Problem 1 [40 points]: Consider the alphabet $\Sigma = \{a, b\}$.

a) Consider the Non-Deterministic Finite Automaton (NFA) depicted below. Why is this automaton non-deterministic? Explain the various source on indeterminacy.

b) Do the sentences $w_1 = \text{“aba”}$ and $w_2 = \text{“aab”}$ belong to the language generated by this FA? Justify.

c) Convert the NFA in part a) to a DFA using the subset construction. Show the mapping between the states in the NFA and the resulting DFA.

d) Minimize the DFA using the iterative refinement algorithm discussed in class. Show your intermediate partition results and double check the DFA using the sentences $w_1$ and $w_2$.

Solution:

a) This is indeed a NFA for two reasons. First, it includes $\varepsilon$-transitions. Second, in state 2 there are two transitions on the same terminal or alphabet symbol, "a".

b) Regarding the word “aba” there is a path from state 0 to the accepting state 6, namely: 0, 4, 5, 6. Regarding the word “aab” the automaton will never be able to reach the state 6 as in order to spell out the “b” character will necessarily be in state 5 and to reach that state we cannot have two consecutive “a” characters we we would be either in state 1 or state 4.

c) Using the subset construction we arrive at the following subsets and transitions.

$$S_1 = \varepsilon\text{-closure } \{\emptyset\} = \{0, 2, 4\} \text{ – this is not a final state.}$$

$$S_2 = \text{DFA}edge(S_1, a) = \varepsilon\text{-closure } (\text{goto}(S_1, a)) = \{1, 3, 5, 6\} \text{ – final state}$$

$$S_3 = \text{DFA}edge(S_1, b) = \varepsilon\text{-closure } (\text{goto}(S_1, b)) = \{2\}$$

$$S_9 = \text{DFA}edge(S_2, a) = \varepsilon\text{-closure } (\text{goto}(S_2, a)) = \{1, 3, 6\} \text{ – final state}$$

$$S_4 = \text{DFA}edge(S_2, b) = \varepsilon\text{-closure } (\text{goto}(S_2, b)) = \{2, 5, 6\} \text{ – final state.}$$

$$S_5 = \text{DFA}edge(S_3, a) = \varepsilon\text{-closure } (\text{goto}(S_3, a)) = \{3, 5, 6\} \text{ – final state.}$$

$$\text{DFA}edge(S_3, b) = \varepsilon\text{-closure } (\text{goto}(S_3, b)) = \{2\} = S_3$$
DFAedge(S4,a) = ε-closure (goto(S4, b)) = \{3, 5, 6\} = S5  – final state.
DFAedge(S4,b) = ε-closure (goto(S4, b)) = \{2, 5, 6\} = S4  – final state.
S6 = DFAedge(S5,a) = ε-closure (goto(S5, a)) = \{3, 6\}  – final state.
S7 = DFAedge(S5,b) = ε-closure (goto(S5, b)) = \{5, 6\}  – final state.
DFAedge(S6,a) = ε-closure (goto(S6, a)) = \{3, 6\} = S6  – final state.
S8 = DFAedge(S6,b) = ε-closure (goto(S6, b)) = \{\}  – final state.
DFAedge(S7,a) = ε-closure (goto(S7, a)) = \{\} = S7  – final state.
DFAedge(S7,b) = ε-closure (goto(S7, b)) = \{5, 6\} = S7  – final state.
DFAedge(S8,a) = ε-closure (goto(S8, a)) = \{\} = S8
DFAedge(S8,b) = ε-closure (goto(S8, b)) = \{\} = S8
DFAedge(S9,a) = ε-closure (goto(S9, a)) = \{1, 3, 6\} = S9
DFAedge(S9,b) = ε-closure (goto(S9, b)) = \{2\} = S3

This results in the DFA shown below with starting state S1.

d) We can try to minimize this DFA by using the iterative refinement partitioning described in class. The figure below depicts a possible sequence of refinements. For each step we indicate the criteria used to discriminate between states in the previous partition. As can be seen, the algorithm leads to no further refinement, so this is a particular case where the sub-set algorithm does yield a minimal DFA.
Problem 2 [30 points]: Consider the DFA below with starting state 1 and accepting state 2:

a) Describe in English the set of strings accepted by this DFA.
b) Using the Kleene construction algorithm derive the regular expression recognized by this automaton simplifying it as much as possible.

Solution:

a) This automaton recognizes all non-null strings over the \{a, b\} alphabet that end with "b".
b) The derivations are shown below with the obvious simplification.

