1) Use the pumping lemma to show that the following language is not context free:

\{0^n#0^{2n}#0^{3n} \mid n \geq 0\}

If the given language were context-free, letting \( p \) be the pumping length, we could express \( 0^p#0^{2p}#0^{3p} \) as \( uvxyz \) with \(|vxy| \leq p\) and \(|v| + |y| > 0\). If either \( v \) or \( y \) contains a #, then \( uv^2xy^2z \) isn’t in \( L \) since it would have the wrong number of #s. Thus, \( v \) only has 0s or \( y \) only has 0s. No matter how these are distributed, these 0s must lie in either 1 or 2 “components” exactly of the string. Thus, the string \( uv^2xy^2z \) must take on one of the following forms: \( 0^{p+i}#0^{2p}#0^{3p} \) or \( 0^p#0^{2p+i}#0^{3p} \) or \( 0^p#0^{2p}#0^{3p+1} \) or \( 0^{p+i}#0^{2p+1}#0^{3p} \) or \( 0^{p+i}#0^{2p}#0^{3p+j} \) or \( 0^p#0^{2p+i}#0^{3p+j} \), where \( i \) and \( j \) are positive and less than or equal to \( p \). But, none of these strings is in \( L \), thus, there’s no valid way to pump this string and the string doesn’t satisfy the pumping lemma for context-free languages. It follows that the given language is not context-free.

2) Use the pumping lemma to show that the following language is not context free:

\{0^n \mid n \in \text{Primes}\}

Assume to the contrary that \( L \) is context-free. Then, there exists a pumping length for \( L \), let this be \( A \). Consider the string \( s = 0^B \), where \( B \) is a prime number greater than \( A \). (Remember that there are an infinite number of prime numbers, so we are guaranteed that \( B \) exists.)

According to the pumping lemma for context free languages, there exists a way to split \( s = uvwxy \) such that for all non-negative integers \( i \), \( uv^iwy^i z \in L \) with \(|vy| > 0\) and \(|vxy| \leq A\).

Note that the length of the string \( uv^iwy^i z \) is \(|s| + |vy|^i|\text{(i-1)}|\) and that the length of \( vy \) may be any positive integer less than or equal to \( A \). Consider the case where \( i = B + 1 \), then the length of the pumped string is \(|s| + |vy|^i (B + 1 - 1)\). Remember that \(|s| = B \), so now we have:

\( B + |vy|^i B = B(1 + |vy|) \).

Remember that \(|vy| > 0\), so \( 1 + |vy| > 1 \). Thus, we’ve factored the length of the pumped string into two factors, meaning that this string is NOT in the language, which contradicts the pumping lemma for Context Free Languages.
3) Show that Turing-decidable languages are closed under the following operations:

a) union  
b) intersection  
c) complementation  
d) concatenation

Let L and M be two Turing-decidable languages. Then there exist two Turing Machines that decide membership in these languages. Let these be A and B, respectively for the remainder of this problem. In each of the four parts, we'll describe how to make a decider for the specified language.

a) To build a TM that decides membership in L union M, simply run A on the given input, followed by running B on the given input. Since A and B are guaranteed to halt, these tasks will complete. After finishing, if either accepted, accept, otherwise reject.

b) Do the same steps as in part (a), but at the very end, accept only if both simulations accepted. Reject otherwise.

c) Here we explain how to decide membership in L complement: Simply run A on the given input. Whatever answer A returns, simply return the opposite.

d) Given the input string, s, of length n, loop through all possible n+1 partitions of the string, s = uv. For each of these partitions, simulate u as input to A and v is input to B. If both simulations accept, accept and halt. If, after this process, the machine hasn’t accepted, reject.

4) Show that Turing-recognizable languages are closed under union and intersection. Why is it necessary to be more clever with these two proofs than those in question number 3?

Let L and M be two Turing-recognizable languages. Then there exist two Turing Machines that recognize these languages. Let these be A and B, respectively for the remainder of this problem.

Now, we describe how to build a TM C that recognizes the union of L and M. Use a multi-tape machine that is capable of simulating the steps of both A and B. In particular, use two work tapes, copying the original input to both. Then, run one step of machine A on the first tape, followed by one step of machine B on the second tape, and then continue alternating in this fashion. If either simulation accepts, accept.

