In [ ]:
```
import os
```

In [ ]:
```
# MDP Environment
import random

class MDP(object):

  def __init__(self,Questions,Prob_Rewards):
    self.Questions = Questions
    self.Prob_Rewards = Prob_Rewards

  def InitState(self):
    return 1
  def EndState(self, state):
    return state == self.Questions
  def Actions(self,state):
    if self.EndState(state):
      return []
    else:
      return ['Stay','Quit']
  def SuccProbReward(self, state, action):
    if action == 'Stay':
      if state > self.Questions:
        return [(state,1.,0.)]
      else:
        Probability = self.Prob_Rewards[state][0]
        Reward = self.Prob_Rewards[state][1]
        return [(state+1,Probability,Reward),(self.Questions+1, 1.-Probability, 0.)
    elif action == 'Quit':
      if state > self.Questions:
        return[(state,1.,0.)]
      else:
        Probability = 1.0
        Reward = self.Prob_Rewards[state][1]
        return [(self.Questions+1, Probability, Reward)]
  def discount(self):
      return 0.9
  def states(self):
    return range(1,self.Questions+2)


# Value Iteration for the above Defined MDP environment

def valueIteration(mdp, epsilon=0.001):
    # initialize values of all states to zero
    V = {s: 0.0 for s in mdp.states()}
    # repeat until convergence
    while True:
        # set change to zero
        delta = 0
        # for each state, update its value using Bellman update
        for s in mdp.states():
            if mdp.EndState(s):
                continue
```

```
            # compute the maximum value over all possible Actions
            MaxValue = -float('inf')
            for a in mdp.Actions(s):
                val = 0
                for NextState, Probability, Reward in mdp.SuccProbReward(s, a):
                    val += Probability * (Reward + mdp.discount() * V[NextState])
                MaxValue = max(MaxValue, val)
            # update value of state
            delta = max(delta, abs(MaxValue - V[s]))
            V[s] = MaxValue
        # check for convergence
        if delta < epsilon:
            break
    # compute the optimal policy using the computed values
    pi = {}
    for s in mdp.states():
        if mdp.EndState(s):
            pi[s] = None
        else:
            MaxValue = -float('inf')
            best_action = None
            for a in mdp.Actions(s):
                val = 0
                for NextState, Probability, Reward in mdp.SuccProbReward(s, a):
                    val += Probability * (Reward + mdp.discount() * V[NextState])
                if val > MaxValue:
                    MaxValue = val
                    best_action = a
            pi[s] = best_action
    # return the computed values and policy
    return V, pi

Questions = 10
Prob_Rewards = {1: (0.99, 100),
                2: (0.9, 500),
                3: (0.8, 1000),
                4: (0.7, 5000),
                5: (0.6, 10000),
                6: (0.5, 50000),
                7: (0.4, 100000),
                8: (0.3, 500000),
                9: (0.2, 1000000),
                10: (0.1, 5000000)}
kbc_MDP = MDP(Questions,Prob_Rewards)

V, P = valueIteration(kbc_MDP)
# kbc_MDP.MonteCarlo(2)
print(V)
print(P)
```

```
{1: 26780.771381760005, 2: 29945.87136000001, 3: 36414.656, 4: 49464.799999999996,
5: 72960.0, 6: 124000.0, 7: 220000.0, 8: 500000.0, 9: 1000000.0, 10: 0.0, 11: 0.0}
{1: 'Stay', 2: 'Stay', 3: 'Stay', 4: 'Stay', 5: 'Stay', 6: 'Stay', 7: 'Stay', 8:
'Quit', 9: 'Quit', 10: None, 11: 'Stay'}
```

