(A) Repeating part of the long question completing the answer: "Homework 1.3" says to convert every Boolean formula into *conjunctive normal form* (CNF). [algorithm skipped] An example of a formula in CNF is

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3).$$

It is not a tautology—it can be made false by the *assignment* $x_1 = 0$, $x_2 = 1$, and $x_3 = 0$. ...Now for the *questions*:

(a) What must happen for a CNF formula to be a tautology? *Answer:* Each individual clause C must be a tautology. Since C is an OR of literals, this means each C contains both x_i and $\bar{x}_i$ for some i—else you could make it false.

(b) What happens when you apply the algorithm to the following two formulas? OK, this is tedious to do directly by the algorithm, but if you can tell whether either of them is a tautology you might find shortcuts.

$$\begin{aligned} ((x \wedge y) \to z) &\longleftrightarrow ((x \to z) \wedge (y \to z)) \\ ((x \wedge y) \to z) &\longleftrightarrow ((x \to z) \vee (y \to z)) \end{aligned}$$

Answer: The left-hand side is equivalent to

$$\neg(x \wedge y) \vee z, \qquad \text{which } \equiv \qquad \bar{x} \vee \bar{y} \vee z.$$

Thus the AND disappears, so the second not the first is the tautology, as you can see by expanding its right-hand side as $(\bar{x} \vee z) \vee (\bar{y} \vee z)$, removing the no-longer-needed parentheses, and using $z \vee z = z$. Applying the algorithm to expand the *longleftrightarrow* is horrible, but what happens is that it creates the "anti" of each literal which eventually come together into the clauses when you distribute $\vee$ over $\wedge$.

(c) What happens when you apply the algorithm to a formula that is already in *disjunctive normal form* (DNF), such as this one?

$$(r \wedge s \wedge t) \vee (u \wedge v \wedge w) \vee (x \wedge y \wedge z).$$

Brief answer: Recursively applying the distributive law to whole clauses and then within clauses duplicates a lot of literals. This one blows up to $3^3 = 27$ terms. When you have m terms in m variables each the blowup can be m^m which is exponential.

(d) The bottom line is that conversion to CNF provides a foolproof way to tell whether a formula is a tautology, but is the algorithm practical? *Answer:* Not when it has exponential blowup...

For-credit portion

(1) Design a deterministic finite automaton with alphabet $\{0, 1\}$ to recognize the language of binary numbers (in standard binary notation with leading zeroes allowed) that are multiples of 5. It is *your choice* whether you prefer to include the empty string λ in this language, or not—you must state your choice clearly. (*Design Hint:* Imitate the process of binary long division by 101 with states denoting the five possible "carry values.")

Then say what would happen if you represented the given number in 2-adic notation rather than standard binary. If you have used a logical design then the change might not be too painful. . . (18 + 9 = 27 pts.)

Answer in words: The DFA M needs 5 states, one q_i for each remainder $i < 5$ that could be "carried" when doing long division by 5. Since long division starts with a carry of 0, the start state is q_0. Since we divide evenly iff there is no carry at the end, we need a "carry 0" state to be the only accepting state. We can make q_0 play the latter role if we don't mind including λ in the language—as another name for zero. Doing so or not was OK so long as the intent was clearly commented. Leading 0s in the input binary number x can also be readily tolerated. Thus from q_0 reading 0 you can stay in q_0, whereas reading 1 takes you to q_1. From 1, 0 goes to q_2 and 1 goes to q_3, the latter since we are now carrying 11 which equals 3. From q_2, 0 takes you to q_4 since $2 = 10$ and $4 = 100$ in binary, A 1 however meshes with the carry $2 = 10$ to create $101 = 5$ and divide evenly up to there, so $\delta(q_2, 1) = q_0$. By the same logic:

$$\begin{aligned}
\delta(q_3, 0) &= q_{110} = q_1 \qquad \text{since } 110 = 6 \text{ and } 6 \bmod 5 = 1. \\
\delta(q_3, 1) &= q_{111} = q_2, \\
\delta(q_4, 0) &= q_3, \qquad \text{since } 100 \cdot 0 = 8 = 3 \text{ modulo } 5 \\
\delta(q_4, 1) &= q_4.
\end{aligned}$$

In 2-adic notation, the most important thing is that you have the same states with the same meanings because the meanings depend on the numbers, not on the notation. If you use the alphabet characters from the text as $1, 2$ then you can still treat λ as zero (and accept it) but there is no numeral zero. Now $\delta(q_0, 1) = q_1$ and $\delta(q_0, 2) = q_2$. The rules $11 = 3$ and $12 = 4$ in 2-adic notation dictate where to go from q_1. Meanwhile $21 = 5$ means you go from q_2 right back to q_0 on 1, and $22 = 6$ means $\delta(q_2, 2) = q_1$. To complete the machine we have:

$$\begin{aligned}
\delta(q_3, 1) &= q_2 \qquad \text{since } 111 = 7 \text{ and } 7 \bmod 5 = 2\ . \\
\delta(q_3, 2) &= q_3, \\
\delta(q_4, 1) &= q_4, \qquad \text{since } 12 \cdot 1 = 9 = 4 \mathit{modulo} 5 \\
\delta(q_4, 2) &= q_0.
\end{aligned}$$

If we preserve the alphabet $\{0, 1\}$ as stated in lecture to define the standard correspondence of $\{0, 1\}^*$ to $\mathbb{N}$, then rework the second machine with 1 in place of 2 and 0 in place of 1. But finally, if you were to adopt the lecture's suggestion of making the correspondence be to $\mathbb{N}^*$, so that λ corresponds to the number 1, then the design cycles all the way back to the first machine: The string for a positive binary number x just takes away the leading 1. So pretend you've already read that 1 by starting your original DFA M up in state q_1 rather than q_0. (Which gives another reason I don't like hard-wiring "q_0" to always mean the start state.)

(2) Consider regular expressions without Kleene stars—or equivalently, consider expressions written as unions and concatenations of languages, such as we did in writing the distributive law as $A \cdot (B \cup C) = (A \cdot B) \cup (A \cdot C)$. Define a notion of "disjunctive normal form" for such expressions and give an algorithm for converting arbitrary expressions with just union and concatenation to that form.

Suppose the basic languages in our expressions are just $\{c\}$ for characters c in the alphabet Σ, possibly including also the languages $\{\lambda\}$ and $\emptyset$ "just for show" (they won't change anything). Note that without the Kleene star operation you can't ever get an infinite language this way. But show that every finite language L has such a "star-free" expression, indeed one in your disjunctive normal form. Intuitively, how does this form relate to the set of strings in L?

Answer: A regexp in DNF is a union of *terms*, where each *term* is a concatenation of characters. Since a concatenation of characters is just a string, and a finite set L is a union of the singleton sets $\{s\}$ for all members $s \in L$, just listing out the members of L basically gives you a *-free regexp for L.