2. (a) For \( i \in \{ -1, 0, 1 \} \), \( q_i \): \( \#_i(x) - \#_0(x) = i \) for consumed input \( x \). 

- \( q_{\text{bad}} \): existence of a prefix \( y \) in consumed input with \( |\#_i(y) - \#_0(y)| > 1 \) (violates the desired condition).

(b) \( q_1 \) and \( q_2 \): for the machine to nondeterministically enter the

"having encountered first 0" and "having encountered second 0" states, respectively — with a separation of length \( s \#_0 x \) for some \( k \geq 0 \).

Ref: ... \( q_i \) account for the length factor of 5.
(c) Consider no follow-up stepwise approach to the solution.

1. We show that finite automata can be used to perform "left-to-right" computation.

For example, we can construct an FA accepting the language

\[ \{ x \in \{ a, b, c \}^* \mid x = b \} \]

left-to-right multiplication

yielding \( b \)

We can use the states to remember the left-to-right multiplication yield on the consumed input so far:

\[ \begin{aligned}
\text{start} & \rightarrow \text{start} \\
a & \rightarrow a \\
b & \rightarrow b \\
c & \rightarrow c \\
\text{state (b)} & \text{is an accepting state} \\
\text{Since we desire } & \overline{a} = b.
\end{aligned} \]

\[ H \uparrow \downarrow b \quad H \in \{ a, b, c \}^* \quad \delta(q, a) = q \cdot b \]

2. Now we show that finite automata can be used to perform "right-to-left" computation.

For an example similar to 1:

\[ \{ x \in \{ a, b, c \}^* \mid x = b \} \]

right-to-left multiplication

yielding \( b \)

We "reverse" our computation, instead of following the multiplication table (for \( \circ \)) "deterministically," we start with our initial guess for the overall right-to-left multiplication on the entire input, and for each symbol scanned, we refine/guess our guess for the right-to-left multiplication on the unconsumed input.
\textbf{Guess} \( b \) is the start state.

Since we desire \( x = b \), we have

From the state guess \( b \):

Transition(s) on \( a \):

Solving \( b = a \circ \theta \) for \( \theta \)

(\text{no solution})

Transition(s) on \( b \):

Solving \( b = b \circ \theta \) for \( \theta \)

(\text{one solution for } \theta = a)

- adjust the guess to \( c \)

Transition(s) on \( c \):

Solving \( b = c \circ \theta \) for \( \theta \)

(\text{one solution for } \theta = c)

Similarly, we can determine transitions from other guess states.

But that is not all. The transitions above are for normal processing for right-to-left multiplications. How should we halt when there is no more multiplication (that is, exactly one symbol left)?

\textbf{Start} \rightarrow \text{guess} \ b \rightarrow \text{accept}

\textbf{Guess} \ a \rightarrow \text{accept}

\textbf{Guess} \ c \rightarrow \text{accept}

To make sure "exactly one symbol left" before entering \text{accept}, there should be no transition out of \text{accept}.
3. Finally, to construct an FA to accept the given language, we superimpose the ideas of 1 and 2.

Not attempting to optimize the state structure, we can:

\[ \text{guess} = a \]
\[ \text{guess} = b \]
\[ \text{guess} = c \]

Start

Structure (1) - Using 1. and 2. to compute the current left-to-right multiplication yield and update the current guess on the right-to-left multiplication yield.

Structures (2) and (3) are similar.
Follow the lecture notes: Proof of the equivalence between NFA-$E$ and NFA's:

1. $M \rightarrow M'$:
   - Compute $E$-closures for states $s$ on $M$.
   - Compute $E$-closures for subsets of states $S$ on $M$.
   - Relate the transition function $S_1$ of $M_1$ to the multi-step transition function $S_2$ of $M$.

2. $M_1 \rightarrow M_2$: Using Subset Construction:
   - Relate the transition function $S_2$ of $M_2$ to the transition function $S_1$ of $M$.

[Sip06/ESip13] Problem 1.33

Idea: 1. For two equal-length bit-sequences $S_1$ and $S_2$, $S_1 + S_2$ generates at most one carry-bit (the highest-order bit in the summation result).

   2. Hence, for a bit-sequence $S_1 \in \{0, 1\}^n$, the bit-sequences corresponding to the (decimal) multiplication:

      \[
      2 \times S_1 \quad \text{is in} \quad \{0, 1\}^{n+1} \\
      2^2 \times S_1 \quad \text{is in} \quad \{0, 1\}^{n+2} 
      \]

      So, the bit-sequence corresponding to the (decimal) multiplication $3 \times S_1$ is in $\{0, 1\}^{n+2}$.

