```
import util, math, random
from collections import defaultdict
from util import ValueIteration

############################################################
# Problem 3a

class BlackjackMDP(util.MDP):
    def __init__(self, cardValues, multiplicity, threshold, peekCost):
        """
        cardValues: list of integers (face values for each card included in the
deck)
        multiplicity: single integer representing the number of cards with each
face value
        threshold: maximum number of points (i.e. sum of card values in hand)
before going bust
        peekCost: how much it costs to peek at the next card
        """
        self.cardValues = cardValues
        self.multiplicity = multiplicity
        self.threshold = threshold
        self.peekCost = peekCost

    # Return the start state.
    # Look closely at this function to see an example of state representation
for our Blackjack game.
    # Each state is a tuple with 3 elements:
    #   -- The first element of the tuple is the sum of the cards in the
player's hand.
    #   -- If the player's last action was to peek, the second element is the
index
    #      (not the face value) of the next card that will be drawn; otherwise,
the
    #      second element is None.
    #   -- The third element is a tuple giving counts for each of the cards
remaining
    #      in the deck, or None if the deck is empty or the game is over (e.g.
when
    #      the user quits or goes bust).
    def startState(self):
        return (0, None, (self.multiplicity,) * len(self.cardValues))

    # Return set of actions possible from |state|.
    # You do not need to modify this function.
    # All logic for dealing with end states should be placed into the
succAndProbReward function below.
    def actions(self, state):
        return ['Take', 'Peek', 'Quit']

    # Given a |state| and |action|, return a list of (newState, prob, reward)
tuples
    # corresponding to the states reachable from |state| when taking |action|.
    # A few reminders:
    # * Indicate a terminal state (after quitting, busting, or running out of
cards)
    #   by setting the deck to None.
    # * If |state| is an end state, you should return an empty list [].
    # * When the probability is 0 for a transition to a particular new state,
    #   don't include that state in the list returned by succAndProbReward.
    def succAndProbReward(self, state, action):
        # BEGIN_YOUR_CODE (our solution is 38 lines of code, but don't worry if
you deviate from this)
        result = []
        totalCardValueInHand, nextCardIndexIfPeeked, deckCardCounts = state
        if deckCardCounts is None:
            return []
        elif sum(deckCardCounts) == 0:
            resultState = (0, None, None)
            resultReward = 0
            if totalCardValueInHand <= self.threshold:
                resultReward = totalCardValueInHand
            result.append((resultState, 1.0, resultReward))

        if action == 'Quit':
            resultState = (0, None, None)
            resultReward = 0
            if totalCardValueInHand <= self.threshold:
                resultReward = totalCardValueInHand
            result.append((resultState, 1.0, resultReward))
        elif action == 'Peek':
            if nextCardIndexIfPeeked is not None:
                return []
            for index, item in enumerate(deckCardCounts):
                if item > 0:
                    transitionProb = float(item) / sum(deckCardCounts)
                    resultState = (totalCardValueInHand, index, deckCardCounts)
                    resultReward = -self.peekCost
                    result.append((resultState, transitionProb, resultReward))
        elif action == 'Take':
            if nextCardIndexIfPeeked is None:
                for index, item in enumerate(deckCardCounts):
                    if item > 0:
                        deckCardCountsList = list(deckCardCounts)
                        transitionProb = float(item) / sum(deckCardCounts)
                        deckCardCountsList[index] -= 1
                        newTotalCardValue = self.cardValues[index] +
totalCardValueInHand
                        resultReward = 0
                        if newTotalCardValue > self.threshold:
                            resultState = (newTotalCardValue, None, None)
                        elif sum(deckCardCountsList) == 0:
                            resultState = (newTotalCardValue, None, None)
                            resultReward = newTotalCardValue
                        else:
                            resultState = (newTotalCardValue, None,
tuple(deckCardCountsList))
                        result.append((resultState, transitionProb,
resultReward))
            else:
                deckCardCountsList = list(deckCardCounts)
                deckCardCountsList[nextCardIndexIfPeeked] -= 1
                newTotalCardValue = self.cardValues[nextCardIndexIfPeeked] +
totalCardValueInHand
                if newTotalCardValue > self.threshold or sum(deckCardCountsList)
== 0:
                    resultState = (newTotalCardValue, None, None)
                else:
                    resultState = (newTotalCardValue, None,
tuple(deckCardCountsList))
                result.append((resultState, 1.0, 0))
        return result
        # END_YOUR_CODE

    def discount(self):
        return 1

############################################################
```

```
# Problem 3b

def peekingMDP():
    """
    Return an instance of BlackjackMDP where peeking is the
    optimal action at least 10% of the time.