Expressions for k = 0
\[ R_{11}^0 = a \mid \epsilon \]
\[ R_{12}^0 = \epsilon \]
\[ R_{21}^0 = a \]
\[ R_{22}^0 = b \mid \epsilon \]

Expressions for k = 1
\[ R_{11}^1 = R_{11}^0 (R_{11}^0)^* R_{11}^0 \mid R_{11}^0 = (a\epsilon) \cdot (a\epsilon)^* \cdot (a\epsilon) \mid (a\epsilon) = a^* \]
\[ R_{12}^1 = R_{11}^0 (R_{11}^0)^* R_{12}^0 \mid R_{12}^0 = (b\epsilon) \cdot (a\epsilon)^* \cdot (b) \mid (b) = ba^* \mid ba^*b \]
\[ R_{21}^1 = R_{21}^0 (R_{21}^0)^* R_{21}^0 \mid R_{21}^0 = (a) \cdot (a\epsilon)^* \cdot (b\epsilon) \mid (a) = a^* \mid a'b \]
\[ R_{22}^1 = R_{21}^0 (R_{21}^0)^* R_{22}^0 \mid R_{22}^0 = (a) \cdot (a\epsilon)^* \cdot (b \mid \epsilon) \mid (b \mid \epsilon) = (a^* \mid b) \]

Expressions for k = 2
\[ R_{11}^2 = R_{11}^1 (R_{11}^0)^* R_{11}^1 \mid R_{11}^1 = (ba^* \mid ba^*b) \cdot (a^* \mid b^*) \cdot (a^* \mid a'b) \mid (a^*) \]
\[ R_{12}^2 = R_{11}^1 (R_{11}^0)^* R_{12}^1 \mid R_{12}^1 = (ba^* \mid ba^*b) \cdot (a^* \mid b^*) \cdot (a^* \mid a'b) \mid (ba^* \mid ba^*b) \]
\[ R_{21}^2 = R_{21}^1 (R_{21}^0)^* R_{21}^1 \mid R_{21}^1 = (ba^* \mid ba^*b) \cdot (a^* \mid b^*) \cdot (a^* \mid a'b) \mid (a^* \mid a'b) \]
\[ R_{22}^2 = R_{12}^1 (R_{12}^0)^* R_{22}^1 \mid R_{22}^1 = (ba^* \mid ba^*b) \cdot (a^* \mid b^*) \cdot (a^* \mid a'b) \mid (a^* \mid a'b) \]

\[ L = R_{12}^2 = (ba^* \mid ba^*b) \cdot (a^* \mid b^*) \cdot (a^* \mid a'b) \mid (ba^* \mid ba^*b) \]

As can be seen the simplification of any of these regular expressions beyond the expressions for k=1 is fairly complicated. This method, although correct by design leads to regular expressions that are far from being a minimal or even the most compact representation of the regular language the DFA recognizes.
Problem 3 [10 points] Let \( L \) be a regular language over a finite alphabet \( \Sigma \). Show that the language consisting of all strings not in \( L \) over the same alphabet is also regular.

Solution: If \( L \) is a regular language then there exists a DFA \( M_1 \) that recognizes it. Now given \( M_1 \) we can construct a DFA \( M_2 \) that recognizes the complement of \( L \) with respect to the input alphabet \( \Sigma \) thus showing that the language recognized by \( M_2 \) is also regular. The DFA \( M_2 \) is a replica of \( M_1 \) but converting all accepting states to non-accepting states and vice versa. The starting state of \( M_2 \) is the same state as in the original \( M_1 \) DFA. Each input string is only accepted by \( M_2 \) iff the path spelled by its characters in the DFA leads to an accepting state. This means that by construction the same string would not be accepted in \( M_1 \). Conversely, the input string spells out a path to a non-accepting stage in \( M_2 \), than it would spell out a path to an accepting state in \( M_1 \). In effect, \( M_2 \) recognized the string not in the language recognized by the original DFA \( M_1 \) thus proving our claim.

Problem 4 [20 points]: Prove (using an informal but convincing argumentation) the pumping lemma for regular languages: If \( L \) is a regular language, then there is an integer \( n \geq 1 \) such that any string \( w \) of length \( |w| \geq n \) can be written as \( xyz \) where \( xy \) has length \( |xy| \leq n \), and \( y \) is not epsilon and for all \( i \geq 0 \), \( xy^iz \) is also in \( L \).

Hint: Choose \( n = \text{number of states in some DFA to recognize } L \) and use the pigeonhole principle.

Solution: Without loss of generality we focus on DFA as DFA and regular languages are equivalent representations of the same set of strings. we further assume there is a single acceptance or final state of the DFA. We therefore observe that for an this DFA automaton that accepts string of arbitrary length, there needs to be a "loop" in the automaton. Otherwise the length of the accepted strings is limited to the number of states of the automaton as you end out of states while spelling the accepted strings (recall you have no loops in this assumption). As a consequence, there must exist a path from the start state to the acceptance state. Given that there is a loop in this path and the fact that all transitions are non-empty (under the assumption this is a DFA) you can traverse the path corresponding to that loop any number of times you would like and still generate a string that belongs to the language accepted by that DFA. This is the case as everytime you visit the "start" state (\( S_{\text{begin}} \)) of this loop (or loops) you have no memory as to how often you have been at that particular state. As a result, if you accept the string after visiting that notorious state \( S_{\text{begin}} \), then you will also accept the string that includes the string spelled out by the "loop", hence proving the pumping lemma.