1. Let $\Sigma = \{0, 1\}$. Recall we have defined enumeration orderings of $\Sigma^*$, such as

$$e, 0, 1, 00, 01, 10, 11, 000, 001, 010, \cdots$$

In this ordering, all strings of length $k$ come before strings of length $k + 1$, $k \geq 0$, and strings of the same length are ordered by their binary values. Prove that a language $L$ over $\Sigma$ is decidable (recursive) if and only if there exists an enumerator TM for $L$ which enumerates the members of $L$ in the order that is consistent with the above ordering of $\Sigma^*$. (Note the if and only if, i.e., two directions are to be proved.) For example, if $L = \{w \mid w$ begins and ends with the same symbol $\}$ then the enumerator must output the members of $L$ in the ordering

$$e, 0, 1, 00, 11, 000, 010, 101, 111, \cdots$$

**ANSWER** First suppose an ordered enumerator $M_o$ for $L$ exits. Then as we did for a recognizing TM, we design a decider $M$ for $L$ that on input $w$, calls $M_o$ to enumerate $L$. $M$ compares $w$ to each string $M_o$ generates as before, and halts in $q_{\text{accept}}$ if any matches $w$. However, now it also halts in $q_{\text{reject}}$ if $M_o$ generates any string greater than $w$ (before $w$ is accepted) in the ordering --- clearly easy to do. If generated string $g$ and $w$ are of same length look at leftmost bit on which they differ and reject if this bit is a 1 in $g$ and a 0 in $w$. Any string $g$ longer than $w$ that $M_o$ generated follows $w$ in the ordering and also causes $M$ to reject $w$. (In fact, it is fine to let $M_o$ continue to enumerate until a $g$ longer than $w$ is generated without worrying about the test for exact position in the ordering.) Hence, $M$ is able to halt whether or not $w$ is in the language $L$. Note that if $L$ is finite (and thus possibly the "last" string in $L$ comes before $w$ in the ordering), we simply assert that all finite languages are recursive (and not use $M_o$ in our proof in this case).
Now suppose M is a decider for L. We can construct an ordered enumerator Mo. As argued previously in class, Mo can generate the strings in Sigma* in the order given above. On each string g, Mo calls decider M. There is no dovetailing here, since M will return Yes or No in all cases. Hence, the Yes strings (strings in L) are output by Mo in the appropriate order.

2. In lecture, we gave the result that a $k$-tape DTM $M_k$ could be simulated by a 1-tape DTM $M$ in such a way that if $T(n)$ bounds the number of steps taken by $M_k$ on any input of length $n$, $M$’s time complexity is bounded by a function which is on the order of $(T(n))^2$. Here, $k$ is a fixed integer constant $k \geq 1$, and we assume $M_k$’s complexity function obeys $T(n) \geq n$ for all $n \geq 0$. Consider the following rough argument intended to show that $M$ always can perform the simulation in a time bound that is on the order of $n \cdot T(n)$. If you believe the argument can be turned into a proof that an order $n \cdot T(n)$ simulation is always possible, supply the details. On the other hand, if you do not believe that the bound of $n \cdot T(n)$ is always possible, identify the key reason(s) that such a simulation is not always possible.

**Argument:** $M$ simulates $M_k$ on input $w$ of length $n$. $M$ partitions its single tape into $k$ logical segments, marking them off with ##. Every move $M_k$ makes at worst requires $M$ to traverse up to $c$ times, for some constant $c$ depending only on $M_k$ and not on the input, its tape, which is divided into $k$ segments, and whose length is bounded by $h \cdot n$ symbols, where $h$ is some constant also depending only on $M_k$. That is, for each simulated move of $M_k$, $M$ can gather the required information from its tape in some constant number $c$ traversals of its single linear tape. $M$ can also rewrite its tape with shifting if necessary within this time bound. Hence, each simulated $M_k$ step requires of $M$ no more than on the the order of $c \cdot h \cdot n$ steps. Since $M_k$ requires no more than $T(n)$ steps, the bound is on the order of $c \cdot h \cdot n \cdot T(n)$ steps, i.e. is order $n \cdot T(n)$. 
ANSWER:
In \( T(n) \) steps of computation, the total number of symbols on the tapes can grow from an input size of \( n \) to \( T(n) \). Hence, the 1 tape machine is going to have to be simulating the \( k \) tape machine on \( T(n) \) symbols distributed across the \( k \) tapes (not just the original \( n \) symbols), so the simulation method now requires \( T(n) \times T(n) \) steps. EG, in the first \( T(n)/2 \) steps the total tape lengths grows to \( T(n)/2 \). The next \( T(n)/2 \) steps of \( M_k \) require on the order of \( T(n)/2 = O( (T(n))^2 ) \) steps each to simulate, making it order \( (T(n))^2 \). Note, in particular, that if no tape ever grows beyond \( n \) symbols, the above argument would be correct. Hence, a correct solution must identify tape growth as the issue.

