CSE491/596, Fall 2020 Problem Set 2 Due Mon. Oct. 5

Lectures and Reading. For this week read section 7 of Debray's notes. Then jump way ahead to section 13 to get the definition of time complexity, but stop at Theorem 13.6 (note that Theorem 13.4 is basically the same as the $\{a^n b^n\}$ example already started in lecture. The definition of space complexity is wedged in section 17 as Definition 17.5, but it's simpler to just read it here: the *space used* by a deterministic Turing machine M on an input x is the number of tape cells that are ever *changed* to a *different* character. Then M runs within space $s(n)$ if for all n and inputs x of length n, M on input x uses no more than $s(n)$ space. If M on input x does not halt, it is conventional to regard the space as well as time used to be infinite, but one can also define terms carefully to avoid needing to rely on this convention. *Then go back* and read section 6, which you'll notice already refers to "time" and "space" as if they were known informally from an algorithms course or somesuch. Finally, for next week, read section 8, but pause before the definition of mapping reductions and the examples on page 26, since we will go over to the notes by John Watrous in parallel with reading page 25.

The attitude of lectures will be to de-emphasize the character-level details of Turing machines once we show how to build a universal Turing machine—one that is fairly time-efficient, in fact. That is to say, we focus on a basic vocabulary of TM operations that suffice to emulate a rudimentary assembly language—for which see my hand-drawn hand-out https://rjlipton.files.wordpress.com/2014/05/utmramsimulator.png, which is enough to establish the principle of how TMs are equivalent to high-level programming languages. Once we have it, we dispense with char-level drawings of Turing machines and allow them to be *described in pseudocode*—provided the pseudocode is low-level enough to indicate the time and space complexity up to linear (or at least polynomial) factors.

This is a short homework. There will also be a Problem Set 3 due on Mon. Oct. 12, before **Prelim I** on **Friday, Oct. 16**. The exam will be held at two staggered times, 11am–noon ET and during the class period, both *remote only*.

(1) Use the Myhill-Nerode proof script to show that the following languages over $\Sigma = \{0, 1\}$ are not regular:

(a) $L_a = \{v1v : v \in \Sigma^*\}$.

(b) $L_b = \{v1w : v, w \in \Sigma^*, |v| = |w|\}$.

(c) $L_c = \{v1w : \#0(v) = \#0(w)\}$.

The points total 18 for all three languages together; you need not have three separate answers depending on how you organize things. Recall from lecture that $\#0(x)$ means the number of 0s in the string x.

(2) Now do $L_d = \{v1w : \#1(v) > \#0(w)\}$. (Be careful to note that the 1 shown could be anywhere in the given string x when you break it as $x = v1w$. 18 pts.)

(3) Design a two-tape Turing machine M such that $L(M) = L_c$ above and M runs in $O(n)$ time. Pseudocode is fine provided it is detailed enough to demonstrate running in $O(n)$ time. (12 pts., for 48 total).

Presentation Ideas. The first set of them will be staggered over two weeks. I will take 1 or 2 volunteers for this week in each group. Note that (b) and (c) relate to each other, so they are amenable to being handled by a team of 2.

(a) Suppose we have two DFAs $M_1 = (Q_1, \Sigma, \delta_2, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ and want to build a DFA $M_3 = (Q_3, \Sigma, \delta_3, s_3, F_3)$ such that $L(M_3) = L(M_1) \cup L(M_2)$. Consider these two possible strategies:

(i) Use the Cartesian product construction with union $\cup$ as the operator.

(ii) Make an NFA N_3 simply by adding an ϵ-arc from a new start state s_3 to s_1 and s_2, then convert N_3 to an equivalent DFA.

Is strategy (i) better than (ii), or could (ii) be better, or are they always about the same? Can the "exponential explosion" ever happen when starting with an NFA of the form N_3 made by putting two DFAs in parallel?

(b) Suppose we are given a DFA $M = (Q, \Sigma, \delta, s, F)$. Suppose we reverse all the arcs to make $M' = (Q, \Sigma, \delta', s, F)$ where $\delta' = \{(q, c, p) : (p, c, q) \in \delta\}$. Note that M' still has the same start and final states. Give examples with at east two states each where:

(i) M' winds up being the same machine as M ("isomorphic to M," in technical parlance).

(ii) M' is not a DFA but rather has nondeterminism at some state(s).

(c) With reference to the original $M = (Q, \Sigma, \delta, s, F)$ in (b), suppose also that F has just one state, f. Now suppose we define M' also by interchanging the start and final states, that is $M' = (Q, \Sigma, \delta', s', F')$ where $s' = f$ and $F' = \{s\}$ as well as $\delta' = \{(q, c, p) : (p, c, q) \in \delta\}$. Then M' literally runs computations by M backwards.

(i) For each $k \geq 1$, show how to build a DFA M_k such that M'_k is the NFA N_k shown in lecture such that $L(N_k) = (0 + 1)^*1(0 + 1)^{k-1}$.

(ii) Conclude that there are regular languages L whose reversal language L^R (namely, $L^R = \{x^R : x \in L\}$), while always still regular, requires exponentially more states for a DFA to accept it.

(d) Consider DFAs M that can take 3 input strings x, y, z, all of the same length, on 3 separate tapes. Each instruction has the form $(p, [c_1, c_2, c_3], q)$ and always advances the heads on all three tapes. Show that such machines can verify the addition relation $z = x + y$ when x, y, z are written in binary notation with their least significant bits first, and padded out with leading 0s to the same length if need be. Then for a variation, build a machine M' that takes just x and y on its first two tapes and outputs their sum $x + y$ on a third tape, again with least significant bit first. This time, M' is allowed to take an extra step at the end to write a final 1, if $x + y$ is a longer string in binary notation than x or y separately.