Definition 1. We will use the following definitions,

- $\mathcal{RE}$: the recursively enumerable languages over an implicitly given alphabet
- $\mathcal{TM}$: the set of Turing Machines
- $\overline{S}$: the complement of a set $S$ over an implicitly given universal set
- $\mathcal{P}(S)$: the power set, the set of all subsets, of $S$.

1. Problem (10.5)

Theorem 1. Suppose that $L_1, L_2, \ldots, L_k$ form a partition of $\Sigma^*$. If each $L_i$ is recursively enumerable, then each $L_i$ is recursive.

Proof. This theorem can be proven directly by constructing the appropriate $\mathcal{TM}$. But given that it is a generalization of Theorem 10.4, we prove it by appealing to Theorem 10.4.

Consider $L_i$, where $i = 1, 2, \ldots, k$. Repeated application of Theorem 10.2 tells us that,

$$L = \bigcup_{j=1}^{k} L_j$$

is recursively enumerable. Since $L_1, L_2, \ldots, L_k$ are a partition of $\Sigma^*$ we have that $\overline{L_i} = L$. Thus $L_i$ and $\overline{L_i}$ are recursively enumerable. That $L_i$ is recursive follows from Theorem 10.4. \qed

2. Problem 2 (10.23)

Theorem 2. The set $S$ of languages $L$ over $\{0,1\}$, so neither $L$ nor $\overline{L}$ are recursively enumerable, is uncountable.

Proof. Suppose, to get a contradiction, that the above claim is false, i.e that $S$ is countable. Note that the complement of $S$, $\overline{S} = \{L \in \mathcal{P}(\Sigma^*) \mid L \in \mathcal{RE} \text{ or } \overline{L} \in \mathcal{RE}\}$ can be written as the union of two sets $\overline{S} = S_1 \cup S_2$ where

- $S_1 = \{L \in \mathcal{P}(\Sigma^*) \mid L \in \mathcal{RE}\}$
- $S_2 = \{L \in \mathcal{P}(\Sigma^*) \mid \overline{L} \in \mathcal{RE}\}$.

$S_1 = \mathcal{RE}$ is countable by Example 10.3. We can then show that $S_2$ is countable by observing that we have a bijection $f : S_1 = \mathcal{RE} \rightarrow S_2$ defined by $L \mapsto \overline{L}$. Thus our two sets have the same cardinality and are therefore countable. By Theorem 10.7, it follows that $\overline{S} = S_1 \cup S_2$ is countable. By using the same theorem again we can
show that \( S \cup \overline{S} \) is countable since both \( S \) and \( \overline{S} \) are countable sets. But we know that \( S \cup \overline{S} = \mathcal{P}(\Sigma^*) \) which has been shown to be uncountable in Theorem 10.8. Thus, by assuming that \( S \) was countable, we have reached a contradiction, showing \( S \) must be uncountable.

We now construct such a language. Let \( \{L_0, L_1, \ldots\} = \mathcal{RE} \) and \( \{x_0, x_1, \ldots\} = \Sigma^* \). We construct \( S \) as follows,

\[
\begin{align*}
\text{if } x_{2n} \in L_n, & \text{ then } x_{2n} \notin S \\
\text{if } x_{2n} \notin L_n, & \text{ then } x_{2n} \in S \\
\text{if } x_{2n+1} \in \overline{L_n}, & \text{ then } x_{2n+1} \notin S \\
\text{if } x_{2n+1} \notin \overline{L_n}, & \text{ then } x_{2n+1} \in S
\end{align*}
\]

To show that \( S \notin \mathcal{RE} \), suppose that it was and let \( S = L_k \) for some \( k \in \mathbb{N} \). Consider if \( x_{2k} \in S \). If \( x_{2k} \in L_k \), then, by construction, \( x_{2k} \notin S \), thus \( S \neq L_k \). If \( x_{2k} \notin L_k \), then \( x_{2k+1} \in S \), thus \( S \neq L_k \). Therefore \( S \notin \mathcal{RE} \).

To show that \( \overline{S} \notin \mathcal{RE} \), suppose that it was and let \( \overline{S} = L_k \) for some \( k \in \mathbb{N} \). This can be restated as \( S = \overline{L_k} \). Consider if \( x_{2k+1} \in S \). If \( x_{2k+1} \in \overline{L_k} \), then, by construction, \( x_{2k+1} \notin S \), thus \( S \neq \overline{L_k} \). If \( x_{2k+1} \notin \overline{L_k} \), then \( x_{2k+1} \in S \), thus \( S \neq \overline{L_k} \). Therefore \( S \notin \mathcal{RE} \).

