Homework #3 Solutions

1. Suppose I have a Turing machine M with a doubly-infinite tape, i.e. a tape that stretches both left and right to $x = \pm\infty$. Prove that such a machine can be simulated with a standard Turing machine M' whose tape is only infinite on one side, with x ranging from 0 to ∞. You don't have to describe the simulation in complete detail, but you should describe it well enough to say how many states and tape symbols M' has as a function of how many M has, and how long the computation time of M' will be in terms of that of M.

   Answer. Let a_x be the symbol at the ith location on M's tape. One way to do the simulation is to "fold" the tape of M to make that of M', as in

   $$a_0 a_1 a_{-1} a_2 a_{-2} a_3 a_{-3} \cdots$$

   Suppose the current simulated location x on M's tape is positive; then to simulate a step in which M moves left or right, M' moves two steps left or right. If x is negative, M' moves two steps in the opposite direction. By marking the end of the tape with a # to the left of a_0, M' can tell when x switches sign, and "bounces" the other way.

   There are many ways to handle the details. One is to give the head of M' two extra bits of information, one of which records the current sign of x (1 for $x > 0$ or 0 for $x \leq 0$) and the other of which allows M' to be in a "moving" mode while it is halfway through its two-step move. Adding two bits multiplies the number of states by 4, so M' then has 4 times as many states as M. The number of tape symbols of M' is the same as for M, plus one for the marker #. Finally, for every step of M, M' takes 2 steps, unless we bound off the left end in which case it takes 3. Thus if M takes T steps, M' takes at most $3T$ (presumably closer to $2T$ if we don't visit a_0 too often).

   There are many alternate solutions, such as moving every symbol one step to the right each time we want to extent the tape to the left. But this takes $O(W)$ time per step if there are W symbols on the tape, for a total of $O(WT) = O(T^2)$ time. □

2. A *queue automaton* or QA is like a PDA, but with a queue rather than a stack. That is, it is a finite-state automaton which can look at an input symbol, its own state, and the front symbol of a queue; its transition function allows it to change its state, pop the front symbol, and/or push a symbol onto the back of the queue.

   Show that, unlike PDAs which can only recognize context-free languages, QAs can simulate Turing machines. Hint: simulating T steps of the Turing machine might take the QA more than $O(T)$ steps. How long does it take?

Answer. We keep the TM's tape in the queue, with a marker # to mark its ends, and a symbol h just to the left of the head's current location. We include the TM's finite state as part of the QA's finite state. By popping off the front and pushing on the back, we can scan the queue circularly. Each time we do so, we can modify the symbols to the left and right of the head's position, by pushing slightly different symbols.

For instance, suppose we want to simulate a move in which the TM changes the current tape symbol from a to a' and moves to the right. Then after popping ha off the queue, we push $a'h$. Moving left is slightly trickier; we can do this by including a three-symbol buffer in the QA's state, so that we pop, say, bha (here b is the symbol to the left of the TM's current location) and push hba'.

Of course, there are many other ways to do this: if we mark the TM's current location with additional symbols rather than putting an h to the left of its current location, we only need a two-symbol buffer. We can also write the TM's state on the queue instead of including it in QA's finite state if we prefer.

Each step of the TM requires the QA to go through the entire queue, so if the TM takes T steps and uses a maximum of W sites on the tape, the QA's running time is $T' = O(WT)$. Since $W \leq T$ (the TM can only touch one site per step) we have $T' = O(T^2)$. □

3. For the simulation in the previous problem to work, it is vital that the QA be able to make ϵ-transitions. A *real-time* QA is one which is not allowed to make ϵ-transitions, i.e., it is required to take exactly one step of its transition function for each symbol of the input, and is given no additional time to think; it must accept or reject as soon as it has read the input.

   How powerful are real-time QAs? Give an example of a non-context-free language which can be recognized by a real-time QA. On the other hand, give an example of a context-free language which seems intuitively hard for a real-time QA to recognize (you don't need to give a proof).

