CS–172 Computability & Complexity, Spring 2021

Homework 6 Solutions

Note: These solutions are not necessarily model answers. Rather, they are designed to be tutorial in nature, and sometimes contain more explanation than is required for full points. Also, bear in mind that there may be more than one correct solution. The maximum total number of points available is 31.

1. (a) First, assume that L has a Turing enumerator E. We construct a multi-tape TM M for L as follows: On input x, M runs a simulation of E, using one of its tapes to represent the output tape of E. As soon as the special symbol $\#$ appears on this tape, it checks whether the string sandwiched between the last two occurrences of $\#$ equals x. If this is the case, M accepts; otherwise, it continues its simulation of E. Note that M will accept all strings in L and loop on all strings outside L. *5pts*

Now suppose that we are given a TM M that recognizes L. To construct an enumerator E for L, we want to run M on all possible inputs and output those that make M accept. However, we must be careful because there may be inputs on which M never halts, and we do not want E to spend all its time processing those inputs. One way to achieve this is the following: Let $x_0, x_1, x_2, \ldots$ be the sequence of all strings listed in lexicographic order. The machine E will run in infinitely many stages. In stage t, E will simulate M on inputs $x_0, \ldots, x_t$ for t steps. For all inputs x_i accepted by M within this time bound, M outputs $x_i\#$. Now if a string x_i is in L, then M accepts x_i in t steps for some t, so E will output x_i by stage $i + t$ at the latest. If x_i is not in L, then no simulations on input x_i will ever accept, so E will never output x_i.

[*For the second implication above, some students simply simulated M on each input $x_0, x_1, x_2 \ldots$ without limiting the number of steps in the simulations. This procedure may get stuck on one simulation that never halts, and is therefore not guaranteed to try all possible inputs. This is an important point—please make sure you understand it!!!*]

(b) If L has a Turing enumerator E that outputs strings in L in lexicographic order, the following machine M will decide L: On input x, M runs a simulation of E until E produces either x or a string that is lexicographically bigger than x. In the former case, M accepts; in the latter case, it rejects. Conversely, given a machine M that decides L, we design a Turing enumerator E as follows: At stage t, E runs M on input x_t, where as above x_t is the tth string in Σ^* in lexicographic order, and outputs $x_t\#$ on its output tape only if M accepts x_t. Clearly strings in L will be output in lexicographic order. *3pts*

2. Let us first prove the simpler $\Leftarrow$ direction. That is, given that $L = \{x : \exists y \text{ with } (x, y) \in L'\}$ for some decidable L', we must show that L is r.e. To do this, we simply build a recogniser M for L using the given decider M' for L'. On input x, this recogniser runs through all possible y's in lexicographic order, and for each y it runs the decider M' to check if (x, y) is accepted by it: if so, it halts and accepts. (Note that the decider always halts, so there is no danger of running forever on any y.) Now, if $x \in L$ then there exists some y such that $(x, y) \in L'$, and so our recogniser will eventually find this y and hence accept x. On the other hand, if $x \notin L$ no such y exists, so the recogniser will in fact run forever. *8pts*

To prove the $\Rightarrow$ direction, we assume that L is r.e. and have to construct an L' with the stated properties, namely: (1) L' is decidable; and (2) L' contains at least one pair (x, y) for each $x \in L$, and no such pair for any $x \notin L$. The idea is to make y a *witness* or *certificate* of the fact that $x \in L$; the language L' is then simply pairs of x and valid certificates for x. There are many ways to design certificates; perhaps the

simplest is the following: let M be a recogniser for L (which we know exists since L is r.e.), and define y to be a certificate for $x \in L$ if y is (an encoding of) an accepting computation of M on x. I.e., we define L' by:

$$L' = \{(x, y) : y \text{ encodes an accepting computation of } M \text{ on } x\}.$$

Then clearly L' has property (2) above. To check property (1), we must show that L' is decidable. But this is easy: we can build a decider for L' that, on input (x, y), verifies that y is indeed an encoding of a valid accepting computation of M on x (using information about the transition function of M, which is hard-wired into the decider).

Another possible certificate is to make y encode (e.g., in binary, or in unary just by its length) the number of steps in an accepting computation of M on x. Then the corresponding L' can also be decided, simply by simulating M on x for the number of steps specified by y.

