Introduction to the Theory of Computation
Homework #4 Solutions

1. (Problem 3.11) 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' needs in terms of how many M has, and how long the computation time of M' will be in terms of that of M.

Answer: We can simulate an doubly-infinite tape T with a singly-infinite one T' by "folding" it. Thus positions $x' = 0, 1, 2, 3, 4\ldots$ on T' will correspond to positions $x = 0, 1, -1, 2, -2, \ldots$ on T. To simulate moving one step right on T, we move two steps right on T' if $x > 0$, and two steps left on T' if $x < 0$; if we hit the left end of T', we know that $x = 0$, in which case we can move one step right to $x = 1$ and know that $x > 0$. We simulate moving left on T similarly.

Our simulation T' runs half as fast as the original since it takes two steps for every step of T. It needs about four times as many states; for each state s of T, we need a state for each combination of s with (x is positive/negative) and (we're moved both steps/we're in the middle of moving two steps). The number of tape symbols is the same as for T. (There are other ways to do this, like using the tape symbol to mark whether x is positive or negative; this doubles the number of tape symbols but divides the number of states we need by two.)

2. (Problem 3.9) Show that a PDA with two stacks, i.e. a finite-state automaton which has two stacks it can examine, push, and pop, can simulate a Turing machine. (Note that it can push one stack while popping the other, so the stacks will not necessarily have the same depth.) Describe the simulation at the same level of detail as in the previous problem.

Now prove that a Turing machine can simulate a PDA with 3 or more stacks (if you like you can use the fact that a single-tape Turing machine can simulate a multitape Turing machine). Therefore, going from 1 to 2 stacks makes a PDA much more powerful, but going to 3 or more stacks doesn't increase its power.

Answer: To show that a 2-stack PDA can simulate a Turing machine, we let the two stacks simulate the left and right halves of the tape. Let's use the top symbol of the right-hand stack to represent the symbol at our current position on the tape. To move left on the tape, we pop a symbol from the left stack and push a (possibly re-written) symbol on the right stack, and to move right on the tape we pop from the right and push on the left.

How many steps of the 2-stack PDA this takes depends on exactly how we define it, but the computation time and the number of states both get multiplied by a small constant. The stack alphabet is simply the tape alphabet.

To show that a 3-or-more-stack PDA is no more powerful than a 2-stack one, all we have to do is note that a multitape Turing machine can easily simulate a multistack PDA, with each tape simulating a stack with the top symbol at its right end. To push, we write a symbol and move right; to pop, we simply move left. Then since multitape TMs can be simulated by single-tape TMs we're done.

3. A *queue automaton* is like a PDA, but with a queue rather than a stack. That is, it is a DFA which can look at the front symbol of a queue; based on that symbol and its finite state, it is allowed 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, a queue automaton can simulate a Turing machine.

Hint; simulating T steps of the Turing machine might take the queue automaton more than $O(T)$ steps. How long does it take?

Answer: Note that unlike a PDA, a queue automaton can "cycle through" its queue by popping off the front and pushing on the back. While it does this, it can look for a particular symbol x, and change it by pushing a modified symbol y (or a word of more than one symbol) by pushing y when it pops x. So, after each step of the TM, our queue contains a string which consists of the entire tape, with the TM's current position marked with an additional symbol indicating its state. (We also use markers for the left and right ends of the tape.) It's easy to see how, by cycling through the queue, the queue automaton can carry out one step of the Turing machine by reading its current state and the current tape symbol, popping them off the front, pushing symbols corresponding to the new state and new symbol on the back, and then continuing cycling through. (In order to move the machine to the left, in which case the TM marker should be pushed before the tape symbol we read before, we can keep a one-symbol "buffer" instead of immediately pushing the symbol we popped.)

If the queue has length L, each step of the TM then takes $O(L)$ steps of the queue. Since after T steps of the TM L is at most $n+T$ where n is the length of the input, the total time to simulate T steps is $O(LT) = O((n+T)T) = O(T^2)$.

4. (Problem 3.13) 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: Clearly the TM can never go back and look at what it wrote before. It reads the input left-to-right, and the "stay put" instruction corresponds to an ϵ-move which does not read another symbol of the input. Now a DFA M which can make ϵ-moves can be simulated by a standard DFA M' which can't (i.e. which reads a symbol at every step), since at each position in the input M either 1) gets stuck and never accepts, 2) accepts there, or 3) eventually moves on in some state, which we call the new state of M'. (Indeed, our definition of NFA allowed ϵ-moves, and we saw before that they only accept regular languages.)

We also need to think about what happens if the TM moves past the input into the blank section of the tape. In this case we just define A as the set of states for which, if it is in that state when it first enters the blank part of the tape, it will eventually accept. Then we add A to the accepting set of M'.

5. (Problem 3.19) Let L be the language containing the single string s on the alphabet $\{a, b\}$, where

$$s = \begin{cases} a & \text{if life exists on other planets within 100 light-years} \\ b & \text{if it doesn't} \end{cases} .$$

Is L decidable? Is it regular?

Answer: Either $L = \{a\}$ or $L = \{b\}$; I don't know which it is, but it's one of them, and both of them are clearly regular and decidable.

This may seem like a trick question `:-)` but the point is that we're not asking the computer to figure out whether there is life on other planets (or whether God exists, which was Sipser's version); we're just asking it to recognize, or print, a single symbol. Consider two programs, one which prints "yes" and the other which prints "no." One of these programs is right; I don't know which one, but they both run very quickly.

6. Recall that L is *decidable* if there is a Turing machine which always halts and, given input w, says "yes" if $w \in L$ and "no" if $w \notin L$. A language L is *recognizable* if there is a Turing machine which halts and says "yes" if $w \in L$, but may or may not halt if $w \notin L$. Show that a language L is decidable if and only if both L and its complement $\overline{L}$ are recognizable.

Answer: We did this in class. If L and $\overline{L}$ are both recognizable, we run both machines in parallel. One and only one of the machines will say yes; if the recognizer for L does then $w \in L$, and if the recognizer for $\overline{L}$ does then $w \notin L$.

Conversely, if L is decidable there is a machine which says "yes" if $w \in L$ (and no if $w \notin L$). This makes it a recognizer for L by definition, and switching "yes" and "no" makes it a recognizer for $\overline{L}$. Thus both L and $\overline{L}$ are recognizable.

7. Recall that a language L is *recursively enumerable* if there is a Turing machine M that prints out a list of words in L, such that any given word w in L will eventually appear somewhere in the list. (Another way to put this is that M takes an input i and prints out a word $f(i)$, and that $w \in L$ if and only if $w = f(i)$ for some i.)

Now suppose there is a Turing machine that prints out a list of words in L *in increasing order* (using dictionary order). Show that L is decidable.

Answer: L is either finite or infinite. If it's finite, it's decidable automatically (just look the input up in a finite list and say "yes" if it's there).

So suppose it's infinite. In that case, if $w \notin L$, M will eventually print out the next word in L in dictionary order beyond w; but then we know that $w \notin L$ since M will never come back and print w. So we watch M, and halt and say "no" if M prints out a word beyond w in dictionary order.

8. 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 according to their length? Give the pseudocode for such a program or prove that none exists.

Answer: We did this in class. The trick is to explore the two-dimensional space of "run program i for j steps," and there are several ways to do this.

If a program existed which prints out halting programs in increase length order, then the Halting Problem would be decidable by the argument in the previous question: if you want to know about a program P, wait until it either prints P or a program longer than P. In the latter case we would know that P will not halt.