    """
    # BEGIN_YOUR_CODE (our solution is 2 lines of code, but don't worry if you
deviate from this)
    return BlackjackMDP(cardValues = [8, 21], multiplicity = 6, threshold = 20,
peekCost = 1)
    # END_YOUR_CODE

############################################################
# Problem 4a: Q learning

# Performs Q-learning.  Read util.RLAlgorithm for more information.
# actions: a function that takes a state and returns a list of actions.
# discount: a number between 0 and 1, which determines the discount factor
# featureExtractor: a function that takes a state and action and returns a list
of (feature name, feature value) pairs.
# explorationProb: the epsilon value indicating how frequently the policy
# returns a random action
class QLearningAlgorithm(util.RLAlgorithm):
    def __init__(self, actions, discount, featureExtractor,
explorationProb=0.2):
        self.actions = actions
        self.discount = discount
        self.featureExtractor = featureExtractor
        self.explorationProb = explorationProb
        self.weights = defaultdict(float)
        self.numIters = 0

    # Return the Q function associated with the weights and features
    def getQ(self, state, action):
        score = 0
        for f, v in self.featureExtractor(state, action):
            score += self.weights[f] * v
        return score

    # This algorithm will produce an action given a state.
    # Here we use the epsilon-greedy algorithm: with probability
    # |explorationProb|, take a random action.
    def getAction(self, state):
        self.numIters += 1
        if random.random() < self.explorationProb:
            return random.choice(self.actions(state))
        else:
            return max((self.getQ(state, action), action) for action in
self.actions(state))[1]

    # Call this function to get the step size to update the weights.
    def getStepSize(self):
        return 1.0 / math.sqrt(self.numIters)

    # We will call this function with (s, a, r, s'), which you should use to
update |weights|.
    # Note that if s is a terminal state, then s' will be None.  Remember to
check for this.
    # You should update the weights using self.getStepSize(); use
    # self.getQ() to compute the current estimate of the parameters.
    def incorporateFeedback(self, state, action, reward, newState):
        # BEGIN_YOUR_CODE (our solution is 9 lines of code, but don't worry if
you deviate from this)
```

```
        if newState is None:
            return
        vStar = max([self.getQ(newState, newAction) for newAction in 
self.actions(newState)])
        residual =- self.getStepSize() * (self.getQ(state, action) - (reward + 
self.discount * vStar))
        for item in self.featureExtractor(state, action):
            key,value = item
            self.weights[key] += residual * value
        # END_YOUR_CODE

# Return a single-element list containing a binary (indicator) feature
# for the existence of the (state, action) pair.  Provides no generalization.
def identityFeatureExtractor(state, action):
    featureKey = (state, action)
    featureValue = 1
    return [(featureKey, featureValue)]

############################################################
# Problem 4b: convergence of Q-learning
# Small test case
smallMDP = BlackjackMDP(cardValues=[1, 5], multiplicity=2, threshold=10, 
peekCost=1)

# Large test case
largeMDP = BlackjackMDP(cardValues=[1, 3, 5, 8, 10], multiplicity=3, 
threshold=40, peekCost=1)

def simulate_QL_over_MDP(mdp, featureExtractor):
    # NOTE: adding more code to this function is totally optional, but it will 
probably be useful
    # to you as you work to answer question 4b (a written question on this 
assignment).  We suggest
    # that you add a few lines of code here to run value iteration, simulate Q-
learning on the MDP,
    # and then print some stats comparing the policies learned by these two 
approaches.