As suggested in the problem statement (assuming the result in Problem 1.31), we may assume that the finite automaton to be constructed reads the input from right-to-left.
Thus:

\[
\begin{bmatrix}
a
\end{bmatrix}
\begin{bmatrix}
a_n & \cdots & a_1 & a_0
b
\end{bmatrix}
\begin{bmatrix}
b_n & \cdots & b_1 & b_0
\end{bmatrix}
\]

carry-bits

we use two states to remember the two possible carry-bits for the successful verification of the relationship.

The two bit-sequences \( b_n b_{n-1} \cdots b_1 b_0 \) = \( 3 \times a_n a_{n-1} \cdots a_1 a_0 \)

- The consumed input so far (decimal)

The possible transitions out of a state or (additional symbol) \( [a] \) (where \( a \in \{0, 1\} \)):

- The unsuccessful verification of the relationship

in the augmented consumed input (using the carry-bits remembered in the current state and the information in \( a \) and \( b \)) transits the finite automaton to a "rejection" state

- The successful verification of the relationship

in the augmented consumed input transits the finite automaton to a state with updated carry-bits.
1.41 We construct a DFA which alternately simulates the DFAs for A and B, one step at a time. The new DFA keeps track of which DFA is being simulated. Let \( M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1) \) and \( M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2) \) be DFAs for A and B. We construct the following DFA \( M = (Q, \Sigma, \delta, s_0, F) \) for the perfect shuffle of A and B.

i) \( Q = Q_1 \times Q_2 \times \{1, 2\} \).

ii) For \( q_1 \in Q_1, q_2 \in Q_2, b \in \{1, 2\}, \) and \( a \in \Sigma: \)

\[
\delta'((q_1, q_2, b), a) = \begin{cases} 
(\delta_1(q_1, a), q_2, 2) & b = 1 \\
(\delta_1(q_1, a), 1) & b = 2.
\end{cases}
\]

iii) \( s_0 = (s_1, s_2, 1) \).

iv) \( F = \{(q_1, q_2, 1) \mid q_1 \in F_1 \text{ and } q_2 \in F_2\} \).

1.40 b. Let \( M = (Q, \Sigma, \delta, q_0, F) \) be an NFA recognizing A, where A is some regular language. We construct \( M' = (Q', \Sigma, \delta', q_0', F') \) recognizing \( \text{NOEXTEND}(A) \) as follows:

i) \( Q' = Q \)

ii) \( \delta' = \delta \)

iii) \( q_0' = q_0 \)

iv) \( F' = \{q \mid q \in F \text{ and there is no path of length } \geq 1 \text{ from } q \text{ to an accept state}\} \).

1.59 If \( M \) has a synchronizing sequence, then for any pair of states \((p, q)\) there is a string \( w_{p,q} \) such that \( \delta(p, w_{p,q}) = \delta(q, w_{p,q}) = h \), where \( h \) is the home state. Let us run two copies of \( M \) starting at states \( p \) and \( q \) respectively. Consider the sequence of pairs of states \((u, v)\) that the two copies run through before reaching home state \( h \). If some pair appears in the sequence twice, we can delete the substring of \( w_{p,q} \) that takes the copies of \( M \) from one occurrence of the pair to the other, and thus obtain a new \( w_{p,q}' \). We repeat the process until all pairs of states in the sequence are distinct. The number of distinct state pairs is \( k^2 \), so \( |w_{p,q}| \leq k^2 \).

Suppose we are running \( k \) copies of \( M \) and feeding in the same input string \( s \). Each copy starts at a different state. If two copies end up at the same state after some step, they will do exactly the same thing for the rest of input, so we can get rid of one of them. If \( s \) is a synchronizing sequence, we will end up with one copy of \( M \) after feeding in \( s \). Now we will show how to construct a synchronizing sequence \( s \) of length at most \( k^3 \).

i) Start with \( s = \epsilon \). Start \( k \) copies of \( M \), one at each of its states. Repeat the following two steps until we are left with only a single copy of \( M \).

ii) Pick two of \( M \)'s remaining copies (\( M_p \) and \( M_q \)) that are now in states \( p \) and \( q \) after reading \( s \).

iii) Redefine \( s = sw_{p,q} \). After reading this new \( s \), \( M_p \) and \( M_q \) will be in the same state, so we eliminate one of these copies.

At the end of the above procedure, \( s \) brings all states of \( M \) to a single state. Call that state \( h \). Stages 2 and 3 are repeated \( k - 1 \) times, because after each repetition we eliminate one copy of \( M \). Therefore \( |s| \leq (k - 1)k^2 < k^3 \).