3. We have previously demonstrated that strings over \( \Sigma \) and encodings of Turing machines can be enumerated and numbered. For example, Question 1 above gives an enumeration and numbering order for \( \Sigma^* \), and the proof of the undecidability of the Halting problem utilizes an enumeration and numbering of the encodings of Turing machines. Similarly, encodings of linear bounded automata (LBA’s) can be enumerated and numbered. Hence, we may refer to \( M_i \) as the \( i^{th} \) LBA in the enumeration, and \( x_i \) as the \( i^{th} \) member of \( \Sigma^* \) in its enumeration.

Consider the language

\[
L = \{ x_i \mid x_i \text{ is not in the language } L(M_i) \text{ of the } i^{th} \text{ LBA } M_i \}.
\]

It is the case that \( L \) is decidable by a Turing machine \( M \). Given input string \( y \in \Sigma^* \), \( M \) can determine \( y \)’s numbering \( i \) in the enumeration of \( \Sigma^* \) (i.e., \( y \) is \( x_i \)) and then create and simulate the LBA \( M_i \) and decide if \( M_i \) accepts \( y \) or not.

Prove that the language \( L \) cannot be the language \( L(M_j) \) of any LBA in the enumeration \( M_1, M_2, \ldots, M_j, \ldots \) of all LBA’s. What does this result imply about \( L \)’s placement in the language hierarchy?
ANSWER Suppose to the contrary, that \( L \) is \( L(M_j) \), for some \( j \). Suppose that \( x_j \) is accepted by LBA \( M_j \). Hence \( x_j \) is in \( L(M_j) \), and then it is not in \( L \), by definition of \( L \). So suppose \( x_j \) is not accepted by LBA \( M_j \). Hence \( x \) is not in \( L(M_j) \), and hence it is in \( L \), by definition of \( L \). Either way, we have a contradiction, so the assumption that \( L \) is \( L(M_j) \), for some \( j \), is false. Thus \( L \) is not the language of any LBA. Thus \( L \) is recursive but not CSL.

4. Consider the following proposed alternative proof that the language

\[
\text{REGULAR}_{TM} = \{ < M > \mid < M > \text{ encodes a Turning machine such that } L(M) \text{ is regular} \}
\]

is undecidable.

We show that if \( \text{REGULAR}_{TM} \) could be decided then \( A_{TM} \) could also be decided. Let \( M_{rtm} \) be a decider for \( \text{REGULAR}_{TM} \) and construct as follows a decider \( M_{tm} \) for \( A_{TM} \). First, we note that each of the following two TM’s, \( M_b \) and \( M_c \), clearly exist and that the encoding of either clearly can be created by \( M_{tm} \).

\( M_b \): Is a Turing machine that accepts every input. Hence,

\[
L(M_b) = \Sigma^*, \text{ which is regular.}
\]

\( M_c \): Is a Turing machine that accepts a string exactly when it is of the form \( 0^n \ 1^n \), for some \( n \geq 0 \). Hence,

\[
L(M_c) = \{ 0^n \ 1^n \mid n \geq 0 \}, \text{ which is not regular.}
\]

Now, on input \( < M, w > \), \( M_{tm} \) creates an encoding of one these two machines. If \( w \) is accepted by the TM \( M \), \( M_{tm} \) creates \( < M_b > \), and if \( w \) is not accepted by TM \( M \), \( M_{tm} \) creates \( < M_c > \). \( M_{tm} \) then passes the encoding it has created to \( M_{rtm} \). If \( M_{rtm} \) returns YES, this means \( M_{tm} \) passed it the encoding of \( M_b \) and hence, since this means \( w \) is
accepted by $M$, $Matm$ accepts its input $< M, w >$. If $Mrtm$ returns NO, this means $Matm$ passed it the encoding of $Mc$ and hence, since this means $w$ is not accepted by $M$, $Matm$ rejects its input $< M, w >$.