In [ ]:
```
# Policy Iteration for the above Defined MDP environment

def policyIteration(mdp, epsilon=0.001):
    # initialize values of all states to zero
    V = {s: 0.0 for s in mdp.states()}
    # initialize policy arbitrarily
    pi = {s: mdp.Actions(s)[0] for s in mdp.states() if not mdp.EndState(s)}
    while True:
        # policy evaluation step
        while True:
            delta = 0
            for s in mdp.states():
                if mdp.EndState(s):
                    continue
                # compute the value of the current policy for this state
                val = 0
                for NextState, Probability, Reward in mdp.SuccProbReward(s, pi[s]):
                    val += Probability * (Reward + mdp.discount() * V[NextState])
                # update value of state
                delta = max(delta, abs(val - V[s]))
                V[s] = val
            if delta < epsilon:
                break
        # policy improvement step
        policy_stable = True
        for s in mdp.states():
            if mdp.EndState(s):
                continue
            old_action = pi[s]
            # find the best action using the updated values
            MaxValue = -float('inf')
            best_action = None
            for a in mdp.Actions(s):
                val = 0
                for NextState, Probability, Reward in mdp.SuccProbReward(s, a):
                    val += Probability * (Reward + mdp.discount() * V[NextState])
                if val > MaxValue:
                    MaxValue = val
                    best_action = a
            # update policy
            pi[s] = best_action
            if old_action != best_action:
                policy_stable = False
        # check for convergence
        if policy_stable:
            break
    # return the computed values and policy
    return V, pi

Questions = 10
Prob_Rewards = {1: (0.99, 100),
                2: (0.9, 500),
                3: (0.8, 1000),
                4: (0.7, 5000),
                5: (0.6, 10000),
```

```
                6: (0.5, 50000),
                7: (0.4, 100000),
                8: (0.3, 500000),
                9: (0.2, 1000000),
                10: (0.1, 5000000)}
kbc_MDP = MDP(Questions,Prob_Rewards)

V ,P = policyIteration(kbc_MDP)

print(P)
print(V)
```

```
{1: 'Stay', 2: 'Stay', 3: 'Stay', 4: 'Stay', 5: 'Stay', 6: 'Stay', 7: 'Stay', 8: 'Quit', 9: 'Quit', 11: 'Stay'}
{1: 26780.771381760005, 2: 29945.87136000001, 3: 36414.656, 4: 49464.799999999996, 5: 72960.0, 6: 124000.0, 7: 220000.0, 8: 500000.0, 9: 1000000.0, 10: 0.0, 11: 0.0}
```

In [ ]:

```
# Sate
def generateSARS(kbc_MDP, pi, num_sequences=1, max_steps=100):
    sequences = []
    for i in range(num_sequences):
        state = kbc_MDP.InitState()
        sequence = []
        for t in range(max_steps):
            if kbc_MDP.EndState(state):
                break
            action = pi[state]
            next_states, probabilities, rewards = zip(*kbc_MDP.SuccProbReward(state
            next_state = random.choices(next_states, probabilities)[0]
            reward = rewards[next_states.index(next_state)]
            sequence.append((state, action, reward, next_state))
            state = next_state
        sequences.append(sequence)
    return sequences
sequences = generateSARS(kbc_MDP, P, num_sequences=10, max_steps=10)
for sequence in sequences:
    print(sequence)
    print("\n")
```

```
[(1, 'Stay', 100, 2), (2, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.
0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11,
'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11)]


[(1, 'Stay', 100, 2), (2, 'Stay', 500, 3), (3, 'Stay', 1000, 4), (4, 'Stay', 5000,
5), (5, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Sta
y', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11)]


[(1, 'Stay', 100, 2), (2, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.
0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11,
'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11)]


[(1, 'Stay', 100, 2), (2, 'Stay', 500, 3), (3, 'Stay', 0.0, 11), (11, 'Stay', 0.0,
11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'St
ay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11)]


[(1, 'Stay', 100, 2), (2, 'Stay', 500, 3), (3, 'Stay', 0.0, 11), (11, 'Stay', 0.0,
11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'St
ay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11)]


[(1, 'Stay', 100, 2), (2, 'Stay', 500, 3), (3, 'Stay', 1000, 4), (4, 'Stay', 5000,
5), (5, 'Stay', 10000, 6), (6, 'Stay', 50000, 7), (7, 'Stay', 100000, 8), (8, 'Qui
t', 500000, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11)]


[(1, 'Stay', 100, 2), (2, 'Stay', 500, 3), (3, 'Stay', 0.0, 11), (11, 'Stay', 0.0,
11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'St
ay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11)]


[(1, 'Stay', 100, 2), (2, 'Stay', 500, 3), (3, 'Stay', 1000, 4), (4, 'Stay', 0.0,
11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'St
ay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11)]


[(1, 'Stay', 100, 2), (2, 'Stay', 500, 3), (3, 'Stay', 1000, 4), (4, 'Stay', 5000,
5), (5, 'Stay', 10000, 6), (6, 'Stay', 50000, 7), (7, 'Stay', 100000, 8), (8, 'Qui
t', 500000, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11)]


[(1, 'Stay', 100, 2), (2, 'Stay', 500, 3), (3, 'Stay', 0.0, 11), (11, 'Stay', 0.0,
11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'St
ay', 0.0, 11), (11, 'Stay', 0.0, 11), (11, 'Stay', 0.0, 11)]
```

