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 that do not contain the substring $00$.

**Solution:**

$$0?(1^+0^?)^*$$

(b) The set of all strings that contain at least three 1s.

**Solution:**

$$0^*10^*10^*1(1|0)^*$$

(c) The set of strings where all characters must appear in consecutive pairs (i.e. 00 or 11). Examples of strings in the language: $\varepsilon$, 000011, and 11. Examples of strings not in the language: 11100, 00100, and 11000).

**Solution:**

$$(00|11)^*$$
2. For each regular language from problem 1, follow the instructions for each part below to convert the corresponding regular expression to a finite automaton or automata. 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 finite automata 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 that do not contain the substring 00.

- Convert your regular expression from (1a) to an NFA. The algorithm to convert a regular expression into an NFA is covered in lecture 4: [http://web.stanford.edu/class/cs143/lectures/lecture04.pdf](http://web.stanford.edu/class/cs143/lectures/lecture04.pdf)
- Convert your NFA to a DFA. The algorithm to convert an NFA into a DFA is covered in lecture 4.
- 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. If a state in your DFA does not map to any states of the original NFA, just put a null set in lieu of the NFA states. Tip: for readability, states in the DFA may be labeled according to the set of states they represent in the NFA. For example, state $s_{012}$ in the DFA would correspond to the set of states $\{s_0, s_1, s_2\}$ in the NFA, whereas state $s_{13}$ would correspond to set of states $\{s_1, s_3\}$ in the NFA. For an empty (reject) state, please use the label `empty` for readability.

**Solution:**

NFA:

![NFA Diagram](image)

DFA:

![DFA Diagram](image)

Correspondences (DFA to NFA):

- $s_{013} = \{s_0, s_1, s_3\}$
- $s_{13} = \{s_1, s_3\}$
- $s_{123} = \{s_1, s_2, s_3\}$
• *empty* = ∅

(b) The set of all strings that contain at least three 1s.

- *Draw a DFA for this language.*

**Solution:** DFA:

(c) The set of strings where all characters must appear in consecutive pairs (i.e. 00 or 11). Examples of strings in the language: \( \varepsilon, 000011, \) and 11. Examples of strings not in the language: 111000, 001000, and 11000).

- *Draw a DFA for this language.*

**Solution:** DFA:
3. Let $L$ be a language over $\Sigma = \{a, b, c\}$, such that string $w$ is in $L$ if and only if at least one character either does not show up in $w$ or only shows up as a single contiguous substring in $w$. That is, each string $w$ must have the form $px^nq$, where $x \in \Sigma$, $p$ and $q$ are strings that do not contain $x$, and $x^n$ denotes $x$ repeated $n$ times. $n$ may be 0.

Examples of strings in $L$: $\varepsilon$, $aa$, $ab$, $bababccccccccabababa$.

Examples of strings not in $L$: $abcabc$, $bbacbca$.

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

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

```
%%
11?0 printf("apple");
(10)*0? printf("banana");
(0110+|1001*1) printf("coconut");
```

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

\[
((\text{coconut})^3(\text{banana})^4)^2(\text{apple})^3,
\]

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:**

\[
((0110010010110)(10100)^4)^2(110)^3
\]
5. 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.