{\}} \}$ and the following language, which we'll call ELEMOF:

$\{ w\mathbf{\in\{}\ell\mathbf{\}} \mid w \in \{\mathbf{a}\}^+$ and ℓ is a comma-separated list of strings in $\{\mathbf{a}\}^+$, at least one of which equals $w \}$.

Essentially this language contains all true "element-of" relations for finite sets of non-empty strings.

iv. Give three examples of strings in ELEMOF, and three examples of strings *not* in ELEMOF.

a∈{a}, **aa∈{aa,a,aa}**, **aaa∈{a,aa,aaa}** ∈ ELEMOF, and **∈{a}**, **a∈{a,,}**, **∈}** ∉ ELEMOF.

v. Prove or disprove: the language ELEMOF defined above is regular.

ELEMOF is not regular. The key insight is that the definition of ELEMOF requires "remembering" a string of arbitrary length, which automata and regular expressions cannot do.

Proof: Consider the set $S = \{ \mathbf{a}^n \mid n \in \mathbb{N} \}$. This set is infinite because it contains one string for each natural number. Now, consider any strings $\mathbf{a}^n, \mathbf{a}^m \in S$ where $\mathbf{a}^n \neq \mathbf{a}^m$. Then $\mathbf{a}^n\mathbf{\in\{}\mathbf{a}^n\mathbf{\}} \in$ ELEMOF but $\mathbf{a}^m\mathbf{\in\{}\mathbf{a}^n\mathbf{\}} \notin$ ELEMOF. Consequently, $\mathbf{a}^n \not\equiv_{\text{REGEXP}} \mathbf{a}^m$. Therefore, by the Myhill-Nerode Theorem, ELEMOF is not regular. ■

vi. Write a context-free grammar for ELEMOF.

Here's one possibility, where we use different nonterminals to keep track of whether we've generated the initial string yet. Here U is building a matching string on both sides of the **∈** symbol, while T and V build the comma-separated list to either side.

$S \rightarrow T\mathbf{\}}$	S corresponds to all strings in ELEMOF
$T \rightarrow T\mathbf{,}A \mid U$	T corresponds to strings of the form $\mathbf{a}^n\mathbf{\in\{}\ell$
$U \rightarrow \mathbf{a}U\mathbf{a} \mid \mathbf{a\in}V\mathbf{a}$	U corresponds to strings of the form $\mathbf{a}^n\mathbf{\in\{}\ell$ where the last string in ℓ is $\mathbf{a}^n$
$V \rightarrow VA\mathbf{,} \mid \mathbf{\{}$	V corresponds to strings of the form **{** or $\mathbf{\{}\ell$,
$A \rightarrow \mathbf{a}A \mid \mathbf{a}$	A corresponds to strings of the form $\mathbf{a}^n$

R vs. RE

For each of the following languages, determine whether it is (a) decidable, (b) recognizable but NOT decidable, or (c) not recognizable, and prove that your choice is correct. (Hint: we haven't seen very many proofs of non-recognizability, but see if you can adapt the ideas from the Guide to Self-Reference.)

i. $L = \{ \langle M_1, M_2 \rangle \mid M_1, M_2 \text{ are Turing machines and } L(M_1) \cap L(M_2) \neq \emptyset \}$

This language is recognizable but not decidable. Intuitively, you should be able to convince someone that $\langle M_1, M_2 \rangle$ is in the language by showing them a string accepted by both machines (and telling them how long to run the machines to see that it is accepted).

Proof: Consider the following verifier:

```
bool checkShareAtLeastOneString(tm M1, tm M2, string w, int s) {
    simulate M1 running on w for s steps;
    simulate M2 running on w for s steps;
    return whether M1 and M2 are both in an accepting state;
}
```

Observe that $\langle M_1, M_2 \rangle \in L$ iff there is some certificate $\langle w, s \rangle$ such that our verifier accepts $\langle \langle M_1, M_2 \rangle, \langle w, s \rangle \rangle$: If $\langle M_1, M_2 \rangle \in L$, then our verifier will accept $\langle M_1, M_2 \rangle$ if given a certificate containing one of the strings accepted by both machines, and how many steps they take to accept that string. Conversely, if there exists some certificate such that our verifier accepts $\langle \langle M_1, M_2 \rangle, \langle w, s \rangle \rangle$, then clearly both machines accept w, so $\langle M_1, M_2 \rangle \in L$.

Furthermore, our verifier always halts, because all it does is run simulations of two Turing machines for a finite number of steps. Thus, since we have exhibited a valid verifier for L, L is recognizable.

We will prove by contradiction that L is not decidable. Assume towards a contradiction that L is decidable, so there exists some decider for L, which we can represent in software as a method `shareAtLeastOneString` that takes as input the source code of two programs, then returns true if there is some string accepted by both programs and returns false otherwise. Given this, we could then construct the following self-referential program P:

```
int main() {
    string me = mySource();
    if (shareAtLeastOneString(me, programThatAlwaysAccepts)) reject();
    else accept();
}
```

Note that if P accepts any strings in Σ^*, then `shareAtLeastOneString(me, programThatAlwaysAccepts)` returns true, in which case P rejects all strings, meaning that it doesn't accept any strings in Σ^*.

