Instructions

Upload a single file to Gradescope for each group. All group members' names and PID.s should be on each page of the submission. Your assignments in this class will be evaluated not only on the correctness of your answers, but on your ability to present your ideas clearly and logically. You should always explain how you arrived at your conclusions, using mathematically sound reasoning. Whether you use formal proof techniques or write a more informal argument for why something is true, your answers should always be well-supported. Your goal should be to convince the reader that your results and methods are sound.

For questions that only ask for diagrams, justifications are not required but highly recommended. It helps to show your logic in achieving the answers and partial credit can be given if there are minor mistakes in the diagrams.

Reading Sipser Sections 1.3 and 1.4

Key Concepts DFA, NFA, equivalence of DFA and NFA, regular expressions, equivalence of DFA and regular expressions, regular languages, closure of the class of regular languages under certain operations, the Pumping Lemma, pumping length, proofs of nonregularity
Problem 1 (10 points)

For each of the regular expressions below, give two examples of strings in the corresponding language and give two examples of strings not in the corresponding language.

a. \((000 \cup 1)^* (0 \cup 111)^*\)

Examples of strings in the language: 000, 1, 0, 111, 11, 00, 10, 00111, empty string
Examples of strings not in the language: 01, 001, 011

b. \((1 \cup 01 \cup 10)^*\)

Examples of strings in the language: 1, 01, 10, 11, 101, 110, empty string
Examples of strings not in the language: 0, 00, 000, 001, 010

c. \(\varepsilon^* \cup (0^* \emptyset) \cup 1\)

Examples of strings in the language: empty string and 1 are the only examples
Examples of strings not in the language: 0, 00, 11, and any string of length 2 or longer

Common Mistakes
- Many people tried to say that strings including 0 were in the language defined by part c. This is incorrect because \((0^* \emptyset) = \emptyset\).
- Many people said that \(\emptyset\) was a string in the language defined by part c. \(\emptyset\) is the empty set, i.e. the language that recognizes no strings. It is a set, not a string.

Problem 2 (10 points)

In this problem, you will convert the following DFA into a regular expression, using the GNFA construction from Lemma 1.60 in the textbook (Sipser section 1.3, page 69). Note that this DFA recognizes the language of strings that contain an odd number of ones.
Create a 4-state GNFA that is equivalent to the above DFA, using the structure suggested by the proof of Lemma 1.60. To help you get started, we have provided the structure of the GNFA, so you only need to fill in the table below.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Regular expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(st \rightarrow q_0)</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>(st \rightarrow q_1)</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>(st \rightarrow \text{acc})</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>(q_0 \rightarrow q_0)</td>
<td>0</td>
</tr>
<tr>
<td>(q_0 \rightarrow q_1)</td>
<td>1</td>
</tr>
<tr>
<td>(q_0 \rightarrow \text{acc})</td>
<td>(\varepsilon)</td>
</tr>
<tr>
<td>(q_1 \rightarrow q_0)</td>
<td>1</td>
</tr>
<tr>
<td>(q_1 \rightarrow q_1)</td>
<td>0</td>
</tr>
<tr>
<td>(q_1 \rightarrow \text{acc})</td>
<td>(\varepsilon)</td>
</tr>
</tbody>
</table>
Note that in the GNFA above, only the transition from the new start state \( st \) to the DFA start state \( q_0 \) is labeled with \( \varepsilon \). The other transitions from \( st \) are labeled with the \( \varnothing \). The transition from \( q_1 \) to the new accept state \( acc \) is labeled \( \varepsilon \) but the transitions from \( st \) and \( q_0 \) to \( acc \) are labeled with \( \varnothing \) because those were not accept states in the DFA.

(b) Create a 3-state GNFA equivalent to the GNFA in part (a), after removing the state \( q_1 \). Again, we have provided the structure of this GNFA, so you only need to fill in the table below. Try to simplify each edge’s regular expression as much as possible (for example, using the results of problem 3a).

