Problem 2: (a) Suppose that \( K \) were regular. Then the language \( \overline{K} \) would be regular, and

\[ \overline{K \cup a^*b^*} \text{ would be regular.} \]

What is \( \overline{K \cup a^*b^*} \)?

\[ \overline{K \cup a^*b^*} = \{ z \in \Sigma^* \mid z \not\in K \cup a^*b^* \text{ and } z \text{ is a permutation of } y \} \]

\[ = \{ a^n b^n \mid n \geq 3 \} \]

Applying the Pumping Lemma for regular languages to \( \overline{K \cup a^*b^*} \),

will show the non-regularity of \( \overline{K \cup a^*b^*} \).

(b) Notice that for all \( x, y \in \{a, b\}^* \),

\( x \) is a permutation of \( y \) if and only if \( \#_a(x) = \#_a(y) \)

and \( \#_b(x) = \#_b(y) \).

Thus, \( x \not\in K \) if and only if \( \#_a(x) \neq \#_a(y) \) or \( \#_b(x) \neq \#_b(y) \).

Now we have \( K = \{ x \in \Sigma^* \mid x \not\in \{a, b\}^* \text{ and } \#_a(x) \neq \#_a(y) \} \)

\[ \cup \{ x \in \Sigma^* \mid x \not\in \{a, b\}^* \text{ and } \#_b(x) \neq \#_b(y) \} \]

Each disjunct is generated by a context-free grammar by considering \( \neq \) as \( < \text{ or } > \).
We construct a PDA $P$ recognizing $D$. This PDA guesses corresponding places on which $x$ and $y$ differ. Checking that the places correspond is tricky. Doing so relies on the observation that the two corresponding places are $n/2$ symbols apart, where $n$ is the length of the entire input. Hence, by ensuring that the number of symbols between the guessed places is equal to the number other symbols, the PDA can check that the guessed places do indeed correspond. Here we give a more detailed description of the PDA algorithm. If something goes wrong, for example, popping when the stack is empty, or getting to the end of the input prematurely, $P$ rejects on that branch of the computation.

1. Read next input symbol and push 1 onto the stack.
2. Nondeterministically jump to either 1 or 3.
3. Record the current input symbol $a$ in the finite control.
4. Read next input symbol and pop the stack. Repeat until stack is empty.
5. Read next input symbol and push 1 onto the stack.
6. Nondeterministically jump to either 5 or 7.
7. Reject if current input symbol differs from $a$.
8. Read next input symbol and pop the stack. Repeat until stack is empty.
9. Accept if input is empty.

Alternatively we can give a CFG for this language as follows.

$$S \rightarrow AB \mid BA$$
$$A \rightarrow XAX \mid 0$$
$$B \rightarrow XBX \mid 1$$
$$X \rightarrow 0 \mid 1$$

2.27

a. To see that $G$ is ambiguous, note that the string

if condition then if condition then a:=1 else a:=1

has two different leftmost derivations (and hence parse trees):

1. $(\text{STMT})$
   $\rightarrow (\text{IF-THEN})$
   $\rightarrow$ if condition then $(\text{STMT})$
   $\rightarrow$ if condition then $(\text{IF-THEN-ELSE})$
   $\rightarrow$ if condition then if condition then $(\text{STMT})$ else $(\text{STMT})$
   $\rightarrow$ if condition then if condition then a:=1 else $(\text{STMT})$
   $\rightarrow$ if condition then if condition then a:=1 else a:=1

2. $(\text{STMT})$
   $\rightarrow (\text{IF-THEN-ELSE})$
   $\rightarrow$ if condition then $(\text{STMT})$ else $(\text{STMT})$
   $\rightarrow$ if condition then $(\text{IF-THEN})$ else $(\text{STMT})$
   $\rightarrow$ if condition then if condition then $(\text{STMT})$ else $(\text{STMT})$
   $\rightarrow$ if condition then if condition then a:=1 else $(\text{STMT})$
   $\rightarrow$ if condition then if condition then a:=1 else a:=1

b. The ambiguity in part a) arises because the grammar allows matching an else both to the nearest and to the farthest then. To avoid this ambiguity we construct a new grammar that only permits the nearest match, by disabling derivations which introduce an $(\text{IF-THEN})$ before an else. This grammar has two new variables: $(\text{E-STMT})$ and $(\text{E-IF-THEN-ELSE})$, which work just like their non-$(\text{E})$ counterparts except that they cannot generate the dangling $(\text{IF-THEN})$. The rules of the new grammar are the same as for the old grammar except we remove the $(\text{IF-THEN-ELSE})$ rules and add the following new rules:

$$(\text{E-STMT}) \rightarrow (\text{E-IF-THEN-ELSE})$$
$$(\text{E-IF-THEN-ELSE}) \rightarrow \text{if condition then } (\text{E-STMT}) \text{ else } (\text{E-STMT})$$
$$(\text{IF-THEN-ELSE}) \rightarrow \text{if condition then } (\text{E-STMT}) \text{ else } (\text{STMT})$$
Problem 5.

(a) First, since $B$ is finite, $B$ is regular, and $\overline{B}$ is also regular.

Then,

$$A - B = A \cap \overline{B} \text{ is context-free}$$

(Intersection with regular languages preserves context-freeness)

Next, the language $B - A = B \cap \overline{A} \subseteq B$. Since $B$ is finite, $B - A$ is also finite, and $B - A$ is context-free (a finite language is regular).

