1. 1. Since this problem **doesn’t clearly show the Markov Property** (Future is independent of the past, given the present) I would implement an agent that uses **Monte Carlo (MC) methods**. (Take the example of two users folding and the other raising, which you call. The following turn, your action is dependent upon previous actions).   
      1. When considering the potential state spaces to model, I decided to keep a simplistic representation, and not include any previous actions of the agent or other players. Thus, there are **two dimensions**:  
         **The potential card combinations**, which has size 1326 (52C2), although this is dependent on how the dealer hands the cards out. (We assume you receive the two cards first).  
         **The number of credits the agent has**, which we will call |C|. In order to reduce the effects of the curse of dimensionality we could attempt to separate the chip sizes into buckets,  
         I.e. |C| / 50.
      2. I chose to represent the action space in **two dimensions** as well.   
         The first dimension would be the action chosen: R, C, F, P (size 4)  
         The second would be the amount wagered, which would be of size 0 for actions C, F and P, however, for action R would be of size (|C| - current\_wager) if > 0,   
         This is because folding, calling, or passing only require that action to be taken or not, whereas raising can be done with any value greater than the current wager (providing the agent can afford to do so)

For the reward function you could have something like R = points + (Change in credits / 100). This combines making sure that you are winning points (as that appears to be the goal) while also not ending up without any credits. You could also have the 100 value change depending on how close you are to zero credits.

Explore first (e-greedy or softmax or something), then exploit after a given time

* 1. R(T) = Σ ∞ k=0 yk rt+1 but rt+1 = 1 for all t, so   
     R(T) = Σ ∞ k=0 yk = Σ ∞ k=0 (z – 1 / z)k = Σ ∞ k=0 (z – 1 / z)k = 1/[1-(z-1/z)]= z
  2. (Assuming this is a tabular Q-Learning agent) We would **update Q(S42, A8)** using the **maximum expected reward** from taking a successor action in successor state S31 as well as using the **observed immediate reward** (of value 5) for the transition, and incorporating the learning rate and discount factor.  
       
     The expression would be the following:  
       
     Q(S42, A8) = Q(S42, A8) + α (5 + y maxa’ Q(S31, a’) - Q(S42, A8))  
       
     NB: If a Q-Network was used we would still update Q(S42, A8), but would also update neighboring Q(S, A) values because the Q–Network is used to approximate a function and its continuous nature must be preserved.
  3. The main difference between Sarsa and Q-Learning methods is that **Sarsa is an On-Policy** Temporal Difference (TD) control method, whereas **Q-Learning is an Off-policy** TD control method. This means that Sarsa uses the TD error term to update estimates of the Q-Value function using observed transitions whilst it executes a policy, whereas Q-Learning uses the expected value of successor action states to learn the Q-function. Note that Q-Learning **does not specify a specific policy**, the policy is derived from the Q-function itself and as the Q-function is updated, **both the agent’s policy and the target policy** that the agent learns from improves over time. Sarsa on the other hand, **maintains a single policy** (also derived from the Q value) and acts on it.  
     1. C and E, (Not sure about D) others either do not explore enough (hence do not satisfy the GLIE conditions) or the learning rate does not satisfy the Robbins-Monroe conditions.
     2. E will follow the optimal policy once it has found it as it is an on-policy method. C will not as it will continue to explore. Although the answer does depend on how the exploration and exploitation is implemented.
  4. Dynamic Programming (DP) **assumes full knowledge** of the environment, which means it must be **representable as an MDP**. TD Does not assume full knowledge, although it does perform better than MC methods in Markov environments, does not necessarily require the environment to be representable as an MDP.   
       
     Because of what is discussed above, DP **performs a full backup**, with the estimates of the value function eventually converging to the true value (with the degree of accuracy specified by the programmer by setting the threshold.) This means that DP relies entirely on bootstrapping: using its knowledge of the environment.  
       
     TD methods perform **bootstrapping from 1 step samples**, this means it uses both estimates from its knowledge of the environment whilst also performing **sampling** to collect empirical data.
  5. Asynchronous value iteration maintains **one copy of the value function** and **performs updates in-place.**  
       
     We see this relationship also with Q-learning as it maintains the best estimate of the Q-function and updates it in place over the course of the episode.   
       
     Both **methods exploit the Bellman Optimality Equation** as well in order to perform updates.