![Diagram of GNFA](image)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( st \to q_0 )</td>
<td>( \varepsilon )</td>
</tr>
<tr>
<td>( st \to acc )</td>
<td>( \varnothing )</td>
</tr>
<tr>
<td>( q_0 \to q_0 )</td>
<td>( 0 \cup 1^*1 )</td>
</tr>
<tr>
<td>( q_0 \to acc )</td>
<td>( 10^* )</td>
</tr>
</tbody>
</table>

When we eliminate state \( q_1 \), the transition from \( st \) to \( acc \) remains \( \varnothing \) because the transition from \( st \) to \( q_1 \) was labeled with \( \varnothing \). The transition from \( q_0 \) to itself becomes the union of its previous value, 0, and the regular expression that describes a computation going from \( q_0 \) to \( q_1 \) and back to \( q_0 \), which is 1 concatenated with \( 0^* \) (because we can use the self-loop on \( q_1 \) any number of times) concatenated with 1, which is expressed as \( 0 \cup 10^*1 \). Similarly, the transition from \( q_0 \) to \( acc \) is \( \varnothing \cup (10^* \cup \varepsilon) \), which simplifies to just \( 10^* \).
(c) Create a 2-state GNFA equivalent to the GNFA in part (b), after removing the state \( q_0 \).
Again, we have provided the structure of this GNFA, so you only need to fill in the table below.
Try to simplify the resulting regular expression as much as possible.

(Note: A good way to check your answer is to convince yourself that the regular expression you find in this step is equivalent to the DFA we originally started with.)

<table>
<thead>
<tr>
<th>Edge</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( st \rightarrow acc )</td>
<td>((0 \cup 1^*1)^<em>10^</em>)</td>
</tr>
</tbody>
</table>

When we eliminate state \( q_0 \), the remaining transition arrow becomes

\[ \emptyset \cup (\varepsilon \circ (0 \cup 1^*1)^* \circ 10^*) \]

We can simplify this to \((0 \cup 1^*1)^*10^*\). This regular expression is the simplest that can be found for “contains an odd number of ones.”

Common mistakes: Many groups confused the empty set transitions with empty string transitions. Many groups made the mistake of trying to account for extra transitions between the states (apart from the one transition prescribed by the algorithm - \( r_i (r_{rip}^* \circ r_j) \)) after removing one state and in the process, ended up not following the algorithm and made mistakes. Some submissions had an equivalent final answer but could not demonstrate the understanding of the GNFA reduction algorithm which was being tested in this question.

Problem 3 (10 points)

Convert the following DFA into a regular expression. For full credit, you must show a GNFA for each step (five states, four states, three states, two states). You can choose whether to label the edges or use a table.
Solution:

Once you get the hang of converting DFAs to regular expressions, it's slightly simpler to label the edges directly with regular expressions. The first step is to transform the 3 state DFA into a 5 state GNFA by adding a new start state \( st \) with an \( \varepsilon \) transition to \( q_0 \) (the old start state) and a new accept state \( acc \) with an \( \varepsilon \) transition from \( q_0 \) (the only accept state in the original machine). For convenience, we omit any arrows that would be labeled with (empty set).

The simplest way to start is to eliminate state \( q_2 \) first. The only transition that changes is the arrow from \( q_1 \) to \( q_0 \). We need to union the original label \( (a) \) with the regular expressions that describes computations going \( q_1 \) to \( q_2 \) to \( q_0 \),
which is $ba^*b$. This gives us the label $a \cup ba^*b$, as shown here.

It’s apparent that the next easiest step is proceed from a four state GNFA to a three state GNFA by eliminating q1. The only transition that changes is the loop from q0 back to itself. In the next diagram we replace the label on this transition with the union of the original label (b) and the regular expression that describes computations moving from q0 to q1 and back to q1. This part is given by $a(a \cup ba^*b)$.

Putting it all together, and simplifying we get $b \cup aa \cup aba^*b$.

