Here is a detailed, formal exposition of the conversion of a DFA into an equivalent regular expression, based on the dynamic programming idea we described in class. I am giving you these notes because this exposition differs quite a bit from that in your textbook. You are required to read and understand this material completely. For your exams, please be prepared to write proofs with this level of detail.

**Theorem:** If the language \( L \) is recognized by a DFA, then it is generated by a regular expression.

**Proof:** Let \( M = (\{q_1, \ldots, q_n\}, \Sigma, \delta, q_1, F) \) be a DFA that recognizes \( L \). For \( i, j \in \{1, \ldots, n\} \) and \( k \in \{0, \ldots, n\} \), let \( R^k_{ij} \) denote the set of all strings in \( \Sigma^* \) that take the DFA \( M \) from state \( q_i \) to state \( q_j \) without “going through” any state numbered higher than \( q_k \). By “going through” we mean both entering and leaving, so the starting point \( q_i \) and the end point \( q_j \) are allowed to be numbered higher than \( q_k \). To make this precise, we use a function \( \hat{\delta} : Q \times \Sigma^* \to Q \) with the following “meaning:”

Imagine putting \( M \) into state \( q \) and then feeding it the string \( w \in \Sigma^* \) as input. This leaves the DFA in a certain state after it processes the input; that state is denoted \( \hat{\delta}(q, w) \).

A precise definition of \( \hat{\delta} \) can be given using recursion, as follows:

\[
\hat{\delta}(q, \varepsilon) = q, \quad \forall q \in Q;
\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a), \quad \forall q \in Q, x \in \Sigma^*, a \in \Sigma.
\]

Using \( \hat{\delta} \), we can define \( R^k_{ij} \) precisely:

\[
R^k_{ij} = \left\{ x \in \Sigma^* : \hat{\delta}(q_i, x) = q_j \text{ and } x \text{ does not have a prefix } y, \text{ with } y \neq \varepsilon \right\}
\]

and \( y \neq x, \text{ such that } \hat{\delta}(q_i, y) = q_m \text{ where } m > k. \)

Now we shall prove, by induction on \( k \), that each of the sets \( R^k_{ij} \) is generated by a regular expression. The base case is \( k = 0 \). According to the definition, a string in \( R^0_{ij} \) must take \( M \) from \( q_i \) to \( q_j \) without going through any intermediate states at all, because every state of \( M \) is numbered higher than \( q_0 \). This makes it clear that

\[
R^0_{ij} = \{ a \in \Sigma : \delta(q_i, a) = q_j \}, \quad \text{if } i \neq j, \text{ and}
\]

\[
R^0_{ii} = \{ \varepsilon \} \cup \{ a \in \Sigma : \delta(q_i, a) = q_i \}.
\]

The important thing is that these are finite sets, so each of them is generated by a simple regular expression that simply lists out all the strings in the set and combines them using “\(|\)”. We now turn to the induction step. Suppose \( 1 \leq k \leq n \). The strings in \( R^k_{ij} \) can be divided into two classes: those that do not take \( M \) through state \( q_k \) and those that do. The strings in the former class clearly do not take \( M \) through any state numbered higher than \( q_{k-1} \); therefore these strings are in fact in \( R^{k-1}_{ij} \). It is also clear that a string which takes \( M \) from \( q_i \) to \( q_j \) avoiding states numbered higher than \( q_{k-1} \) also avoids states numbered higher than \( q_k \); so, \( R^{k-1}_{ij} \) is exactly the set of strings in the former class.

Now let us focus on the latter class; let \( x \) be a string in this latter class. When \( M \) is put in state \( q_i \) and fed the input \( x \), it must, at some point, reach state \( q_k \) for the first time and, at some point, leave \( q_k \) for the last time. In other words, it must be possible to write

\[
x = uvw
\]

where \( u \) takes \( M \) from \( q_i \) to \( q_k \) without going through any state numbered higher than \( q_{k-1} \) and \( w \) does the same from state \( q_k \) to \( q_j \). In other words, \( u \in R^{k-1}_{ik} \) and \( v \in R^{k-1}_{kj} \).

What about \( v \)? It takes \( M \) from \( q_k \) to \( q_k \) without going through any state numbered higher than \( q_k \), but it may go through \( q_k \) itself several times (zero or more times), say \( t \) times. Then we can write

\[
v = v_1 v_2 \ldots v_{t+1}
\]
where each \( v_r \) (for \( r \in \{1, \ldots, t+1\} \)) takes \( M \) from \( q_k \) to \( q_k \) without going through \( q_k \). In other words, each \( v_r \in R_{kk}^{k-1} \) and so \( v \in (R_{kk}^{k-1})^* \). Combining this with our observations about \( u \) and \( w \), we have

\[
x = uvw \in R_{ik}^{k-1} \left( R_{kk}^{k-1} \right)^* R_{kj}^{k-1}.
\]

Thus, the strings in \( R_{ij}^k \) in the latter class all belong to \( R_{ik}^{k-1} \left( R_{kk}^{k-1} \right)^* R_{kj}^{k-1} \). It is also clear that, conversely, a string in \( R_{ik}^{k-1} \left( R_{kk}^{k-1} \right)^* R_{kj}^{k-1} \) definitely takes \( M \) through state \( q_k \) and never through a state numbered higher than \( q_k \). Therefore, \( R_{ik}^{k-1} \left( R_{kk}^{k-1} \right)^* R_{kj}^{k-1} \) is exactly the set of strings in the latter class.

Combining the two classes of strings in \( R_{ij}^k \), we get

\[
R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1} \left( R_{kk}^{k-1} \right)^* R_{kj}^{k-1}.
\]

By our induction hypothesis, each of the sets on the right-hand side of this equation is generated by a regular expression. Combining these regular expressions using the union, concatenation and star operators gives us a regular expression for \( R_{ij}^k \). This completes the induction step.

Having completed our induction proof, we address the question of writing a regular expression for the language \( L \). Clearly, \( x \in L \) iff \( x \) takes \( M \) from its start state \( q_1 \) to some final state \( q_i \in F \) without going through a state numbered higher than \( q_n \) (there are no states numbered higher than \( q_n \)!) Thus,

\[
L = \bigcup_{q_i \in F} R_{1i}^n,
\]

and since we have proved that each set \( R_{1i}^n \) is generated by a regular expression, and the above union is a finite union, we see that \( L \) is also generated by a regular expression. This completes the proof of the theorem. QED.

**Exercise:** Show that the above proof can in fact be modified to yield a dynamic programming algorithm that takes as input a description of a DFA and outputs a regular expression equivalent to it.