9.2.4

Solution by: Mark

First observe that any proof that one of the languages is recursive generalizes to each L_i. So let's prove L_1 to be recursive.
By the construction in Exercise 9.2.6 (a), we know the RE languages are closed under set union. This means the union of $L_2,\ L_3, \ldots L_k$ yields an RE language. This resulting language is the complement of L_1. Theorem 9.4 states that if a language and its complement are both RE, then both languages are also recursive. Thus we conclude L_1 to be recursive.

There is an alternative solution found on the textbook's website. Take TM's $M_1,\ M_2, \ldots M_K$ for each of the languages $L_1,\ L_2, \ldots L_K$, respectively. Design a TM M with k tapes that accepts L_1 and always halts. M copies its input to all the tapes and simulates M_i on the ith tape. If M_1 accepts, then M accepts. If any of the other TM's accepts, M halts without accepting. The problem statement (parts 1 and 2) assures that every string appears in exactly one of the languages so we know exactly one of the M_i will accept. Therefore M is sure to halt and we conclude that L_1 is recursive.

9.2.5

Solution by: Brian

In this problem, we are asked to show whether L' and $\overline{L'}$ are recursive, RE, or non-RE.

$$L' = \{0w \mid w \in L\} \cup \{1w \mid w \in \overline{L}\}$$

Let us assume, for a contradiction, that L' is RE. We thus have a machine M that accepts L'.
Then, we could create a machine M' to accept $\overline{L}$, which is non-RE. To do this, we need a machine that, on input w, accepts w if and only if $w \in \overline{L}$. So, our machine M' will take an input w, add a 1 to it, making the input $1w$, then run $1w$ through our machine M. We know by the definition of L' that $1w$ will be accepted if and only if $w \in \overline{L}$. Thus, we have created a machine to accept $\overline{L}$, which is assumed to be non-RE. This is a contradiction, so therefore L' must also be non-RE.
What about $\overline{L'}$? We can see easily that

$$\overline{L'} = \{1w \mid w \in L\} \cup \{0w \mid w \in \overline{L}\}$$

Using the same argument as above (except that we pass $0w$ to our machine for $\overline{L}$), we see that $\overline{L'}$ must also be non-RE.

Comments: Most people did not discuss $\overline{L'}$, and two points were taken off for this.

9.2.6

Solution by: André

In what follows, the languages are L_1 and L_2 when there are two languages, and L_1 when there is only one. There are Turing Machines respectively called M_1 and M_2 which accepts those languages. They always halt when L_1 and L_2 are recursive.

a) Union Recursive and Recursively Enumerable languages are closes under union. Let's built a Turing Machine M which is going to simulate M_1 and M_2 on the input it gets. M will accept if either accept. So with more details, M has two tapes, receives the input on its first tape, copies it on the second tape, then simulate M_1 on its first tape, and M_2 on its second tape. We thread the execution (so we run M_1 for say 10 moves, then M_2 for 10 moves, then M_1 again, etc. As soon as one machine terminates and accept, we accept (and stop). If both don't accept, then we don't accept. If both run forever (or one does, and the other rejects), then we run forever. We need to thread the execution in case the first one runs for ever but the second one accepts. In the recursive case, we don't need to bother with threading, our M always stops. Conclusion: both R and RE are closes under union.

b) Intersection This is really the same construction as in a), you just accepts if and only if both M_1 and M_2 accepts. Note that you don't need to thread here since for M to accepts, you need to have M_1 accepts (hence stop).
Some of you tried using the De Morgan laws. Indeed $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
For recursive languages, that shows directly that intersection is closed, because union are complement are. For RE languages, well, it doesn't work. Complement of a RE languages is not necessarily RE. In fact, it is not if the languages is not recursive. But you don't really know what's going on for the union of two not RE languages. Presumably not RE. And what about the complement of a not-RE language? Could be RE.

c) Concatenation We would like to build a TM which given a string w, split it in such a way that the first part is in L_1 and the second in L_2. One way of doing it is to have the TM non deterministically split w into xy, run M_1 on x and M_2 on y. The non-determinism takes care of all you need to worry about. In the case of recursive language, it's going to terminate (the splitting, and M_1 and M_2 are always going to stop). RE: well it stops if it accepts both parts, which is want you want.
Another way of doing it is to split w in all possible ways. But be careful, a Turing Machine doesn't have an infinite amount of tapes. You can't just say, I create a new tape for every possible split. You have to build your Turing Machine before seeing w (so the number of tapes is defined before knowing how many you would like to use, and I can always give you a w for which you need more than what you had). With this method, one of the right way is to write all possible splitting on one tape (separated by a blank of whatever else); then simulate one at a time. In the case of RE languages, make sure you thread the running (you run each of the possibilities for a fixed amount of time) so that you don't get trapped by a string for which you don't have termination when they were stills splittings which could be accepted.