Finally we eliminate the last remaining state from the original DFA. We end up with one label on the arrow from st to acc. This arrow is labelled with $\varepsilon \circ (b \cup aa \cup aba^*b)^* \circ \varepsilon$, which simplifies to $(b \cup aa \cup aba^*b)^*$. Now that we have a GNFA with just two states, we can read the equivalent regular expression off the last remaining arrow which is $(b \cup aa \cup aba^*b)^*$.

Common mistakes: When using the GNFA reduction algorithm to find the regular expression equivalent to the DFA, it is important to follow the algorithm as is. Here again, many groups made the mistake of trying to account for extra transitions between the states (apart from the one transition prescribed by the algorithm - $r_i(r_{rip} \ast) r_j$) after removing one state and in the process ended up not following the algorithm and made mistakes. Also, many groups wrongly marked empty set transitions as either empty string transitions or some other non-empty transitions. Some groups chose to reduce the GNFA in the order q0 -> q1 -> q2, which made the regular expressions really long and cumbersome to handle and therefore led to mistakes. It
is not wrong to reduce the GNFA in any order but identifying the nodes which lead to simpler regular expressions can save significant time and effort.

Problem 4 (10 points)

Given a string \( w = x_1x_2 \ldots x_n \) its reversal is \( w^R = x_n \ldots x_2x_1 \). For example: \((abbc)^R = cbba\).

Given a language \( L \subseteq \Sigma^* \) we define its reversal is \( L^R = \{ w^R : w \in L \} \).

In this problem, we will prove in two different ways that if \( L \) is a regular language then \( L^R \) is also a regular language. In other words, regular languages are closed under reversal.

a. First prove that regular languages are closed under reversal using the fact that a language is regular if and only if a DFA or NFA recognizes it. Let \( L \) be a regular language and let \( M = (Q, \Sigma, \delta, q_0, F) \) be a DFA such that \( L = L(M) \). Construct an NFA (or DFA) \( N \) such that \( L^R = L(N) \) and give a proof that your construction is correct.

Solution:

We will define an NFA \( N = (Q', \Sigma, \delta', q'_0, F') \) that recognizes \( L^R \). (Note that the alphabet \( \Sigma \) remains the same.)

We will create a new state \( q_{new} \) and define the states of \( N \) to be \( Q' = Q \cup \{ q_{new} \} \). The initial state of \( N \) will be \( q'_0 = q_{new} \), and the accepting states of \( N \) will be a set that contains only the initial state of \( D \); that is, \( F' = \{ q_0 \} \). We will define the new transition function \( \delta' \) by adding epsilon edges from \( q_{new} \) to all of the accepting states \( F \) from the original DFA, and by reversing all of the arrows in the original DFA. Formally, we will define \( \delta' \) as follows:

\[
\delta'(q_{new}, \varepsilon) = F \\
\delta'(q, x) = \{ p \in Q | \delta(p, x) = q \} \text{ if } q \neq q_{new} \text{ and } x \neq \varepsilon \\
\delta'(q, x) = \{ \} \text{ otherwise}
\]

Next, we need to prove the correctness of this construction. That is, we need to show that the language recognized by \( N \) is \( L^R \); specifically, we’ll show that a string \( w \) is accepted by \( D \) if and only if \( w^R \) is accepted by \( N \). In what follows, we will write \( w = x_1x_2 \ldots x_n \) and its reversal as \( w^R = x_n \ldots x_2x_1 \).

First, note that \( w \) is accepted by \( D \) if and only if there exists a sequence of states \( (r_0, r_1, \ldots, r_n) \) such that \( r_0 = q_0, r_n \in F \), and for each \( i \) in \( 0, \ldots, n-1 \), \( r_{i+1} = \delta(r_i, x_{i+1}) \).