To build a TM that recognizes the intersection of L and M, we can actually simulate one followed by the other, just like we did in question #3. This is because if the input string isn’t in the first language (ie. never gets accepted by A), it can’t be in L intersect M and we have no need to run the simulation of B.
In question #3, since machines A and B are deciders, we can be confident that any simulation of those machines will halt, so we can just run any simulation we want till completion and continue to future simulations. In this question, it was imperative that we carry out both simulations for the union case. Thus, we couldn’t wait for one simulation to finish before starting the other. In the intersection case, no extra care needs to be taken due to the nature of the problem.

5) Find a match in the following instance of the PCP: \( \{ [\frac{ab}{abab}], [\frac{b}{a}], [\frac{aba}{b}], [\frac{aa}{a}] \} \).

Here is one solution: \( [\frac{ab}{abab}] [\frac{ab}{abab}] [\frac{aba}{b}] [\frac{b}{a}] [\frac{aa}{a}] [\frac{aa}{a}] \).

In this arrangement, both the top and bottom read, “ababababaaaa”.

6) Show that \( A_{TM} \) is not mapping reducible to \( E_{TM} \).

Let’s assume to the contrary of what we are trying to prove, that \( A_{TM} \leq_m E_{TM} \). If A is mapping reducible to B, then \( \overline{A} \) is mapping reducible to \( \overline{B} \). Applying this theorem to our assumption, we get \( \overline{A}_{TM} \leq_m \overline{E}_{TM} \).

It is also the case that \( \overline{E}_{TM} \) is Turing-recognizable. To see this, consider building a recognizer for \( \overline{E}_{TM} \). We would simply try running each string of length 1 for 1 step on M, then each string of length 2 or less try running M for 2 steps, etc. If there is any string, \( w \), that is accepted by M, this procedure would eventually recognize that fact and correctly accept the machine M.

According to theorem 5.22 in the text if \( A \leq_m B \) and B is Turing-recognizable, then A is as well. Applying this Theorem to \( A_{TM} \leq_m E_{TM} \) and using the fact that \( \overline{E}_{TM} \) is Turing-recognizable, we arrive at the conclusion that \( \overline{A}_{TM} \) is Turing-recognizable.

But, at the end of chapter 4, we proved that \( \overline{A}_{TM} \) is NOT Turing-recognizable. This is our contradiction. Thus, it follows that our initial assumption was incorrect. We conclude that \( A_{TM} \leq_m E_{TM} \) is false.

Note: To see that \( \overline{A}_{TM} \) is NOT Turing-recognizable, note that \( A_{TM} \) is NOT decidable, but is Turing recognizable. Also, note that if L is Turing-recognizable and \( \overline{L} \) is Turing-recognizable, then L is decidable. This is because if we had recognizers for both L and \( \overline{L} \), we could build a decider for L by running both of our recognizers in alternate steps (one step for L, one step for \( \overline{L} \)) and we’d be guaranteed that one of the two would accept and halt. We can then use that to decide L.
7) Let $S = \{ <M> | M \text{ is a TM that accepts } w^R \text{ whenever it accepts } w \}$. Show that $S$ is undecidable.

Let’s assume the contrary, that $S$ is decidable. We then assume that a Turing Machine, $T$, decides the language $S$. We show how to build a Turing Machine, $M$, that decides $A_{TM}$, under this assumption.

Our goal, given a Turing Machine $M$ and a string $w$, is to decide whether or not $M$ accepts $w$. We will create a new Turing Machine $M'$ as follows:

$M'$ reads its input. If the input is 01, $M'$ accepts. If the input is 10, $M'$ will erase its input and write $w$ on its tape and then simulate the steps of $M$ on $w$. If the input is anything else, $M'$ will automatically reject. This, the language of $M'$ is either $\{01\}$ or $\{01, 10\}$. It is the former if $M$ does NOT accept $w$, and it is the latter if $M$ does accept $w$.

Thus, once we construct $M'$, we make the call $T(M')$, which is a decider. If $T$ accepts, then we accept. If $T$ rejects, then we reject. (If the language of $M'$ does not satisfy the requirement, that means that it does not accept 10 and that the original $M$ doesn’t accept $w$. On the other hand, if the language of $M'$ does satisfy the requirement, that means $M'$ accepts 10 which means $M$ accepts $w$.)