An horrible mistake is to assume that a Turing Machine works like a DFA, that it takes its input gradually. You cannot say that the Turing Machine is going to read the beginning of its input, and as soon as it accepts, that gives you x. No. That's plain wrong. When a Turing Machine takes a string as input, it uses the whole string to decide whether it accepts it or not. In other words, if you run the Turing Machine on a partial input, you cannot start from where it stopped and feed it the rest of the input, because that remaining part (or the absence of) will change its computation.

d) Kleene Closure For memory, $L^* = \cup_{i \in \mathbb{N}} L^i$. We are going to proceed like in part c). First we non deterministically split the input w into $w = w_1 w_2 \ldots w_k$, then we simulate M on each of the w_i. If all accepts, then we accept. The non deterministic splitting guessed the decomposition. Similarly, we can adapt the deterministic, enumerative version.

You cannot show closure using concatenation and union, because here you have an infinite number of unions, and you can't go there using induction.

e) Homomorphism Taken from http://www-db.stanford.edu/˜ullman/ialcsols/sol9.html
Consider the case where L is RE. Design a NTM M for $h(L)$, as follows. Suppose w is the input to M. On a second tape, M guesses some string x over the alphabet of L, checks that $h(x) = w$, and simulates the TM for L on x, if so. If x is accepted, then M accepts w. We conclude that the RE languages are closed under homomorphism.
However, the recursive languages are not closed under homomorphism. To see why, consider the particular language L consisting of strings of the form (M, w, c^i), where M is a coded Turing machine with binary input alphabet, w is a binary string, and c is a symbol not appearing elsewhere. The string is in L if and only if M accepts w after making at most i moves.
We have defined this particular language to break the closure of recursive language. First lest check that indeed L is recursive. Well, yes it is, because to see whether a string is in the language, we have to simulate M on w for at most i moves. So you can stop the machine after $i+1$ moves and know for sure you should reject. However, if we apply to L the homomorphism that maps the symbols other than c to themselves, and maps c to ε, we find that $h(L)$ is the universal language, which we called L_u. We know that L_u is not recursive. So in three words, L is recursive, $h(L) = L_u$ is RE but not recursive. So recursive languages are not closes under homomorphism.

Another version to show that recursive languages are not closed under homomorphism. Consider a Turing Machine M, whose language L is RE. Now consider the language L' defined as the sequence of computation the TM does when it accepts a string. Actually, put hats on the first ID. So you have $l = q_0\hat{x_1}\hat{x_2}\dots\hat{x_n} \ \vdash \ x_1'a_1x_2\dots x_n \ \vdash \ \dots$ where $w = x_1x_2\dots x_n$ is the input to M. The language is clearly recursive. Just check whether the transitions respect the transitions of M. Consider the homomorphism h that maps hatted symbols to their version without hats, and everything else, including $\vdash$, to ε. Clearly, $h(l) = w$. So $h(L') = L$. In other words, you have a homomorphism between a recursive language and a RE language.

f) Inverse Homomorphism Both recursive and RE languages are closed. Consider the following Turing Machine: it takes its input w. It should tell whether $w \in h^{-1}(L)$. That's easy: compute $h(w)$. We for sure have $w \in h^{-1}(L) \Rightarrow h(w) \in L$ because that's the definition of h^{-1}. Now test if $h(w) \in L$. If so, then $w \in h^{-1}(L)$; if not, it isn't.
You cannot say you just apply part e) here. First because I don't know what that means, two because $h(h^{-1}(L) = L$ is true, but what you are using is $h^{-1}(h(L))$, and that's usually not the same as L. (Note the difference in the oder between h and h^{-1}. Consider L to consist of only one string, or in fact whatever you want, finite, recursive, RE, etc. h to map everything to 0. $h(L) = \{0\}$ but $h^{-1}(h(L)) = \Sigma^*$.