(See Sipser p. 40.) Then, note that we can rewrite \( w \) as \( w^R = y_1y_2 \ldots y_my_{n+1} \), where \( y_1 = \varepsilon \) and \( y_i = x_{n+2-i} \) for all \( i \in 2, \ldots, n+1 \). That is, we can write \( w^R = \varepsilon x_{n+1}x_{n-1} \ldots x_2x_1 \). Then, note that the sequence of states \( (q_{new}, r_n, r_{n-1}, \ldots, r_2, r_1, r_0) \) satisfies the definition of NFA acceptance.
for $N$ on input $w^R$ (see Sipser p. 54) if and only if $(r_0, r_1, \ldots, r_n)$ satisfies the definition of DFA acceptance for $D$ on input $w$. This is because our construction of $N$ defines $q_0' = q_{new}$ and guarantees the following biconditionals:

$$r_0 = q_0 \Leftrightarrow r_0 \in F', \quad r_n \in F \Rightarrow r_n \in \delta(q_{new}, \varepsilon),$$

and

$$r_{i+1} = \delta(r_i, x_{i+1}) \Leftrightarrow r_i \in \delta(r_{i+1}, x_{i+1}) \text{ for all } i \in 0, \ldots, n - 1.$$

Finally, we use Corollary 1.40 from the textbook to conclude that $L^R$ is a regular language.

Common Mistakes: In the construction of the machine $N$ that recognizes the reverse, the mistakes that materially affected the correctness of the construction involved the start state of $N$. Several groups chose to take an arbitrary accept state $q_f \in F$ of the original machine and designate this the start state of $N$; then $\varepsilon$-transitions were added to all of the other states of $F$. This is not correct because if we add an $\varepsilon$-transition from $q_f$ to $q_f'$ in $F$ we can introduce other strings into the language of the NFA $N$. Imagine a computation that starts in $q_f$, meanders around $N$, eventually returns to $q_f$, takes the $\varepsilon$-transition to $q_f'$ and then eventually moves to the accept state. This computation is not the reverse of any accepting computation in $M$. Thus we see that a new start state is needed; this way the $\varepsilon$-transitions can only be used before we reach the states in $N$.

Some added $\varepsilon$ to $\Sigma$ in the construction of $N$. Remember that $\varepsilon$ is never in an alphabet; it is a string of length zero, not a character.

In the correctness proof, many groups only proved that $L^R \subseteq L(N)$. It’s necessary to prove the converse that $L(N) \subseteq L^R$ (or use a proof structured with “if and only if” statements like the above). This amounts to proving that any string accepted by $L(N)$ is actually in the language $L^R$. This is where the constructions with the above mistakes would fail; the extra epsilon arrows would introduce strings to $L(M')$ not found in $L^R$.

b. Next, prove that regular languages are closed under reversal using the fact that a language is regular if and only if some regular expression describes it. Let $L$ be a regular language and let $R$ be a regular expression such that $L = L(R)$. We need to show that there exists a regular expression $R'$ such that $L^R = L(R')$. Hint: To do this, give a proof by induction on the size of the regular expression $R$. The size of a regular expression is the number of symbols used in it. Use Definition 1.52 (Sipser p.64) and notice that there are three base cases (items 1-3) and three inductive cases (because items 4-6 are defined recursively in terms of smaller regular expressions).

Solution:
Let $L$ be a regular language and let $R$ be a regular expression such that $L = L(R)$. We will give our proof by induction on the size of $R$. Remember that we want to show there exists a regular expression $R'$ corresponding to $L^R$.

Our base case will be that our language $L$ corresponds to a regular expression $R$ of size 1. By Definition 1.52, this is actually three base cases, as there are three types of regular expression of size 1:

1. $R = a$, for some $a \in \Sigma$. Then $L = L(R) = \{a\}$, because there’s only one string in the corresponding language. But then $L^R = \{a^R\} = \{a\} = L$, and $L^R$ is regular because it corresponds to the regular expression $a$ as well. Thus we can take $R' = R = a$.

2. $R = \varepsilon$. Then $L = L(R) = \{\varepsilon\}$. The reverse of the empty string is still the empty string, so $L^R = \{\varepsilon\} = L(R)$. So we can take $R' = R = \varepsilon$.

3. $R = \emptyset$. Then $L = L(R) = \emptyset$, so the reversed language $L^R = \emptyset$ as well. We can take $R' = R = \emptyset$.