Therefore, $A \triangleleft B = (A - B) \cup (B - A)$ is context-free.

(b) If $B$ is regular but not necessarily finite, then $A \triangleleft B$ is not necessarily context-free.

Consider the example: $B = \Sigma^*$ and a context-free language $A$ such that $A$ is not context-free.

For example, $\Sigma = \{a, b, c\}$, $B = \Sigma^*$, and
\[ A = \{ a^nb^nc^n \mid n \geq 0 \} \]

\[ \text{complementation:} \]

Note that:

1. The language \( \{ a^nb^nc^n \mid n \geq 0 \} \) is not context-free — stated in lecture.

2. The language \( A = \{ a^nb^nc^n \mid n \geq 0 \} \) is context-free — Why?

Now, \( A \triangle B = (A - B) \cup (B - A) \)
\[ = \emptyset \cup (\varepsilon^* - A) = \overline{A} \]
\[ = \{ a^nb^nc^n \mid n \geq 0 \} \]
which is not context-free.
Idea: We construct a PDA $M$ accepting $B$ (with final state) as follows:

1. On input of the form $x^R y$, $M$ remembers the first half (prefix $x$) in its stack, by nondeterministically guessing the "midpoint" of the input.

$$x_1 x_2 \cdots x_n y_1 y_2 \cdots y_n$$

Then, for the second half (suffix $y^R$), $M$ computes the bit-wise exclusive-or of bit-pairs in succession and

$$x_n \oplus y_1 \quad (n-th \ bit \ of \ x \oplus y)$$

verifies the condition

$$x_i \oplus y_2 \quad ((i-1)st \ bit \ of \ x \oplus y) \quad \forall x \oplus y \in A.$$ 

$$x_1 \oplus y_n \quad (1st \ bit \ of \ x \oplus y)$$

2. Notice that the successive computations are performed on $x^R \oplus y^R$, hence the verification of $x \oplus y \in A$ is equivalent to that of $x^R \oplus y^R \in A^R$.

Since $A$ is regular, $A^R$ is also regular.

Let $A^R = L(M_A)$ for some deterministic finite automaton

$$M_A = (Q_A, \Sigma, \delta_A, q_0, F_A).$$
The ideas above suggest that:

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \] where:

\[ Q = \{ q_0, q_{\text{accept}} \} \quad \text{and} \quad q_0 \text{ and } q_{\text{accept}} \text{ are two (new)} \]

\[ F = \{ q_{\text{accept}} \} \]

\[ \Sigma = \{ 0, 1 \} \]

\[ \Gamma = \{ 0, 1, Z_0 \} \]

\[ \forall a \in \Sigma \forall x \in \Gamma \quad \delta(q_0, a, x) \text{ includes } (q_0, a \chi) \quad \text{- remember the } \text{free} \text{ input in \text{the steel}} \]

\[ \forall x \in \Gamma \quad \delta(q_0, \varepsilon, x) \text{ includes } (q_{\text{accept}}, x) \quad \text{- non-deterministically guess the “midpoint” of the input and ready to verify the condition } \varepsilon \text{ (free input) } \]

\[ \forall q \in Q \quad \forall a \in \Sigma \forall x \in \Gamma \quad \delta(q, a, x) \text{ includes } (q_{\text{accept}}, a \chi) \quad \text{- simulate } M \text{ in state } q \text{ with bit-pair } a \text{ and } x \text{ and } \delta \text{ in state } q_{\text{accept}} \]

\[ \forall q \in F \quad \delta(q, \varepsilon, Z_0) \text{ includes } (q_{\text{accept}}, Z_0) \quad \text{- } M \text{ is ready to accept the input when } M \text{ has verified } \delta(q_{\text{accept}}, a \chi, y) \text{ for all } a \text{ and } y \text{ in } \Sigma \]

For all other combinations, \delta-value is \( \bot \).
Problem 7
(a) Note that
\[ f(a) = f(j) \]

\[ 2 \cdot \left( \text{number of 0s over index range } [1 \ldots i] \right) - \left( \text{number of 1s over index range } [1 \ldots j] \right) = 2 \cdot \left( \text{number of 0s over index range } [i+1 \ldots j] \right) - \left( \text{number of 1s over index range } [i+1 \ldots j] \right) \]

\[ 2 \cdot \left( \text{number of 0s over index range } [i+1 \ldots j] \right) = \text{number of 1s over index range } [i+1 \ldots j] \]

\[ x_{i+1}, x_{i+2}, \ldots, j \in L \]

(b) Consider a string \( x \) of length \( 3n \), and consider the function \( f \) for this given string \( x \).

We have \( f(1) = 2 \) or \( f(3n) = 0 \).

Note that, the function \( f \) can only decrease its value by 1 at each index position.

So, we must have \( f(i) = 1 \) or \( f(j) = 0 \) at some indices \( i \) and \( j \) with \( i < j \).

Let \( i \) and \( j \) be the first such indices, that we see a "1" and "0", respectively.

Observe that these two indices can not come from increases caused by seeing a "0", so they must be "1"s.

We also have \( f(i+1) - f(i) = 0 \), which means that the segment with index range \([2 \ldots i-1]\) is also in \( L \). Similarly, the segments with index range \([i+1 \ldots j-1]\) or index range \([j+1 \ldots 3n]\) are also in \( L \).
(c) We use one variable, no start variable $S$,

\[
S \to \varepsilon \\
S \to 0S1S1S | 1S0S1S | 1S1S0S
\]

3 possible permutations: 0, 1, 1