CS 21 Problem Set #2
Ishaan Mantripragada.

Problem 1

a. A 2-NPDA is a 7-tuple $(Q, \Sigma, \Gamma, \Gamma^*, \delta, q_0, F)$ where: Q is the set of all states (finite set), Σ is the input alphabet (finite set), Γ is tape alphabet of stack 1 (finite set), Γ^* is the tape alphabet of stack 2 (finite set), $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times (\Gamma^* \cup \{\epsilon\}) \to P(Q \times (\Gamma \cup \{\epsilon\}) \times (\Gamma^* \cup \{\epsilon\}))$ is the transition function, q_0 is the start state where $q_0 \in Q$, and F is the set of accept states (finite set) where $F \subseteq Q$.

The 2-NPDA accepts an input string, s, if it can be written in the form $s = s_1 s_2 ... s_m$ where each $s_i \in \Sigma \cup \{\epsilon\}$, and there exists a sequence of states $c_0, c_1, ..., c_m$ and pairs of strings $(u_0, v_0), (u_1, v_1), ..., (u_m, v_m)$ where (u_i, v_i) represents the contents of the two stacks at each step, such that:

- $c_0 = q_0$ is the initial state.
- $(u_0, v_0) = (\epsilon, \epsilon)$ is the initial stack content.
- For each step i, the transition $\delta(c_i, s_{i+1}, a, b)$ determines the next state c_{i+1} and the next stack contents (u_{i+1}, v_{i+1}), based on the current state c_i , the next input symbol s_{i+1}, and the top of the stacks a and b. The next input symbol s_{i+1} can be ϵ if there is no more input to read, and similarly, a and b can be ϵ if the corresponding stack is empty.
- c_m is an accept state from the set F.

b. Let M be the 2-NPDA and let u and v be the two stacks that M uses. First, \$ gets pushed onto both u and v. Then, M begins reading a's from the tape. As it reads them, it pushes them onto stack u. Once it finishes reading all a's, it begins reading b's from the tape. As it reads b's it pushes them onto stack v and simulataneously pops a's from stack u. If M tries to pop an a from u that does not exist (i.e. there are more b's than a's) then it rejects. Also, if after reading all b's from the tape, stack u is not empty (i.e. there are more a's than b's), it rejects. After reading all b's it then begins reading c's from the tape. Again simultaneously, it pops b's from stack v. Again if M tries to pop a b from v that does not exist (i.e. there are more c's than b's) then it rejects. Finally, if stack v is not empty after reading all c's from the tape (i.e. there are more b's than c's), it rejects. Otherwise, it accepts.

c. To prove that 2-NPDAs are equivalent to Turing Machines, we can show that given any 2-NPDA, we can construct a Turing Machine that accepts the same language and given a Turing Machine, we can construct a 2-NPDA that accepts the same language.

Turing Machine $\to$ 2-NPDA
We can construct a 2-NPDA that utilizes 2 stacks, u and v, that simulate a Turing Machine. To initialize this 2-NPDA, we have u begin as empty while v contains the input string. This ensures that the 2-NPDA is in a state that is equivalent to the Turing Machine's head being

at the start of the input string. We can let u simulate the portion of the Turing Machine's tape that is to the left of the head while v simulates the portion right of the head. Reading from the tape in a Turing Machine is simulated by popping from v and writing is simulated by pushing onto v. Moving left on the Turing Machine tape corresponds to popping from v and pushing onto u. Moving right corresponds to popping from u and pushing onto v. Also, if popping on either u or v results in that stack being empty, we push a blank symbol onto the other stack which ensures that the head of the simluated Turing Machine is maintained.

2-NPDA $\rightarrow$ Turing Machine
We can construct a Turing Machine from a 2-NPDA by using a 3 tape Turing Machine. In this construction, one tape is the input tape while the other two tapes work as "stacks" that simulate the stacks from the 2-NPDA. Pushing and popping operations from the stacks are simulated by moving right and left on the stack tapes as well as writing or reading symbols from the input tape.

Thus, since we have shown a construction from Turing Machine to 2-NPDA and from 2-NPDA to Turing Machine, we have shown that they are equivalent.

Problem 2

a. A Queue Automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where: Q is the set of all states (finite set), Σ is the input alphabet (finite set), Γ is tape alphabet of queue (finite set), $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \rightarrow P(Q \times (\Gamma \cup \{\epsilon\}))$ is the transition function, q_0 is the start state where $q_0 \in Q$, and F is the set of accept states (finite set) where $F \subseteq Q$.