3. Problem 3 (10.31)

**Theorem 3.** Every infinite recursively enumerable set \( L \) has an infinite recursive subset.

**Proof.** Since \( L \) is recursively enumerable, by Theorem 10.5 there is some \( T \in \mathcal{T} \mathcal{M} \) by which it can be enumerated. This enumeration lists the elements in \( L \) one at a time, and \( T \) continues to execute forever since \( L \) is infinite. Since \( L \in \mathcal{RE} \), the strings are not generally listed in canonical order. However, we can enumerate a subset \( L' \) of \( L \) in canonical order as follows. We construct \( T' \in \mathcal{T} \mathcal{M} \) that uses \( T \) to enumerates \( L \) on a second tape. The first string that \( T' \) enumerates is the first string that \( T \) enumerates. We proceed to enumerate \( L \) and every time \( T \) lists another string of \( L \), we test this string against the last string generated by \( T' \) to determine if the new element listed is canonically after the previous. If it is, it is copied to the output tape and \( T' \) continues enumerating \( L \). Thus from the set \( L \), a subset \( L' \) can be listed by \( T' \), such that all elements are listed in strictly increasing canonical order. Therefore \( L' \) is recursive by Theorem 10.6. Since \( L \) is infinite, no matter how large an element is listed by \( T' \), it is guaranteed that a canonically larger element will eventually be enumerated by \( T' \). Thus \( L' \) must also be infinite.

All that remains is to argue that the test for the canonical ordering of two strings is computable. We can do this by enumerating \( \Sigma^* \) in canonical order and checking which of the two strings is listed first. The first listed is canonically earlier that the second. Since both strings are of finite length, they will eventually be enumerated, thus this test will always finish. ■
4. Problem 4 (11.1)

The language produced by

\[
S \rightarrow LaR \\
L \rightarrow LD \mid LT \mid \Lambda \\
Da \rightarrow aaD \\
Ta \rightarrow aaaT \\
DR \rightarrow R \\
TR \rightarrow R \\
R \rightarrow \Lambda
\]

is \( \{a^n \mid n = 2^p3^q, p \in \mathbb{N}, q \in \mathbb{N} \} \). The first grammar derivation consist of \( S \rightarrow LaR \), which produces a left marker \( L \) and a right marker \( R \) (both of which eventually derive \( \Lambda \)) and a single \( a \) terminal. The left marker can derive \( LD \) and \( LT \). The \( D \) variable doubles all \( a \)'s in the string as it sweeps from left to right across the string and the \( T \) variable triples all \( a \)'s in the string. The \( D \) and \( T \) non-terminals are not allowed to cross, thus the doubling and tripling actions are not allowed to intermix. When the \( L \) and \( T \) variables reach the right marker \( R \), they disappear. We can derive strings \( a, aa, aaa, aaaa \), and \( aaaaaa \), etc. but \( aaaaa \) cannot be derived since no combinations of doubling or tripling our first single \( a \) can produce \( a^5 \) (5 has no factors of 2 or 3).

Consider the language generated by

\[
S \rightarrow ABC S \mid ABC \\
AB \rightarrow BA \\
BA \rightarrow AB \\
AC \rightarrow CA \\
CA \rightarrow AC \\
BC \rightarrow CB \\
CB \rightarrow BC \\
A \rightarrow a \\
B \rightarrow b \\
C \rightarrow c
\]

The language produced by this grammar is \( \{x \in \{a, b, c\}^+ \mid \text{num}_a(x) = \text{num}_b(x) = \text{num}_c(x)\} \), where \( \text{num}_l(x) \) is the number of occurrences of the letter \( l \) in the string \( x \). The first grammar rule, \( S \rightarrow ABC S \mid ABC \), ensures we have at least one set of \( ABC \) and we have an equal number of \( A \)'s, \( B \)'s, and \( C \)'s. The following rules allow the variables to be swapped into any order without changing the numbers of them. Finally, the variables are frozen into their respective terminal counterparts at any random point in the swapping. Thus, we can not know anything about the order we reached before freezing to terminals, but the relative numbers of \( a \)'s, \( b \)'s, and \( c \)'s must be equal. We never changed the number of each variable after the first rule, and each variable derives exactly one terminal.