   Answer. For an example of a non-context-free language which a real-time QA can recognize, consider the copy language with a marker in the middle,

$$L_{\text{copy}} = \{w\#w \mid w \in \{a,b\}^*\}$$

   We showed before that the copy language $\{ww\}$ is not context-free; the same proof works for this version.

   To recognize L_{copy}, the QA starts by pushing w onto the queue; when it sees the # it switches to popping, and checks that each symbol it pops off the queue matches the input. (Without a marker, we would need to non-deterministically guess when to switch.) We accept if the queue is empty at the end and has not gone empty before.

   However, just as the copy language is easy for QAs and hard for PDAs, for palindromes it's the other way round. The language

$$L_{\text{pal}} = \{w\#w^R \mid w \in \{a,b\}^*\}$$

is clearly context-free, but seems hard for a QA. The problem is that a queue is a first-in, first-out (FIFO) device, so strings come out in the same order we put them in. It's just as hard for a QA to reverse the order of w as it is for a PDA to repeat it in the same order. In particular, to recognize L_{pal}, a QA has to push the first half onto the queue in order to store it, but then it needs to compare the first symbol after # to the *last* symbol of the first half. But this symbol is at the tail of the queue, not the head, and a QA is only allowed to read the head.

Indeed, this intuition can be made rigorous. There's a nice paper on real-time queue automata which shows that the QA and PDA languages are incomparable (i.e., neither set is contained in the other) and also describes real-time queue languages in terms of a kind of "breadth-first grammar." $\square$

4. Suppose that instead of being able to move left or right, a Turing machine is only allowed to stay put or move right. Show that such a machine can only recognize regular languages.

   Answer. The idea is that the Turing machine can't go back and read what it wrote before. It can then be simulated by a finite-state automaton which keeps track of the TM's current state and tape symbol. We will describe this simulation in the deterministic case; the generalization to the nondeterministic case is straightforward.

   Suppose M is a TM with states Q, tape alphabet Γ, and transition function

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{0, +1\}$$

   where 0 and +1 indicate staying put or moving to the right. Then, define a DFA M' with transition function δ' as follows. Suppose M is in state q when it first moves rightward onto a symbol a of the input. There are several possibilities. If M accepts or rejects without ever moving farther to the right, M' makes a transition to an accepting or rejecting state in which it stays regardless of the rest of the input. If M never moves farther right (which means it falls into a loop) M' also rejects. Otherwise, let $\delta'(s, a)$ be the state in which M first moves to the right; clearly this is determined by q and a.

   Alternately, let the states of M' be $Q' = Q \times \Gamma$, and define an NFA M' which can make ϵ-moves. In this case we define $\delta' : Q' \times (\Sigma \cup \{\epsilon\}) \to Q'$ as

$$\begin{aligned} \delta'((q,\gamma),\epsilon) &= (q',\gamma') \quad \text{if} \quad \delta(q,\gamma) = (q',\gamma',0) \\ \delta'((g,\gamma),a) &= (q',a) \quad \text{if} \quad \delta(q,\gamma) = (q',\gamma',+1) \end{aligned}$$

   These two cases describe the TM staying still and changing its state and the current tape symbol, or moving right onto a new input symbol a (in which case the symbol γ' it writes at its previous location is irrelevant). Finally, let the accepting states of M' be $(q_{\text{accept}}, \gamma)$ for all γ. $\square$

5. Let's say that a language L is *enumerable in order* if there is a Turing machine that prints out the words in L in lexicographic order (although it might take arbitrary time to print each one). Show that if L is enumerable in order, then L is decidable.

   Answer. Since all finite languages are decidable (a hard-coded Turing machine can just look the input up) we focus on the case where L is infinite.

   If L is infinite, there is a word $v \in L$ which comes after w in lexicographic order, and the enumerator will print v in some finite time. If it prints w first, then $w \in L$; but if it hasn't printed w by the time it prints v, we know that $w \notin L$ since the enumeration is ordered. Since we know either way in a finite amount of time, L is decidable. $\square$

6. Write the pseudocode of a C program which eventually prints out every C program (which run on their own with no input) which halts. Feel free to use words like "compile" and "run for t steps", but make sure that every program which will ever halt is printed out by your program at some finite time.

