Reading: Tuesday’s lecture will cover the theorem from section 1.2 that converts any given NFA into an equivalent DFA. One extra point to note is that when the NFA has \(k\) states, the DFA need not use all \(2^k\) subset-states. Breadth first search will find only those subset-states that are possible and needed, often saving much work. Thursday’s lecture will close the ring of conversions by showing how to convert any NFA—hence any DFA too since a DFA “Is-A” NFA—into a regular expression for the same language. This last conversion picks up from page 70 of the book on the topic of “Generalized NFAs” which I regard not as “real code” but a visual bookkeeping device. All told, the conversions show that DFAs, NFAs, and regular expressions all represent the same class of languages, which are called the regular languages. The following week will then move on to non-regular languages, in which I will veer away from the “Pumping Lemma” by using instead the Myhill-Nerode Theorem which is covered in the chapter exercises—and which is a UB (and Cornell) product.

Assignment 2, due in hardcopy; please staple any multiple-sheet submission

(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 \(12+12 = 24\) pts. Note correction to (e).

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

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

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

Instead, however, design an NFA \(N\) with a start state that forks out with two \(\epsilon\)-transitions (as in examples in lecture) such that \(L(N) = A\). 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? \((9+6+6+3 = 24\) pts. Update: I switched 0 and 1 from the text, but please do it as worded above—or write your final DFA and regexp both ways. “Teachable moment”: write descriptive comments to help in cases of “silly switching.”)

(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)