It will help to clarify our inductive step by explicitly stating both the inductive hypothesis and what we need to show. Let $k > 1$ be an integer. Our hypothesis is “For every regular language $L$ which corresponds to a regular expression $R$ of size less than $k$, the reverse of the language, namely $L^R$, is also regular.” The statement we need to prove is: “For every regular language $L$ which corresponds to a regular expression $R$ of size $k$, the reverse $L^R$ is also regular.”

To prove this, we start with a regular language $L$ which corresponds to a regular expression $R$ of size $k$. In order to use the inductive hypothesis, we need to break down $R$ into smaller regular expressions. So what does $R$ look like? Refer again to Definition 1.52 and the six possibilities. We assumed $k > 1$ (because we already handled the base case) so $R$ cannot be possibilities (1) - (3). That leaves three possibilities for $R$.

In item (4), we have that there exist two regular expressions $R_1$ and $R_2$ such that $R = (R_1 \cup R_2)$. Thus we have that $L = L(R) = L(R_1) \cup L(R_2)$. By the definition of the reverse of a language we have that $L^R = L(R_1)^R \cup L(R_2)^R$.

Furthermore, we can express the size of $R$ as size($R$) = size($R_1$) + size($R_2$) + 3. Thus we see that, in particular, the size of $R_1$ and $R_2$ are each less than $k$. Then by the inductive hypothesis, the reverse of both $L(R_1)$ and $L(R_2)$ are regular. That means there exist regular expressions $R_1'$ and $R_2'$ such that $L(R_1)^R = L(R_1')$ and $L(R_2)^R = L(R_2')$. Then we see that $L^R$ corresponds to the regular expression $R' = R_1' \cup R_2'$, and so $L^R$ is regular.

In item (5), we have that there exist two regular expressions $R_1$ and $R_2$ such that $R = (R_1 \circ R_2)$. Thus we have that $L = L(R) = L(R_1) \circ L(R_2)$. By the definition of the reverse of a language we
have that $L^R = L(R_2)^R \circ L(R_1)^R$. Note that we changed the order of the two languages being concatenated when we reversed them.

Furthermore, we can express the size of $R$ as $\text{size}(R) = \text{size}(R_1) + \text{size}(R_2) + 3$. Thus we see that, in particular, the size of $R_1$ and $R_2$ are each less than $k$. Then by the inductive hypothesis, the reverse of both $L(R_1)$ and $L(R_2)$ are regular. That means there exist regular expressions $R'_1$ and $R'_2$ such that $L(R_1)^R = L(R'_1)$ and $L(R_2)^R = L(R'_2)$. Then we see that $L^R$ corresponds to the regular expression $R' = R'_2 \circ R'_1$, and so $L^R$ is regular.

In item (6), we have that there exists a regular expression $R_1$ such that $R = (R_1^*)$. Thus we have that $L = L(R) = L(R_1)^*$. By the definition of the reverse of a language we have that $L^R = (L(R_1)^R)^*$.

Furthermore, we can express the size of $R$ as $\text{size}(R) = \text{size}(R_1) + 3$. Thus we see that, in particular, the size of $R_1$ is less than $k$. Then by the inductive hypothesis, the reverse of $L(R_1)$ is also regular. That means there exists a regular expression $R'_1$ such that $L(R_1)^R = L(R'_1)$. Then we see that $L^R$ corresponds to the regular expression $R' = (R'_1)^*$, and so $L^R$ is regular.

Therefore, by showing that the reverse of every language corresponding to a regular expression also has a regular expression, we have shown that the reverse of every regular language is regular.

Common Mistakes:

1) Induction needs to be applied here, to know that the reversals of $R_1$ and $R_2$ are regular. You need to clearly state the induction variable here.

2) Having a RE like $R_1a$ is a good case to test your hypothesis, it is not actually general enough, since the RE is not restricted to be of this form.