   Now, is there such a program which prints out halting programs in order of increasing length? Give the pseudocode for such a program, or prove that none exists.

   Answer.

```
for (i=0 to infinity) {
  for (w ranging over all ascii strings of length i or less) {
    if w compiles,
      run w for up to i steps   // breadth-first search
      if w has halted by then, print w
  }
}
```

   To avoid getting stuck running a single non-halting program, we run each program for a fixed but growing amount of time; this is like the breadth-first search we do to simulate a non-deterministic TM with a deterministic one. (This program prints out each halting program many times; if we like we can maintain a list of what we've printed so far and check whether `w` is already in that list before we print it.)

   Note that the order in which the programs get printed out depends both on their length and on the number of steps they take before halting. If we could modify this to print them out in order of increasing length, the Halting Problem would be decidable using the proof for the previous problem: to tell whether a program halts, just wait to see whether it, or a longer program, gets printed first. $\square$

7. True or false: any subset of a decidable language is decidable. If true, give a proof; if false, give a counterexample.

   Answer. False. For a trivial example, let L_1 be the set of all strings of 0's and 1's, and let $L_2 \subset L_1$ be the set of strings which encode halting Turing machines. In a sense, decidability is about the complexity of the boundary between a set and its complement; a smaller set can have a much more complicated boundary. $\square$

8. The Goldbach Conjecture states that all even numbers can be written as the sum of two odd primes. Let L be the language

$$L = \begin{cases} Y & \text{if Goldbach's conjecture is true} \\ N & \text{if it isn't} \end{cases}$$

   Is L decidable?

   Answer. Yes. Either $L = \{Y\}$, or $L = \{N\}$. We don't know which it is, but either way it's decidable (and regular for that matter).

   The point here is that decidability, and more generally computational complexity, only enters when we ask an infinite family of questions. Any single question (or any finite number), no matter how complicated, can simply be answered by a lookup table. Once we do the necessary "preprocessing" to construct this table (which could be very difficult) the question becomes trivial. To put it differently, I don't know how to write a halting program which will answer Goldbach's question correctly; but I can show you two short programs, one of which is correct.

   On the other hand, answering general questions along the lines of Goldbach's conjecture is much harder. For instance, the set of true sentences of the form "for all x, there exists y such that some simple property of the integers x and y is true" is undecidable, and gets more undecidable as we add more alternating quantifiers. $\square$

9. A Turing machine is a *decider* if it halts on all inputs. IsDecider is the following problem: given the description of a Turing machine M, tell whether is a decider. Show that IsDecider is undecidable by reducing the Halting Problem to it.

   Answer. We wish to convert an example of the Halting Problem, which asks whether a TM M halts on an input string x, into an example of IsDecider. To do this we build a TM M' with the string x hard-coded into it; it ignores its input, writes x on its tape, and then acts like M. Clearly M' halts on all inputs if and only if M halts on x. $\square$

10. Now, show that the set of deciders is not even recursively enumerable! In other words, show that there is no Turing machine enumerator which prints out descriptions of all the deciders. This proves that telling whether a Turing machine halts on *all* inputs is actually harder than telling whether it halts on a particular input. Hint: suppose such an enumerator exists, and use diagonalization.

Answer. Assume that an enumerator E exists, and let D_i be the ith decider it prints out. We will use the diagonal trick to construct a decider $\overline{D}$ which differs from all the D_i, and thus prove by contradiction that E does not exist.

The idea is to set $\overline{D}(i) = \text{NOT}\, D_i(i)$, i.e., $\overline{D}$ accepts an input i if and only if D_i does not. In that case $\overline{D} \neq D_i$ for all i, since they differ on input i. We have to prove that $\overline{D}$ is a decider by showing we can implement it in finite time. To implement $\overline{D}$ on input i, first run E until it prints out the ith decider D_i; then use a universal Turing machine to simulate D_i on input i; then accept if and only if D_i rejects. Each of these steps takes finite time, and this completes the proof. □