Otherwise, P accepts no strings, so `shareAtLeastOneString(me, programThatAlwaysAccepts)` will return false, in which case P accepts all strings, meaning that it does accept a string in Σ^*.

In both cases we reach a contradiction, so our assumption must have been wrong. Therefore, we conclude that $L \notin \mathbf{R}$, as required. ■

ii. $L = \{ \langle M \rangle \mid M$ writes at least one non-blank symbol on the tape when given input ε $\}$

This language is decidable.

Proof: Consider the following decider:

```
bool writesOnTape(tm M) {
    set statesVisited = {};
    set up a simulation of M running on ε;
    while (true) {
        if (M's current state is in statesVisited) return false;
        simulate the next step of M running on ε;
        if (M wrote something to the tape) return true;
        add M's current state to statesVisited;
    }
}
```

Observe that $\langle M \rangle \in L$ iff *writesOnTape(M)* returns true: If *writesOnTape(M)* returns true then M wrote something on the tape, so $\langle M \rangle \in L$. Conversely, if $\langle M \rangle \in L$, then M must write something to the tape before it repeats a state twice; otherwise it will loop forever on an empty tape. Thus, *writesOnTape(M)* returns true.

Furthermore, our decider always halts, because all it does is run a simulation of a Turing machine for a finite number of steps (at most equal to the number of states in the machine). Thus, since we have exhibited a valid decider for L, L is decidable. ■

iii. $L = \{ \langle M \rangle \mid M$ is a Turing machine but NOT a decider $\}$

This language is not recognizable.

Proof: Assume towards a contradiction that L is recognizable, so there exists some Turing machine for L, which we can represent in software as a method *notADecider* that takes as input the source code of a program, then returns true if there is some string that causes the program to loop and either loops *or* returns false otherwise. Given this, we could then construct the following self-referential program P:

```
int main() {
    string me = mySource();
    if (notADecider(me)) reject();
    while (true) {};
}
```

Note that if P loops on some input, then *notADecider(me)* returns true, in which case P rejects all strings.

Otherwise, P always halts, so *notADecider(me)* will loop or return false, in which case P loops anyway.

In both cases we reach a contradiction, so our assumption must have been wrong. Thus, $L \notin$ **RE**, as required. ■

Closure Properties of RE

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 Turing machines M_1 and M_2 such that $\mathcal{L}(M_1) = L_1$ and $\mathcal{L}(M_2) = L_2$, then write pseudocode for recognizing whether a given input string is in $L_1 \cap L_2$. Then, briefly justify why your construction is correct.

Proof: Consider the following pseudocode:

```
bool L1uL2(string w) {
    simulate M1 running on w;
    simulate M2 running on w;
    if (M1 and M2 are both in accepting states) return true;
    return false;
}
```

Observe that $w \in L_1 \cap L_2$ if and only if our code returns true: If $w \in L_1 \cap L_2$, then both simulations will accept, so our code returns true. If $w \notin L_1 \cap L_2$, then at least one of the simulations either loops forever or rejects; either way, our code will not return true. Thus we have constructed a Turing machine that recognizes $L_1 \cap L_2$. ■

ii. Repeat part (i), except proving that the **RE** languages are closed under union.

Proof: Consider the following pseudocode:

```
bool L1uL2(string w) {
    set up a simulation of M1 running on w;
    set up a simulation of M2 running on w;
    while (true) {
        simulate the next step of M1 running on w;
        simulate the next step of M2 running on w;
        if (either M1 or M2 are in an accepting state) return true;
    }
}
```

Observe that $w \in L_1 \cup L_2$ if and only if our code returns true: If $w \in L_1 \cup L_2$, then eventually one of the simulations will accept, so our code returns true. If $w \notin L_1 \cup L_2$, then neither simulation will ever accept, so our code loops. Thus we have constructed a Turing machine that recognizes $L_1 \cup L_2$. ■

Verifiers

In class, we proved that the diagonal language L_D is not recognizable. Explain why the following program is NOT a verifier for L_D.

```
bool checkLD(tm M, int c) {
    run M on ⟨M⟩ for c steps;
    if (M is in a rejecting state) accept();
    else reject();
}
```

Consider a machine M which loops on the input ⟨M⟩. By definition, ⟨M⟩ ∈ L_D, but no matter what certificate c we provide to the proposed verifier, the verifier will always reject.

Complexity Theory

For each of the following statements, determine whether it is true or false and justify your answer.

i. For every language L, if $L \in \mathbf{P}$ then $L \in \mathbf{REG}$.

False. We proved in class that $\{ a^n b^n \mid n \in \mathbb{N} \}$ is not regular, but you could write a linear-time program to count the a's and b's to check that they are equal.

ii. For every language L, if $L \in \mathbf{P}$ then $L \in \mathbf{R}$.

True. If there is a polynomial-time decider for L, then there is a decider for L.

iii. For every language L, if $L \in \mathbf{P}$ then $L \in \mathbf{RE}$.

True by part (ii) and the fact that $\mathbf{R} \subseteq \mathbf{RE}$.

iv. For languages A and B, if $A \leq_p B$ and $B \in \mathbf{P}$ then $A \in \mathbf{NP}$.

True. We learned in class that if $A \leq_p B$ and $B \in \mathbf{P}$ then $A \in \mathbf{P}$, plus we know $\mathbf{P} \subseteq \mathbf{NP}$.