1.61 Let \( M \) be a DFA. Say that \( w \) leads to state \( q \) if \( M \) is in \( q \) after reading \( w \). Notice that if \( w_1 \) and \( w_2 \) lead to the same state, then \( w_1p \) and \( w_2p \) also lead to the same state, for all strings \( p \).

Assume that \( M \) recognizes \( C_k \) with fewer than \( 2^k \) states, and derive a contradiction. There are \( 2^k \) different strings of length \( k \). By the pigeonhole principle, two of these strings \( w_1 \) and \( w_2 \) lead to the same state of \( M \).

Let \( i \) be the index of the first bit on which \( w_1 \) and \( w_2 \) differ. Since \( w_1 \) and \( w_2 \) lead \( M \) to the same state, \( w_1b^{i-1} \) and \( w_2b^{i-1} \) lead \( M \) to the same state. This cannot be the case, however, since one of the strings should be rejected and the other accepted. Therefore, any two distinct \( k \) bit strings lead to different states of \( M \). Hence \( M \) has at least \( 2^k \) states.
6. (a) Similar to an example in lecture:

\[ x \in (a,b)^* \mid |x| \leq 4 \] is denoted by \((a+b+e)^4\).

Here we have:

\[ (3+0+1^*)^3 \]

i.e., \((3+0+1^*) (3+0+1^*) (3+0+1^*)\)

(b) View a typical string in the language as:

\[
\begin{align*}
&1^*01^*01^*01^* & \quad 1^*01^*01^*01^* \\
\hline
&\text{segment containing} & \text{exactly }3\text{ }0s
\end{align*}
\]

A candidate regular expression is:

\[ (1^*01^*01^*01^*)^* \]
Let \( r = \lambda (s + \lambda) \) and \( r = \lambda (s + \lambda) \). Then, we have the desired equality.

Here we elect to follow the approach of showing
\[
(\lambda(s)L(r)UL(r))^* \leq \lambda(s)L(r)UL(r)^*
\]
and "vice versa."

Let \( x \in (\lambda(s)L(r)UL(r))^* \) be arbitrary, that is, \( x = yz \) where \( y \in (\lambda(s)L(r)UL(r))^* \) and \( z \in \lambda(r) \).

Case when \( y = \lambda \): Then \( x = \lambda z \in \lambda(r) \leq \lambda(s)L(r)UL(r)^* \).

Case when \( y \neq \lambda \): Then \( y = y_1y_2\cdots y_n \) for some \( n \geq 1 \) with \( y_i \in \lambda(s)L(r)UL(r) \) for \( i = 1, 2, \ldots, n \).

So, for each \( i = 1, 2, \ldots, n \)
\[
y_i = u(v) \text{ where } u \in \lambda(r) \text{ and } v \in \lambda(s)
\]
or
\[
y_i = u \text{ where } u \in \lambda(r).
\]
This says that
\[ x = y_1 y_2 \cdots y_n z \in L(r)(L(s)L(r) U L(r))^n. \]

Therefore \( x \in L(r)(L(s)L(r) U L(r))^* \).

Thus, \( (L(r) L(s) U L(r))^* L(r) \subseteq L(r)(L(s)L(r) U L(r))^* \).

Similar argument can show the reverse subset containment.

Therefore, \( (r^*s + r)^* r = r(s^*r + r^*) \).

\[ (b) \quad (r+s)^* \neq r^* + s^* \text{ in general.} \]

Counter-example: \( r = a \text{ and } s = b \quad (\Sigma = \{a, b\}) \)

\( (r+s)^* \) denotes the language \( \{a, b\}^* \),
and \( r^* + s^* \) denotes the language \( \{a\}^* U \{b\}^* \).

\[ (c) \quad (r^*s^*)^* = (r+s)^* \quad \text{True; prove it.} \]
10. An arbitrary language $L$ on an alphabet $\Sigma$ satisfies that:
$$L \neq \emptyset, \ L \supseteq \Sigma^*, \ \text{and} \ L^2 = L.$$ 

1. Claim that $x \in L$.

Question: What is a string $x \in L$ with minimum length? Let $x$ be a shortest string in $L$ (and we show that $x = \epsilon$).
Now $xx \in L^2$ (why?)

In fact $xx$ is a shortest string in $L^2$ (why?)

Notice that $L^2 = L$ (by assumption),

$xx$ is a shortest string in $L$.

Hence $|xx| = 2|a|$ (both are shortest of $L$).
So we must have $6| \Rightarrow x = \epsilon$.

2. Claim that $L$ is not finite.

Notice that $L$ is not finite if

$$\exists n \forall x \in L (|x| \geq n).$$

There exists string of $L$
sufficiently long.

Question: Suppose the contrary that $L$ were finite.
Is it possible to have a string of $L$ with maximum length?

Similarly done.