Problem 1 (a). We wish to show that the halting problem for deterministic linear bounded automata is decidable. In other words, that the set

\[ \{ M\#x : \text{LBA } M \text{ halts on input } x \} \]

is recursive. Using the result from Problem 4(a) on the previous homework, this is easy. We build a universal Turing machine \( T \) such that when \( T \) is given \( M\#x \) as input, it simulates the linear bounded automata \( M \) on input \( x \). Before the simulation begins, \( T \) calculates an upper bound \( c \) on the number of possible configurations that \( M \) can take (as in the previous homework), and stores this value on a separate tape. On yet another tape, \( T \) will increment a counter for each step of the simulation and compare it to \( c \). If the number of steps taken ever exceeds \( c \), then we are certain that \( M \) enters an infinite loop on input \( x \), so \( T \) halts and rejects. If, however, the simulation of \( M \) halts before this point, \( T \) halts and accepts.

Problem 1 (b). Let \( \text{LBA}_k \) represent the LBA with input alphabet \( \{0, 1\} \) whose encoding is the binary number \( k \). Now let’s build a Turing machine \( T \) that, on input \( x \), simulates \( \text{LBA}_x \) on input \( x \). Whether \( T \) accepts or rejects will depend on what \( \text{LBA}_x \) does on input \( x \):

- If \( \text{LBA}_x \) accepts on input \( x \), then \( T \) halts and rejects.
- If \( \text{LBA}_x \) rejects on input \( x \), then \( T \) halts and accepts.
- If \( \text{LBA}_x \) loops on input \( x \), then \( T \) halts and accepts.

We can detect whether \( \text{LBA}_x \) loops on input \( x \) because we just showed that the halting problem for LBAs is decidable.

Now, it should be clear that \( x \in L(T) \Leftrightarrow x \notin L(\text{LBA}_x) \). Since \( T \) is total (it halts on all input), \( L(T) \) is a recursive set. However, it is not accepted by any LBA because for any given LBA, say \( \text{LBA}_y \), there is at least one string for which \( L(T) \) and \( \text{LBA}_y \) disagree, namely the binary string \( y \).
Problem 9.3.7. (b) Let \( L_b = \{(M_1, M_2) | L(M_1) \cap L(M_2) = \emptyset \} \). Show that \( L_b \) is not recursively enumerable.

We use the fact that \( L_e = \{M | L(M) = \emptyset \} \) is non-recursively enumerable. Suppose there were a Turing machine \( M \) that accepts \( L_b \). Suppose we’re given a Turing machine \( N \), and we wish to decide if \( L(N) = \emptyset \). Then we can give \((N, N)\) as input to \( M \). Since \( L(N) \cap (N) = N \), this effectively decides whether \( L(N) = \emptyset \). This gives a reduction from \( L_e \) to \( L_b \), and since \( L_e \) is non-RE, \( L_b \) must also be non-RE.

(c) Let \( L_c = \{(M_1, M_2, M_3) | L(M_1) = L(M_2) \cdot L(M_3) \} \). Show that \( L_c \) is not recursively enumerable.

Suppose there is a TM \( M \) which accepts \( L_c \). Let \( T \) be a TM such that rejects on every input, so \( L(T) = \emptyset \). Take any TM \( N \), and let \( M \) decide whether \((N, N, T) \in L_c \). This evaluates whether \( L(N) = L(N) \cdot L(T) \), and since \( L(T) = \emptyset \), we have \( L(N) \cdot \emptyset = \emptyset \), so \( M \) decides whether \( L(N) = \emptyset \). This gives us a reduction from \( L_e \) to \( L_c \), and since we know that \( L_e \) is not RE, we conclude that \( L_c \) is also not RE.
To compute the gcd, we basically repeatedly reduce one number by the other, alternating between the two, until one of them is zero. Our machine will switch back and forth between considering the left side as being the number $m$ or the number $n$.

Start with the left side being $m$.

Repeat:

1. If $n = 0$ (i.e., $n = a^0$), we can stop. (If we wanted, we can also remove the # symbol from the tape and shift the rest of the symbols as necessary so that what’s left on the tape is $a^{\gcd(m,n)}$.)

2. While $m \geq n$, remove $n$ symbols from $m$. More explicitly, we compare the two sides by marking off one symbol on each side until one side is completely marked off. If $n$ was the side that got completely marked off, we delete the marked symbols from the $m$ side, shifting the rest of the symbols as necessary. We clear the marks from the $n$ side, and do this again. When we are finally unable to mark off all symbols on the $n$ side, we clear the marks and go to the next step.

3. Switch what we consider to be the $m$ and $n$ sides, and go to the next iteration.
Problem 4

c) Concatenation – Build a TM which given a string w, splits it in such a way that the first part is in $L_1$ and the second is in $L_2$. To do this, we have the TM non-deterministically split w into xy, run $M_1$ on x and $M_2$ on y. For recursive languages, the TM will terminate (both in the splitting process and $M_1$ and $M_2$ execution). For REs, the TM is guaranteed to stop if both x and y are accepted by $M_1$ and $M_2$, which is what we want.

d) Kleene Closure – For memory, $L^* = \bigcup_{i \in \mathbb{N}} L_i$. We then proceed like in part c). First, we non-deterministically split the input w into $w = w_1w_2w_3...w_k$, then we simulate $M$ on each of the $w_i$. If all accept, we accept. The non-deterministic splitting guessed the decomposition. Similarly, we can adapt the deterministic, enumerative version.

f) Inverse Homomorphism – Both recursive and RE languages are closed. Consider the following TM: It takes input w. It should tell whether $w \in h^{-1}(L)$. To do this, we first compute $h(w)$. Now, test if $h(w) \in L$. If it is, then $w \in h^{-1}(L)$, if not, it isn’t.