How does this proposed proof differ from that in Sipser and given in lecture, and why is it not a valid proof?

ANSWER The proof in Sipser has $Matm$ describe a machine $M_R$ that simulates $M$ on $w$; this is possible to do. The above requires the construction of the description of $M_R$ ($Mb$ or $Mc$) based on $Matm$ knowing the answer to the undecidable question. Hence, the actions required of $Matm$ are not possible.

5. Let $L_1$ and $L_2$ be languages over $\Sigma$ and suppose we have a total, computable function $f$ mapping from $\Sigma^*$ to $\Sigma^*$ such that

$$w \in L_1 \iff f(w) \in L_2.$$  

Hence, we say $L_1 \leq_m L_2$.

a) Suppose we know $L_2'$ (the complement of $L_2$) is decidable. Can we conclude $L_1$ is decidable? Support your answer.

ANSWER Yes. Simply observe that $L_2$ complement decidable implies $L_2$ decidable, and hence the result follows from standard definition of $\leq_m$.

OR

Let $M'$ be a decider for $L_2$ complement. Decider $M$ for $L_1$ applies $f$ to input $w$, then calls $M'$ on $f(w)$. Since $M'$ is a decider, it always answers YES or NO. If $M'$ answers YES, $f(w)$ is in $L_2$ complement and thus not in $L_2$ and thus $w$ is not in $L_1$, so $M$ answers NO. If $M'$ answers NO, $f(w)$ is not in $L_2$ complement and hence is in $L_2$ and hence $w$ is in $L_1$. 

and M answers YES.

b) Suppose we know \( L_2' \) (the complement of \( L_2 \)) is recognizable. Can we conclude \( L_1' \) (the complement of \( L_1 \)) is recognizable? Support your answer.

ANSWER Yes. Simply observe that \( f \) also has the property that
\[ w \text{ not in } L_1 \text{ iff } f(w) \text{ not in } L_2, \text{ hence} \]
\[ w \text{ in } L_1 \text{ complement iff } f(w) \text{ in } L_2 \text{ complement} \]

Hence, we have

\[ L_1 \text{ complement} \leq_m L_2 \text{ complement} \]

OR

Let \( M' \) be a recognizer for \( L_2 \) complement. Recognizer \( M \) for \( L_1 \) complement applies \( f \) to input \( w \), then calls \( M' \) on \( f(w) \). If \( w \) is in \( L_1 \) complement it is not in \( L_1 \) and hence \( f(w) \) is not in \( L_2 \) and hence \( f(w) \) is in \( L_2 \) complement. Since \( M' \) recognizes \( L_2 \) complement it must return YES on \( f(w) \) and hence \( M \) returns YES on \( w \). If \( w \) is not in \( L_1 \) complement it is in \( L_1 \) and hence \( f(w) \) is in \( L_2 \) and hence \( f(w) \) is not in \( L_2 \) complement. Recognizer \( M' \) for \( L_2 \) complement thus will either return NO or loop on \( f(w) \). Recognizer \( M \) for \( L_1 \) complement thus can do the same on such \( w \).

c) Is \( \{f^{-1}\} \) (the inverse of function \( f \)) necessarily a valid function that shows

\[ L_2 \leq_m L_1 \]?

Support your answer.

ANSWER No. \( f \) might not even have an inverse. \( f \) is not guaranteed to be either onto nor 1-1. For example, \( f \) could map each member of \( L_1 \) to the same member of \( L_2 \), and each member of \( L_1 \) complement to the same member of \( L_2 \) complement.