The Queue Automaton accepts an input string, s, if it can be written in the form $s = s_1 s_2 ... s_m$ where each $s_i \in \Sigma \cup \{\epsilon\}$, and there exists a sequence of states $c_0, c_1, ..., c_m$ and strings $u_0, u_1, ...u_m \in (\Gamma \cup \{\epsilon\})$ such that:

- $c_0 = q_0$ is the initial state.
- $u_0 = \epsilon$ is the initial queue content.
- For each step i, the transition $\delta(c_i, s_{i+1}, a)$ determines the next state c_{i+1} and the next item in the queue u_{i+1}, based on the current state c_i, the next input symbol s_{i+1}, and the head of the queue a. The next input symbol s_{i+1} can be ϵ if there is no more input to read, and similarly, a can be ϵ if the queue is empty.
- c_m is an accept state from the set F.

b. Just as in part (1c), we can show that a Queue Automaton and Turing Machines are equivalent by simulating each other in both directions.

Turing Machine $\rightarrow$ Queue Automaton
The Queue Automaton will always be in the form HR\$L such that H is the head of the Turing Machine, R represents the symbols to the right of the head, \$ is the start of the tape, and L are the symbols to the left of the head. To simulate moving the head to the right, we

dequeue the head and enqueue it which moves it to the back of the queue to the immediate left of the head. To simulate moving left, we can again dequeue the head and enqueue it. Then, we can enqueue some marker symbol $*$. We dequeue every element until we reach this $*$ and finally dequeue this $*$. The head has now shifted left and the order of everything else is preserved. To simulate writing over the head, we enqueue a $*$, dequeue the head, and enqueue the symbol being written. Similar to the process for shifting left, we dequeue and enqueue every symbol until the marker $*$, which we dequeue. If the Turing Machine moves the head to a blank portion of the tape, we need to be able to add some symbol for blank spaces. When we attempt to shift left or right while \$ is the head, we can enqueue a blank symbol $\_$. We can then dequeue and enqueue every symbol until the $\_$ is at the front of te queue. Any number of $\_$ symbols can be added in this manner.

Queue Automaton $\rightarrow$ Turing Machine
We can again use a multitape Turing Machine with two tapes to simluate the process of a Queue Automaton (2 tape TM is equivalent to a single tape TM by lecture). The first tape keeps track of the input while the second tape keeps track of the queue. The Turing Machine starts by writing a \$ on the queue tape to mark the beginning. To simulate enqueuing, the head of the queue tape moves to the first blank space on the right and writes the symbol being read from the input tape. To simulate dequeueing, the head moves to the first symbol to the right of \$ and writes a \$ over it.

Thus, since we have shown a construction from Turing Machine to a Queue Automaton and from a Queue Automaton to Turing Machine, we have shown that they are equivalent.

Problem 3

To prove that a language L is recursively enumerable (RE) if and only if it can be expressed in terms of a decidable language R as $L = \{x : \text{there exists y for which } (x, y) \in R\}$, we can approach it in two parts: (1) If L is expressed as $L = \{x : \text{there exists y for which } (x, y) \in R\}$ where R is a decidable language, then L is RE. (2) If L is RE, then there exists a decidable language R_L such that L can be expressed as $L = \{x : \text{there exists y for which } (x, y) \in R\}$.

(1) For the first part: Assume R is decidable. This means there exists a Turing machine M_R that halts on all inputs and decides R. For language L, we can construct a nondeterministic Turing machine M_L that works as follows: On input x, M_L nondeterministically guesses a string y. M_L then runs M_R on input (x, y). If M_R accepts, then M_L accepts; if M_R rejects, then M_L rejects. Since M_R halts on all inputs, and M_L only accepts if M_R accepts, M_L is an enumerator for L, making L a recursively enumerable language.

(2) For the second part: Assume L is RE. This means there exists a Turing machine M_L that enumerates L. We want to show that there exists a related decidable language R_L. We can construct R_L as follows: Let R_L be the language consisting of pairs (x, y) where x is a string in L and y encodes the computation history of M_L that leads to the acceptance of x. Language R_L is decidable because for any given pair (x, y), we can simulate M_L according to y and check if it leads to the acceptance of x. If it does, accept; otherwise, reject. Since

the computation history y is finite and the simulation is deterministic, the process will halt.

By showing both parts, we conclude that a language L is RE if and only if it can be expressed in terms of some decidable language R as given. This establishes the equivalence between RE languages and languages expressible via the existence of a string in a decidable relation.