8) Consider the problem of testing whether a Turing machine $M$ on an input $w$ ever attempts to move its head left at any point during its computation on $w$. Formulate this problem as a language and show that it’s decidable.

Let $L = \{ <M, w> | \text{During } M \text{'s computation on } w, M \text{ moves the tape head left} \}$

We will build a multi-tape TM that decides this language to prove that it is decidable.

Simulate $M$ running on $w$ for $|w|$ steps or until a left move occurs. If a left move occurs first, accept and halt. If the machine halts without ever moving left, reject. Alternatively, we can assume that we’ve moved R on each of $|w|$ moves and have read in each character of the input. The next step will be reading in the first blank at the right of the input. From this point on, continue the simulation, keeping track of what state the machine $M$ is in and store a list of states visited after reaching the end of the input. We know that if all of these moves are right moves, that the only input character will be a blank. If we ever hit a left move, we accept and halt. If we ever hit a repeated state (which MUST happen in $|Q|$ moves), we can guarantee that the machine will loop and never go left. At this point, we reject and halt.

The key with making this decision is that for machines that never move left, eventually we end up just reading in blanks and nothing else. Once we enter this restricted “state”, we CAN detect looping behavior in a finite amount of time.
9) Show that the PCP is decidable over a unary alphabet, that is, over the alphabet $\Sigma = \{1\}$.

An algorithm to decide PCP for a unary alphabet: Clearly, if any tile has the same number of 1s on the top and bottom, an arrangement of tiles exists. Alternatively, if all of the tiles have more 1s on the top than bottom, then no arrangement exists. Similarly, if all the tiles have more 1s on the bottom than top, then no arrangement exists. (The proof in the first case is simply that for any number of tiles greater than 0 the top MUST have more 1s than the bottom has.) Finally, consider a case where at least one tile has more 1s on the top and a different tile has more 1s on the bottom. Let the first tile have $x$ more 1s on the top and the second tile have $y$ more 1s on the bottom. Place $y$ of the first tile, so we’ve accumulated $x \cdot y$ more 1s on the top. Then, place $x$ of the second tile. These last tiles have $y \cdot x$ more 1s on the bottom. Counting, we see that this arrangement must have an equal number of 1s on both the top or bottom.

10) Show that the PCP is undecidable over a binary alphabet, that is, over the alphabet $\{0, 1\}$.

Assume to the contrary, that the PCP problem over the binary alphabet IS decidable. Now, we will use this result to show that the PCP problem in general is decidable, to achieve our contradiction. Consider an arbitrary instance of PCP with an alphabet size of $k$, where $k \leq 2^m$, for some constant $m$. Simply replace each alphabet letter with a unique $m$-bit code of 0s and 1s, and then replace each tile by substituting each letter with the assigned $m$-bit code.

For example, if the tiles in the original PCP problem are $\{\text{a}, \text{b}, \text{c}, \text{d}\}$, and we assigned the codes $a = 00$, $b = 01$, $c = 10$, and $d = 11$, then our new tiles would be $\{0010, 1101, 00100011, 11110001\}$.

Now, we will prove that this new instance of PCP with an alphabet of size 2 has a solution, if and only if the original instance of PCP has a solution.

Clearly, if the original instance of PCP has a solution, the new one does, since each code was a one-for-one substitution. What we have to be weary of, is a situation where the original tiles DIDN’T have a solution, but where our new tiles do. We will show that this never happens by showing that if there is a solution in the new instance of PCP, that corresponds to a solution in the old instance.

Consider any solution in the new instance of PCP. Place those tiles out. Then, simply replace each set of $m$ bits in each of these tiles with the letters from the original instance. This will be your solution from the original version of the problem.

Now, it’s important to note why this proof doesn’t work for a unary alphabet. If we tried it for a unary alphabet, we would be FORCED to apply variable length codes, like $a = 1, b = 11, c = 111, d = 1111$. Then, the matching becomes ambiguous when you try to go backwards. Does 1111 = d, or does it equal ac, or does it equal bb? With the binary alphabet, I can enforce a fixed length code, so that there’s a unique way to translate back from 0s and 1s to the old alphabet.