Problem 5 (10 points)

a. Prove that the language $L = \{0^m1^01 \mid m \geq n\}$ is not regular.

b. Prove that the following language $L$ (which contains all binary strings that are not palindromes) is not regular: $L = \{w \in \{0, 1\}^* \mid w \neq w^R\}$

Solution

(a) Suppose, for the sake of contradiction, that $L$ is regular. Then we know that there must exist a positive integer $p$ satisfying the premises of the pumping lemma.
Let \( s = 0^p 1^p 1 \) and note that \( s \in L \). Now, consider all possible ways to express \( s \) as \( s = xyz \) such that \(|xy| \leq p\) and \(|y| > 0\). These cases can all be written as \( x = 0^k \), \( y = 0^m \), \( z = 0^{p-k-m} 1^p 1^p \), where \( m \geq 1 \), \( k \geq 0 \), and \( m + k \leq p \).

Now consider the string \( s' = xz = 0^k 0^{p-k-m} 1^p 1 = 0^{p-m} 1^p 1 \). Since \( m \geq 1 \), we know that \( s' \) is not in \( L \) because \( p - m < m \). Therefore, we have identified a string \( s \in L \) that cannot be "pumped", thus violating the pumping lemma. This is a contradiction, which means we can conclude that \( L \) must not be regular. Note that this is the technique that Sipser refers to as "pumping down" on p. 82.

(b) Suppose, for the sake of contradiction, that \( L \) is regular. Then the complement of \( L \), namely \( \overline{L} = \{ w \in \{0, 1\}^* | w = w^R \} \), the language of palindromes must be regular too. By the pumping lemma, we know that there must exist a positive integer \( p \) which is a pumping length of \( \overline{L} \).

Let \( s = 0^p 1^p \) and note that \( s \in \overline{L} \) because \( s \) is a palindrome. Now, consider all possible ways to express \( s \) as \( s = xyz \) such that \(|xy| \leq p\) and \(|y| > 0\). These cases can all be written as \( x = 0^k \), \( y = 0^m \), \( z = 0^{p-k-m} 1^p \), where \( m \geq 1 \), \( k \geq 0 \), and \( m + k \leq p \).

Now consider the string \( s' = xy^2z = 0^{p+m} 1^p \). Since \( m \geq 1 \), we know that \( s' \) is not in \( \overline{L} \). Therefore, we have identified a string \( s \in \overline{L} \) that cannot be "pumped", thus violating the pumping lemma. This is a contradiction, which means we can conclude that our original assumption is wrong, and \( L \) must not be regular.

Common mistakes: part b

- Many people said that \( \overline{L} = \{ ww^R | w \in \{0, 1\}^* \} \) which is incorrect because this leaves out strings such as 0 and 1
- Many people choose an explicit string for \( s \) (e.g. \( s = 1011 \)) but \( s \) needs to be a string whose length is in terms of \( p \) (the pumping length)
- Many people choose explicit strings for \( x \), \( y \), and \( z \) (e.g. \( s = 1^{p-1} 01^p \)) and then they would state that \( x = 1^{p-1} \), \( y = 0 \), \( z = 1^p \). This is incorrect because \(|xy| \leq p \) by the pumping lemma and the pumped string must hold for all values of \( x \), \( y \), and \( z \) so it is possible that \( y \) contains some of the leading 1s.)
- Many people did not state what value of \( i \) they used and simply stated that the pumped string would not be in the language.
- Many people said that \(|xy| = p \) (the correct expression is \(|xy| \leq p \))
- Many people defined \( x \) and \( y \) but not \( z \)
Problem 6 (10 points)

According to the statement of the pumping lemma, every regular language has a **pumping length** $p$ such that every string $x \in L$ can be pumped if $|x| \geq p$. The **minimum pumping length** is the smallest number $p$ such that $p$ is a pumping length for $A$. For each of the following languages, what is the minimum pumping length $p$? Justify your answer; this means you must also explain why $p-1$ is not a pumping length for $L$

a. $110^*11$

b. $(01^*0^*) \cup 1111$

**Solution**

(a) The minimum pumping length is 5. If $s$ is a string in $L$ of length at least 5, then $s = 110^k11$ for some $k \geq 1$. We can then pump $s$ by taking $x = 11$, $y = 0$ and $z = $ rest of the string. However, if $s$ is a string in $L$ of length 4, then $s = 1111$, which cannot be pumped because every string in $L$ contains exactly four ones. But conditions 2 and 3 of the pumping lemma imply that we must take $y$ to be 1, 11, 111, or 1111. Then $xy^2z$ is not in $L$, so this violates condition 1 of the pumping lemma. Thus the pumping length cannot be less than 5, and so 5 is the minimum pumping length.

**Common mistake:**
Several groups have stated that 1111 cannot be pumped without giving any justification. You need to provide an explanation for this statement.

(b) 5 is the minimum pumping length. It’s easy to see that 4 cannot be the pumping length for $L$ because 1111 is the only string in $L$ that starts with a 1 and therefore it cannot be pumped. If we were to divide $1111 = xyz$ with $y \neq \epsilon$ then $y$ would consist of one or more 1s. Then $xy^2z$ would consist of five or more 1s and is not in the languages.

To show that 5 is a pumping length, let $s \in L$ be a string of length at least 5. $s$ must start with 0 and we can classify $s$ according to its next symbol.
If the second symbol of \( s \) is 0, then we know so has the form \( 0^n \) and so we can set \( x = 0, y = 0, \) and let \( z \) be all but the first two symbols of \( s \). Then for any value of \( i \geq 0 \), we know that \( xy^iz \) will have the form \( 0^{(i-1)}0^i \) which matches \( 01^*0^* \) and is thus in the language.

If the second symbol is 1, then we can set \( x = 0, y = 1, \) and let \( z \) be all but the first two symbols of \( s \). Then \( z \) must be a string consisting of zero or more 1s followed by zero or more 0s. Then for any value of \( i \geq 0 \), we know that \( xy^iz \) will consist of 0 followed by some number of 1s, followed by zero or more 0s and thus is in \( L \).

In each of these cases, \(|y|\) is 1 (which satisfies condition 2 of the pumping lemma), \(|xy| \leq 5\) (satisfies condition 3), and we have shown that for any \( i \geq 0 \), the string \( xy^iz \) belongs to \( L \) (satisfies condition 1).

**Common mistake:**
Several groups have argued that minimum pumping length of the language is 2. This is not true as every string with length greater than or equal to \( p \) should satisfy the conditions of pumping lemma. As 1111 is in the language and cannot be pumped, the minimum pumping length cannot be 2.

(c) 2 is the minimum pumping length. Recall from Homework 1, Problem 5 that we can express \( L \) as the language of strings that start and end with the same symbol. We consider the case of strings of length 2 separately from length 3 or greater.

If \( s \in L \) has length 2 then \( s \) is either 00 or 11. Each of these strings can be pumped by taking \( x = \epsilon, y = \) the first character. If \( s \in L \) has length 3, then set \( x = \) the first symbol, \( y = \) the second symbol, and \( z = \) the rest of the string. Then for any \( i \geq 0 \), the string \( xy^iz \) belongs to \( L \) because the first and last character of \( xy^iz \) are the same as the first and last character of \( s \) and thus match each other.

We still need to argue that the pumping length cannot be 1. This time the analysis is different because each string of length 1 in \( L \), namely 0 and 1, can actually be pumped. However consider the string \( s = 101 \). If 1 were the pumping length, then the only choice for \( x, y, \) and \( z \) that satisfies the conditions of the pumping lemma is \( x = \epsilon, y = 1, z = 01 \) because we must have \(|xy| \leq 1\) and \(|y| \geq 1\) which implies \(|x| = 0, |y| = 1\). But then \( xy^0z = xz = 01 \) which is not in \( L \).

**Common mistake:**
Some groups have tried to prove that 2 is pumping length by setting \( y \) to be the first two symbols in the input. However, this approach would fail as removing the first two symbols can make the resulting strings to be not a part of the language.