    # BEGIN_YOUR_CODE
    mdp.computeStates()
    qLearning = QLearningAlgorithm(mdp.actions, mdp.discount(), 
featureExtractor, 0.2)
    util.simulate(mdp, qLearning, numTrials = 30000, maxIterations = 1000, 
verbose = False, sort = False)
    qLearning.explorationProb = 0.0
    valIteration = ValueIteration()
    valIteration.solve(mdp)
    diffAction = 0
    for state in mdp.states:
       if qLearning.getAction(state) != valIteration.pi[state]:
           diffAction += 1
    print(diffAction, len(mdp.states))
    # END_YOUR_CODE


############################################################
# Problem 4c: features for Q-learning.

# You should return a list of (feature key, feature value) pairs.
# (See identityFeatureExtractor() above for a simple example.)
# Include the following features in the list you return:
# -- Indicator for the action and the current total (1 feature).
# -- Indicator for the action and the presence/absence of each face value in the 
deck.
#       Example: if the deck is (3, 4, 0, 2), then your indicator on the
```

```
presence of each card is (1, 1, 0, 1)
#       Note: only add this feature if the deck is not None.
# -- Indicators for the action and the number of cards remaining with each face
value (len(counts) features).
#       Note: only add these features if the deck is not None.
def blackjackFeatureExtractor(state, action):
    total, _, counts = state

    # BEGIN_YOUR_CODE (our solution is 7 lines of code, but don't worry if you
deviate from this)
    keyValuePairs = []
    if counts is None:
        return keyValuePairs
    featureKey = (action, total)
    keyValuePairs.append((featureKey, 1))
    cList = list(counts)
    for index, item in enumerate(counts):
        featureKey = (action, index, item)
        keyValuePairs.append((featureKey, 1))
        if item > 0:
            cList[index] = 1
    featureKey = (action, tuple(cList))
    keyValuePairs.append((featureKey, 1))
    return keyValuePairs
    # END_YOUR_CODE

############################################################
# Problem 4d: What happens when the MDP changes underneath you?!

# Original mdp
originalMDP = BlackjackMDP(cardValues=[1, 5], multiplicity=2, threshold=10,
peekCost=1)

# New threshold
newThresholdMDP = BlackjackMDP(cardValues=[1, 5], multiplicity=2, threshold=15,
peekCost=1)

def compare_changed_MDP(original_mdp, modified_mdp, featureExtractor):
    # NOTE: as in 4b above, adding more code to this function is completely
optional, but we've added
    # this partial function here to help you figure out the answer to 4d (a
written question).
    # Consider adding some code here to simulate two different policies over the
modified MDP
    # and compare the rewards generated by each.
    # BEGIN_YOUR_CODE
    original_mdp.computeStates()
    valIteration = ValueIteration()
    valIteration.solve(originalMDP)
    # Fixed RL algo
    fixedRL = util.FixedRLAlgorithm(valIteration.pi.copy())
    reward = util.simulate(original_mdp, fixedRL, numTrials = 30000,
maxIterations = 1000, verbose = False, sort = False)
    print ('\nAverage utility for original_mdp is: ',
float(sum(reward))/float(len(reward)))
    fixedRL.explorationProb = 0.0

    reward = util.simulate(modified_mdp, fixedRL, numTrials = 30000,
maxIterations = 1000, verbose = False, sort = False)
    print ('\nAverage utility for modified_mdp is: ',
float(sum(reward))/float(len(reward)))

    fixedRL.explorationProb = 0.0
    modified_mdp.computeStates()
    # QLearning algo
    qLearning = QLearningAlgorithm(modified_mdp.actions,
modified_mdp.discount(), featureExtractor, 0.2)
    reward = util.simulate(modified_mdp, qLearning, numTrials = 30000,
maxIterations = 1000, verbose = False, sort = False)
    print ('\nAverage utility for algo is: ',
float(sum(reward))/float(len(reward)))
    # END_YOUR_CODE
```