In [ ]:

```
# Monto Carlo Environment for the above problem
import random

class MontoCarlo(object):
```

```python
    def __init__(self, Questions, Prob_Rewards):
        self.Questions = Questions
        self.Prob_Rewards = Prob_Rewards

    def play(self, policy):
        # initialize variables
        state = self.InitState()
        done = False
        rewards = []
        # play until the game is over
        while not done:
            # choose an action according to the given policy
            action = policy(state)
            # take the action and observe the next state and reward
            next_state, reward, done = self.SuccProbReward(state, action)[0]
            # store the reward
            rewards.append(reward)
            # update the current state
            state = next_state
        # compute the cumulative rewards for each time step
        cumulative_rewards = [sum(rewards[i:]) for i in range(len(rewards))]
        # create a list of (state, action, cumulative reward) tuples
        episodes = [(self.InitState(), None, 0)]
        for i in range(len(rewards)):
            episodes.append((i+1, policy(i+1), cumulative_rewards[i]))
        return episodes

    def InitState(self):
        return 1

    def EndState(self, state):
        return state == self.Questions + 1

    def Actions(self, state):
        if self.EndState(state):
            return []
        else:
            return ['Stay', 'Quit']

    def SuccProbReward(self, state, action):
        if action == 'Stay':
            if state > self.Questions:
                return [(state, 0, True)]
            else:
                prob, reward = self.Prob_Rewards[state]
                if random.random() <= prob:
                    return [(state+1, reward, False)]
                else:
                    return [(self.Questions+1, 0, True)]
        elif action == 'Quit':
            if state > self.Questions:
                return [(state, 0, True)]
            else:
                reward = self.Prob_Rewards[state][1]
                return [(self.Questions+1, reward, True)]
```

```
    def discount(self):
        return 0.9

    def states(self):
        return range(1, self.Questions+2)


# define a random policy
def random_policy(state):
    actions = ['Stay', 'Quit']
    return random.choice(actions)

# create a MontoCarlo environment
env = MontoCarlo(10, [(0.99, 100),(0.9, 500),(0.8, 1000),(0.7, 5000), (0.6, 10000),

# run a random policy for one episode
episode = env.play(random_policy)

# print the episode
print(episode)
```

```
[(1, None, 0), (1, 'Stay', 6500), (2, 'Quit', 6000), (3, 'Quit', 5000), (4, 'Qui
t', 0)]
```

In [ ]:

```
import numpy as np

# Define the KBC game
Questions = 10
Prob_Rewards = [(0.99, 100),(0.9, 500),(0.8, 1000),(0.7, 5000), (0.6, 10000), (0.5,

# # Create the transition matrix
num_states = Questions + 1
num_actions = 2

R = np.zeros(num_states).astype(int)
for s in range(num_states):
    if s == num_states - 1:
        R[s] = 0 # end of game
    else:
        R[s] = Prob_Rewards[s][1]

# # Format the output of the transition matrix

P1 = np.array([[0.99, 0.01, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0.9, 0.1, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0.8, 0.2, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0.7, 0.3, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0.6, 0.4, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0.5, 0.5, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0.4, 0.6, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0.3, 0.7, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0.2, 0.8, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0.1, 0.9],
                 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]])

print("Transition matrix : \n")
```

```
for i in range(num_states):
    for j in range(num_states):
        print(P1[i][j], end = "    ")
    print("\n")
print("Rewards Vector : \n")
print(R)


# Set up the discount factor
gamma = 0.9

# Solve the system of linear equations
A = np.eye(num_states) - gamma * P1
b = R
V = np.linalg.inv(A).dot(b) # value function

# Print the value function vector
print("\n Value function : \n")
print(V)
```

```
Transition matrix :

0.99   0.01   0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0

0.0    0.9    0.1    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0

0.0    0.0    0.8    0.2    0.0    0.0    0.0    0.0    0.0    0.0    0.0

0.0    0.0    0.0    0.7    0.3    0.0    0.0    0.0    0.0    0.0    0.0

0.0    0.0    0.0    0.0    0.6    0.4    0.0    0.0    0.0    0.0    0.0

0.0    0.0    0.0    0.0    0.0    0.5    0.5    0.0    0.0    0.0    0.0

0.0    0.0    0.0    0.0    0.0    0.0    0.4    0.6    0.0    0.0    0.0

0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.3    0.7    0.0    0.0

0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.2    0.8    0.0

0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.1    0.9

0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    1.0

Rewards Vector :

[    100     500    1000    5000   10000   50000  100000  500000 1000000
 5000000       0]

 Value function :

[  63648.10943705  759738.21429313 1598336.23017438 2480745.24693792
 3381021.26432234 4292416.05996743 5135175.18440464 5900948.36670179
 6043956.04395604 5494505.49450549       0.        ]
```

In [ ]: