1. Write regular expressions for the following languages over the alphabet $\Sigma = \{0, 1\}$. Hint: some of these languages may include $\varepsilon$.

(a) The set of all strings whose 1st, 3rd, 5th, …, characters are the same.

**Solution:** We divide this set into two cases: strings that start with 0 and strings that start with 1. Notice that $\varepsilon$ should be included.

$$(0(0|1))^*0? \ | \ (1(0|1))^*1?$$

(b) The set of all strings that represent the concatenation of one odd number, one even number, and another odd number expressed in binary. (E.g., 011001, but not 0110.)

**Solution:** In binary, odd numbers always end with a 1 and even numbers always end with a 0.

$$(0|1)^*1 (0|1)^*0 (0|1)^*1$$

(c) The set of all strings, except the string 0000.

**Solution:** Split into those strings with a 1 and those that do not. Notice that $\varepsilon$ is included in this language.

$$(0|1)^*1(0|1)^* \ | \ \varepsilon \ | \ 0 \ | \ 00 \ | \ 000 \ | \ 000000^*$$
2. Draw DFAs for each of the languages from question 1. Note that a DFA must have a transition defined for every state and symbol pair. You must take this fact into account for your transformations. Your DFAs should not have more than 10 states.

Notice that a short regular expression does not automatically imply a DFA with few states, nor vice versa.

(a) The set of all strings whose 1\textsuperscript{st}, 3\textsuperscript{rd}, 5\textsuperscript{th}, \ldots, characters are the same.

\textbf{Solution:}

\begin{figure}[h]
\centering
\begin{tikzpicture}

\t\node[state] (s0) at (0,0) {$0$};
\t\node[state] (s1) at (2,0) {$0$};
\t\node[state] (s2) at (4,0) {$1$};
\t\node[state] (s3) at (2,-2) {$0,1$};
\t\node[state] (s4) at (0,-2) {$0,1$};
\t\node[state] (s5) at (2,-4) {$0$};

\t\draw (s0) edge[above] node {$0,1$} (s1);
\t\draw (s1) edge[above] node {$0$} (s2);
\t\draw (s1) edge[below] node {$1$} (s5);
\t\draw (s2) edge[above] node {$1$} (s3);
\t\draw (s3) edge[above] node {$0,1$} (s4);
\t\draw (s4) edge[loop right] node {$0,1$} (s4);
\end{tikzpicture}
\end{figure}

(b) The set of all strings that represent the concatenation of one odd number, one even number, and another odd number expressed in binary. (E.g., 011001, but not 0110.)

\textbf{Solution:}

\begin{figure}[h]
\centering
\begin{tikzpicture}

\t\node[state] (s0) at (0,0) {$0$};
\t\node[state] (s1) at (2,0) {$1$};
\t\node[state] (s2) at (4,0) {$0$};
\t\node[state] (s3) at (6,0) {$1$};
\t\node[state] (s4) at (8,0) {$1$};

\t\draw (s0) edge[above] node {$0$} (s1);
\t\draw (s1) edge[above] node {$1$} (s2);
\t\draw (s2) edge[above] node {$0$} (s3);
\t\draw (s3) edge[above] node {$1$} (s4);
\t\draw (s4) edge[loop right] node {$1$} (s4);
\end{tikzpicture}
\end{figure}

(c) The set of all strings, except the string 0000.

\textbf{Solution:}

\begin{figure}[h]
\centering
\begin{tikzpicture}

\t\node[state] (s0) at (0,0) {$0$};
\t\node[state] (s1) at (2,0) {$0$};
\t\node[state] (s2) at (4,0) {$0$};
\t\node[state] (s3) at (6,0) {$0$};
\t\node[state] (s4) at (8,0) {$0$};
\t\node[state] (s5) at (10,0) {$0$};

\t\draw (s0) edge[above] node {$0$} (s1);
\t\draw (s1) edge[above] node {$0$} (s2);
\t\draw (s2) edge[above] node {$0$} (s3);
\t\draw (s3) edge[above] node {$0,1$} (s4);
\t\draw (s4) edge[loop right] node {$0,1$} (s4);
\t\draw (s0) edge[loop below] node {$0,1$} (s0);
\t\draw (s1) edge[loop below] node {$0,1$} (s1);
\t\draw (s2) edge[loop below] node {$0,1$} (s2);
\t\draw (s3) edge[loop below] node {$0,1$} (s3);
\t\draw (s4) edge[loop below] node {$0,1$} (s4);
\end{tikzpicture}
\end{figure}
3. Using the techniques covered in class, transform the following NFAs over the alphabet \{a, b, c\} into DFAs. Your DFAs should not have more than 10 states. Note that a DFA must have a transition defined for every state and symbol pair, whereas a NFA need not. You must take this fact into account for your transformations. Hint: Is there a subset of states the NFA transitions to when fed a symbol for which the set of current states has no explicit transition? Also include a mapping from each state of your DFA to the corresponding states of the original NFA. Specifically, a state \( s \) of your DFA maps to the set of states \( Q \) of the NFA such that an input string stops at \( s \) in the DFA if and only if it stops at one of the states in \( Q \) in the NFA.

Tip: for readability, states in the DFA may be labeled according to the set of states they represent in the NFA. For example, state \( q_{012} \) in the DFA would correspond to the set of states \( \{q_0, q_1, q_2\} \) in the NFA, whereas state \( q_{13} \) would correspond to set of states \( \{q_1, q_3\} \) in the NFA.

(a) Original NFA:

Solution: DFA:

Correspondences (DFA to NFA):

\[
\begin{align*}
q_0 &\mapsto \{q_0\}, & q_01 &\mapsto \{q_0, q_1\}, & q_02 &\mapsto \{q_0, q_2\}, \\
q_{012} &\mapsto \{q_0, q_1, q_2\}, & q_0 &\mapsto \emptyset.
\end{align*}
\]
(b) Original NFA:

Solution: DFA:

Correspondences (DFA to NFA):

\[ q_2 \mapsto \{q_2\}, \quad q_{02} \mapsto \{q_0, q_2\}, \quad q_{012} \mapsto \{q_0, q_1, q_2\}, \quad q_0 \mapsto \emptyset. \]
(c) Original NFA:

\[
\begin{array}{c}
\text{start} \\
q_0 & q_1 & q_2 & q_3 \\
a, b, c & a, b, c & a, b, c & a, b, c \\
\end{array}
\]

**Solution: DFA:**

\[
\begin{array}{c}
\text{start} \\
q_{02} & q_{12} & q_{13} & q_{23} & q_3 \\
a & b & a, c & b & a, b, c \\
a, b, c & b & b & c & a, b, c \\
a, b, c & a, c & c & a, c & a, b, c \\
\end{array}
\]

Correspondences (DFA to NFA):

- \( q_{02} \mapsto \{ q_0, q_2 \} \)
- \( q_{0123} \mapsto \{ q_0, q_1, q_2, q_3 \} \)
- \( q_2 \mapsto \{ q_2 \} \)
- \( q_{12} \mapsto \{ q_1, q_2 \} \)
- \( q_{123} \mapsto \{ q_1, q_2, q_3 \} \)
- \( q_3 \mapsto \{ q_3 \} \)
- \( q_{13} \mapsto \{ q_1, q_3 \} \)
- \( q_{23} \mapsto \{ q_2, q_3 \} \)
- \( q_0 \mapsto \emptyset \)
4. Let $L$ be a language over $\Sigma = \{a, b, c\}$, such that string $w$ is in $L$ if and only if $w$ is not $\varepsilon$ and the last character of $w$ appears at most twice in $w$.

Examples of strings in $L$: $aa$, $aba$, $bababac$.

Examples of strings not in $L$: $\varepsilon$, $bbb$, $cabaa$.

Draw an NFA for $L$. Your solution should have no more than 15 states.

Solution:
5. Consider the following tokens and their associated regular expressions, given as a **flex** scanner specification:

```
%%
01?1             printf("apple");
0(10)+10          printf("banana");
(1011*0|0100*1)   printf("coconut");
```

Give an input to this scanner such that the output string is 

\[(\text{apple})^3((\text{banana})^2 \text{ coconut})^2,\]

where \(A^i\) denotes \(A\) repeated \(i\) times. (And, of course, the parentheses are not part of the output.) You may use similar shorthand notation in your answer.

**Solution:**

\[(011)^3((01010)^2 01001)^2\]
6. Recall from the lecture that, when using regular expressions to scan an input, we resolve conflicts by taking the largest possible match at any point. That is, if we have the following **flex** scanner specification:

```flex
%%
do { return T_Do; }
[A-Za-z_][A-Za-z0-9_]* { return T_Identifier; }
```

and we see the input string “dot”, we will match the second rule and emit T_Identifier for the whole string, not T_Do.

However, it is possible to have a set of regular expressions for which we can tokenize a particular string, but for which taking the largest possible match will fail to break the input into tokens. Give an example of no more than two regular expressions and an input string such that: a) the string can be broken into substrings, where each substring matches one of the regular expressions, b) our usual lexer algorithm, taking the largest match at every step, will fail to break the string in a way in which each piece matches one of the regular expressions. Explain how the string can be tokenized and why taking the largest match won’t work in this case.

As a challenge (not necessary for credit), try to find a solution that only uses one regular expression.

**Solution:** Consider the following scanner for a JavaScript-like language that has both `==` and `===` operators:

```flex
%%
==   { return T_Equal; }
===   { return T_StrictEqual; }
```

and the string “====”. This can be broken into ‘==’, followed by ‘==’. However, the largest possible match strategy will first consume the beginning of the string as ‘===’, then stop when it finds that the remainder of the input is just ‘=’, which can’t be matched to any token.