(1) Text, exercise 1.4 (p83 in the 3rd ed.), parts (e) and (g) only. You may either follow the text’s directions to use the Cartesian product construction, or you may design the final machine directly—but in the latter case it is imperative to have a descriptive comment saying what current condition on the portion of the input read so far is enforced by the state. (To compare: in my library lights example, state 2 had the comment label “A off, B on” and enforced the condition that you would be in that state if and only if the sequence of switch-flip characters read so far would leave light A off and light B on.) The languages in the textbook are:

(e) \{x \in \{a,b\}^*: x \text{ starts with } ‘a’ \text{ and } \#b(x) \leq 1\}.

(g) \{x \in \{a,b\}^*: |x| \text{ is even and } \#a(x) \text{ is odd}\}.

Answers in words: (e) Since \(\epsilon\) is not in the language \(L_e\) of (e), the start state \(s\) is not accepting. Since any string that begins with \(b\) is disqualified from belonging, the arc on \(b\) from \(s\) goes to a dead state, \(d\). The arc on \(a\) must go to an accepting state since \(a \in L_e\) (it has zero \(b\)s which complies with “at most one \(b\)”), so it can’t loop back to \(s\) or go to \(d\). Call the new state \(q_a\). Once we’ve fulfilled the “starts with \(a\)” requirement, the symbol \(a\) becomes a “don’t care” so it can self-loop at \(q_a\) and any more states we get. An arc on \(b\) from \(q_a\) is still OK, but it brings “danger” since another \(b\) will be too many. So it must go to an accepting state (call it \(q_{ab}\)) that is different from \(q_a\). Again we self-loop on \(a\) at \(q_{ab}\), but an arc on \(b\) from it goes to \(d\). That finishes the machine by logical design. We got \(Q = \{s,d,q_a,q_{ab}\}\), \(F = \{q_a,q_{ab}\}\), and

\[ \delta = \{(s,a,q_a),(s,b,d),(q_a,a,q_a),(q_a,b,q_{ab}),(q_{ab},a,q_{ab}),(q_{ab},b,d),(d,a,d),(d,b,d)\}. \]

(g) Picture yourself in a boat on a river, with tangerine trees, marmelade skies, and banks labeled “Even \(|w|\)” and “Odd \(|w|\)” where \(w\) stands for the portion of the input \(x\) read so far. Your boat starts on the even bank and crosses the river on each character. If the char is \(b\) then it goes straight across, but if \(a\) it crosses diagonally downstream or back upstream. The girl with kaleidoscope eyes is Downstream on the even bank; if you call that state \(D\) then the language is \(L_{SD}\).

(2) Think about problem 1.6(e) in the text, asking to design a DFA \(M\) such that \(L(M) = A\), where

\[ A = \{x : x \text{ starts with } 0 \text{ and has even length, or starts with } 1 \text{ and has odd length}\}. \]

Instead, however, design an NFA \(N\) with a start state that forks out with two \(\epsilon\)-transitions (as in examples in lecture) such that \(L(N) = A\). Also write a regular expression \(R\) such that \(L(R) = A\). Then finally “nudge” your \(N\) into an equivalent DFA—this does not require the conversion to be shown in Tuesday’s lecture. Finally, answer, did the “fork trick” and NFA help you get the DFA, or was it/would it be just as easy to design the DFA directly?

Answer in words: Let \(M_0\) have a nonaccepting start state \(s_0\) that goes to dead on 1 and a nonaccepting state \(q_o\) on 0, then from \(q_0\) both chars go to an accepting state \(q_e\) (meaning: \(|w|\) has even length), from which both chars go back to \(q_o\). Also build \(M_1\) with start state \(s_1\) going to dead
on 0 but to an accepting state \( r_o \) on 1. Both chars go from \( r_o \) to a state \( r_e \) meaning “\(|w|\) has even length” and bumping both chars back to \( r_o \) like before, but this time \( r_e \) is rejecting. Then build the NFA \( N \) with an extra start state \( s \) that has \( \epsilon \)-arcs to \( s_0 \) and \( s_1 \). The regular expression this represents is

\[
0(0 + 1)((0 + 1)(0 + 1))^* + 1((0 + 1)(0 + 1))^*.
\]

Finally you can get the DFA by lumping \( s, s_0, s_1 \) together as one start state \( S \). No dead state is needed now: the dead case for \( M_0 \) is handled by the live case for \( M_1 \) and vice-versa. This gives you a DFA \( M \) with 5 states, and that is fine.

But in fact you can now combine states \( q_0, r_e \) and \( q_e, r_o \), yielding a 3-state DFA answer. This seems to destroy the meanings of those four states given above, so how do we know we can get away with it? The Myhill-Nerode coverage will explore this kind of issue. As for the final question, either answer is fine but here is one way of seeing it: If you design the DFA directly, it’s at the level of problem (1g) or the “Lights A and B” example from lecture. Whereas \( M_0 \) may feel simpler and \( M_1 \) is just a morph of \( M_0 \)—then the NFA lets you combine them without doing Cartesian products and the DFA suggested itself smoothly. It’s more steps but each step is smaller—so it’s like the issue with whether to evolve code in several small steps or do a (re)write from scratch.

(3) Text, exercise 1.13 (page 85 in the 3rd. ed.): Call a string \( x \in \{0, 1\}^* \)”bad” if it has two 1’s that are separated by an odd number of symbols. For instance, 10001 is bad and so is 10011 because although the 1’s in places 0 and 3 are fine, and the 11 in places 3 and 4 is fine, the first and last 1’s are not. But \( \epsilon \) and 1001 are not bad—call them “good.” Design a DFA with 5 states that recognizes the language of good strings. The text suggests first designing a 4-state NFA that recognizes the bad strings as a stepping-stone, but you may find it just as helpful to start by answering the question, can any string with three 1’s in it be good? (18 pts., for 66 total on the set)

**Answer in words:** Starting with the last question, let any string \( x \) with three (or more) 1’s be given. If the first two 1’s in \( x \) are separated by an odd number \( k \) of symbols, then \( x \) is already bad, so let \( k \) be even. Now consider the second and third 1 letting \( m \) be the number of symbols between them. If \( m \) is odd then \( x \) is cooked, so let \( m \) be even. Then however the first and third 1 are separated by \( k + 1 + m \) symbols which in this leftover case is odd. So \( x \) is bad any way you slice it.

Thus we have a case where \( x \) is good by default if it has at most one 1 and bad if 3 or more, so the only critical case is \( x \) having two 1’s. That depends on whether the number of 0’s between them is odd or even. Leading and trailing 0’s are benign, so put a self-loop \( (s, 0, s) \) but a new state \( q_1 \) for \( (s, 1, q_1) \). Now from \( q_1 \) a 1 goes to \( q_2 \) which is still accepting, but any further 1 from \( q_2 \) trips the “three or more” bad condition and goes to a dead state d (while \( q_2, 0, q_2 \) treads water). Finally from \( q_1 \) a 0 goes to a fifth distinctive state—call it \( q_3 \)—from which a 1 is now bad (so \( q_3, 1, d \) happens) but a 0 goes back to \( q_1 \) since the chars-between are even again. All states except d are accepting. We get: \( Q = \{s, d, q_1, q_2, q_3\} \), \( F = Q \setminus \{d\} \), and

\[
\delta = \{(s, 0, s), (s, 1, q_1), (q_1, 0, q_3), (q_1, 1, q_2), (q_2, 0, q_2), (q_2, 1, d), (q_3, 1, d), (q_3, 0, q_1)\}
\]

plus the two dead-state self-loops.