[*Most students who attempted this question did quite well on it. However, some students didn't submit a solution to it. Please work through the above solution carefully!*]

3. (a) To see that L_{int} is r.e., we build a machine M_{int} that takes as input a pair $\langle M_1, M_2 \rangle$ and proceeds in rounds. In round i, M_{int} simulates both M_1 and M_2 on the first i input strings (in lexicographic order) for i steps. If during some round both the machines end up accepting the same string, then M_{int} accepts. To see that M_{int} is a recogniser for L_{int}, note that if $\langle M_1, M_2 \rangle \in L_{\text{int}}$ then there is some string x that both accept: suppose they accept x after t_1, t_2 steps respectively. Moreover, suppose that x is the jth input string (in lexicographic order). Then our machine M_{int} will certainly terminate after $\max j, t_1, t_2$ rounds and accept (though it may accept sooner than that). Clearly M_{int} never accepts pairs $\langle M_1, M_2 \rangle \notin L_{\text{int}}$. Hence M_{int} is a recogniser for L_{int} and so L_{int} is r.e. as claimed. *3pts*

[*As in Q1(a), many students did the simulations of M_1, M_2 without bounding the number of steps. This leads to the same problem as in the note following Q1(a) above. This is an important point that you should be aware of!*]

(b) To show L_{int} is undecidable we shall give a mapping reduction f from A_{TM} as follows. The function f on input $\langle M, w \rangle$ will return $\langle M_1, M_2 \rangle$, an encoding of a pair of TMs M_1 and M_2 defined as follows. M_1 is a TM that first tests if its input is w, and if not immediately rejects; otherwise, it behaves like M on w and accepts if M accepts. Note that the only string that M_1 might accept is w; more precisely, we have that *4pts*

$$L(M_1) = \begin{cases} \{w\} & \text{if } M \text{ accepts } w; \\ \emptyset & \text{otherwise.} \end{cases}$$

The second TM, M_2, is just a trivial machine that accepts all inputs. Clearly the function f is computable: if we are given $\langle M, w \rangle$ we can easily construct $\langle M_1, M_2 \rangle$. To see that this is a valid reduction, we have to check that M accepts w *if and only if* $L(M_1) \cap L(M_2) \neq \emptyset$. First, suppose M accepts w; then $L(M_1) = \{w\}$, so $L(M_1) \cap L(M_2) = \{w\} \neq \emptyset$, as required. For the other direction, suppose $L(M_1) \cap L(M_2) \neq \emptyset$; this means that $L(M_1) \neq \emptyset$ and hence that M accepts w. This completes the verification of the reduction.

Since A_{TM} is undecidable and $A_{\text{TM}} \leq_m L_{\text{int}}$, we can conclude that L_{int} is undecidable.

(c) To show that L_{nint} is not r.e., note that L_{nint} is essentially $\overline{L_{\text{int}}}$. From parts (a) and (b), we know that L_{int} is r.e., so $\overline{L_{\text{int}}}$ is not r.e. (else both L_{int} and $\overline{L_{\text{int}}}$ would be decidable). *2pts*

There is the following familiar detail here: the above argument actually proves that $\overline{L_{\text{int}}}$, the complement of L_{int}, is not r.e. But L_{nint} is not exactly $\overline{L_{\text{int}}}$; rather, $\overline{L_{\text{int}}} = L_{\text{nint}} \cup L_{\text{junk}}$, where L_{junk}

denotes all strings that are not valid encodings $\langle M_1, M_2 \rangle$ of pairs of Turing machines. However, L_{junk} is clearly decidable, so if L_{nint} were r.e. then $\overline{L_{\text{int}}}$ would be r.e. also. Hence L_{nint} is not r.e.

[*We did not deduct points for failing to mention L_{junk} (but you should be aware of this detail).*]

4. Here is one reason to be skeptical about the POC: it could be used to check if an arbitrary Java program P halts when fed an arbitrary input x (as explained below). But the undecidability of the halting problem implies that there can be no Turing machine (and hence, by the Church-Turing thesis, no program written in any language) that decides whether an arbitrary program halts on an arbitrary input. *6pts*

To see how we can use the POC to solve the halting problem, take the program P and input x that you want to test and construct a new program P_x that ignores its real input and simply proceeds to run P on input x. Now if P loops forever on x, then P_x is just a program that loops forever on *all* inputs, so the perfectly optimized version of P_x is just a single line looping back on itself. So to check if P halts on x, we just submit P_x to the POC and check if the output is the single looping line. If it is, we can conclude that P does not halt on x, and otherwise we know that P does halt on x.

An alternative reason you might question the claims on the box is that the POC can be used to check if two programs are "equivalent", i.e., compute the same function or have the same behaviour. Given two programs P_1 and P_2, if they are exactly the same then the POC will output the same perfectly optimized version for both. So to check if two programs are equivalent, we can feed each of them to the POC and look to see if the outputs are identical. But we know that the problem of checking whether two TMs accept the same language is undecidable, and of course we could translate any TM program into a Java program, so testing equivalence of two Java programs is also undecidable.

[*Many students came up with some version of the following bogus argument: "The POC has to check whether the original Java code and the minimized Java code have the same input/output behavior. But checking equivalence is an undecidable problem, so the POC cannot exist." The argument is bogus for two reasons: first, it makes an unjustified assumption about how the POC works, as it could be that the inventors of the POC have come up with a code optimization method that does not require an explicit equivalence check; and second, even if the POC did check equivalence, then it would not be doing so for two arbitrary Java programs, but rather for one arbitrary program and a second program (namely, the output from the POC) that is related to it, so the worst-case undecidability of the equivalence problem does not apply here.*]