EX.NO: 1 **ELUCIDATING POLICY ITERATION IN JACK'S CAR**
DATE: **RENTAL PROBLEM**

AIM

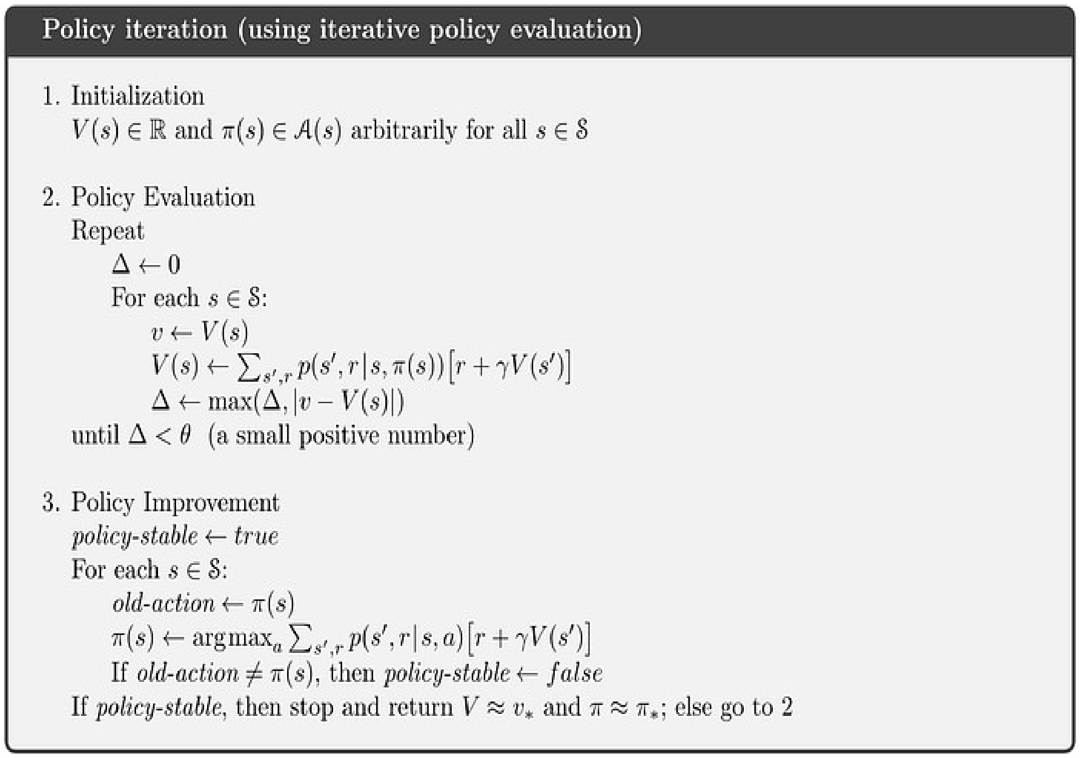
To develop a Python program to elucidate value iteration and policy iteration in Jacks' Car Rental problem.

PROBLEM SATEMENT

Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrives at each location to rent cars. If Jack has a car available, he rents it out and is credited $10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of $2 per car moved.

POLICY ITERATION ALGORITHM

A policy is a mapping from states to actions, i.e., given a state, how many cars should Jack move overnight. Now, suppose Jack has some policy π, then given this π, the value of a state (say s) is the expected reward that Jack would get when he starts from s and follows π after that



```
Policy iteration (using iterative policy evaluation)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
       $\Delta \leftarrow 0$
       For each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ true
   For each $s \in \mathcal{S}$:
       old-action $\leftarrow \pi(s)$
       $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
       If old-action $\neq \pi(s)$, then policy-stable $\leftarrow$ false
   If policy-stable, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

The policy iteration algorithm, as shown in the above image, consists of three components. The first component is the initialization. Initialize the value and policy matrices to zero. Given a policy, define a value for each state, and since state is a pair of two numbers where each number takes a value between 0 and 20, hence represent value by a matrix of shape

(21 x 21). The policy takes a state and outputs an action; hence, it can also be represented by a matrix of the same shape.

The second component is policy evaluation. By policy evaluation, we mean that following this policy, what should be the value of any state. As mentioned above, given a policy π, the value of a state (say s) is the expected reward that Jack would get when he starts from s and follows π after that. This Bellman equation forms the basis of the value update shown in the policy evaluation component. After many such updates, V(s) converges to a number which almost satisfies (with at most some θ error) the Bellman equation and hence represents the value of state s.

$$V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))[r+\gamma V(s')]$$

The third component is policy improvement. Given a state (say s), assign π(s) to be equal to that action which maximizes the expected reward. The policy becomes stable when none of the action maximization step in any state causes a change in the policy.

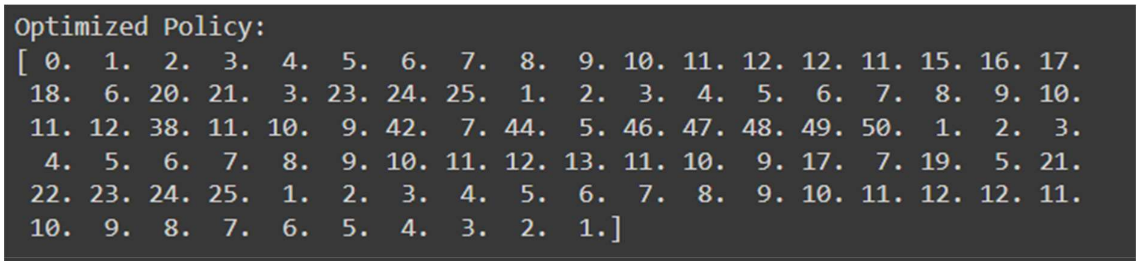
Algorithm:

1. Initialize Policy by starting with a random policy (e.g., move 0 cars between locations).
2. For each state, calculate expected rewards by following the current policy and update the value of each state based on expected future returns.
3. For each state, take different actions and update the policy with the action that maximizes expected profits.
4. Repeat policy evaluation and improvement until the policy remains unchanged.
5. Return the final policy that maximizes profits for each state.

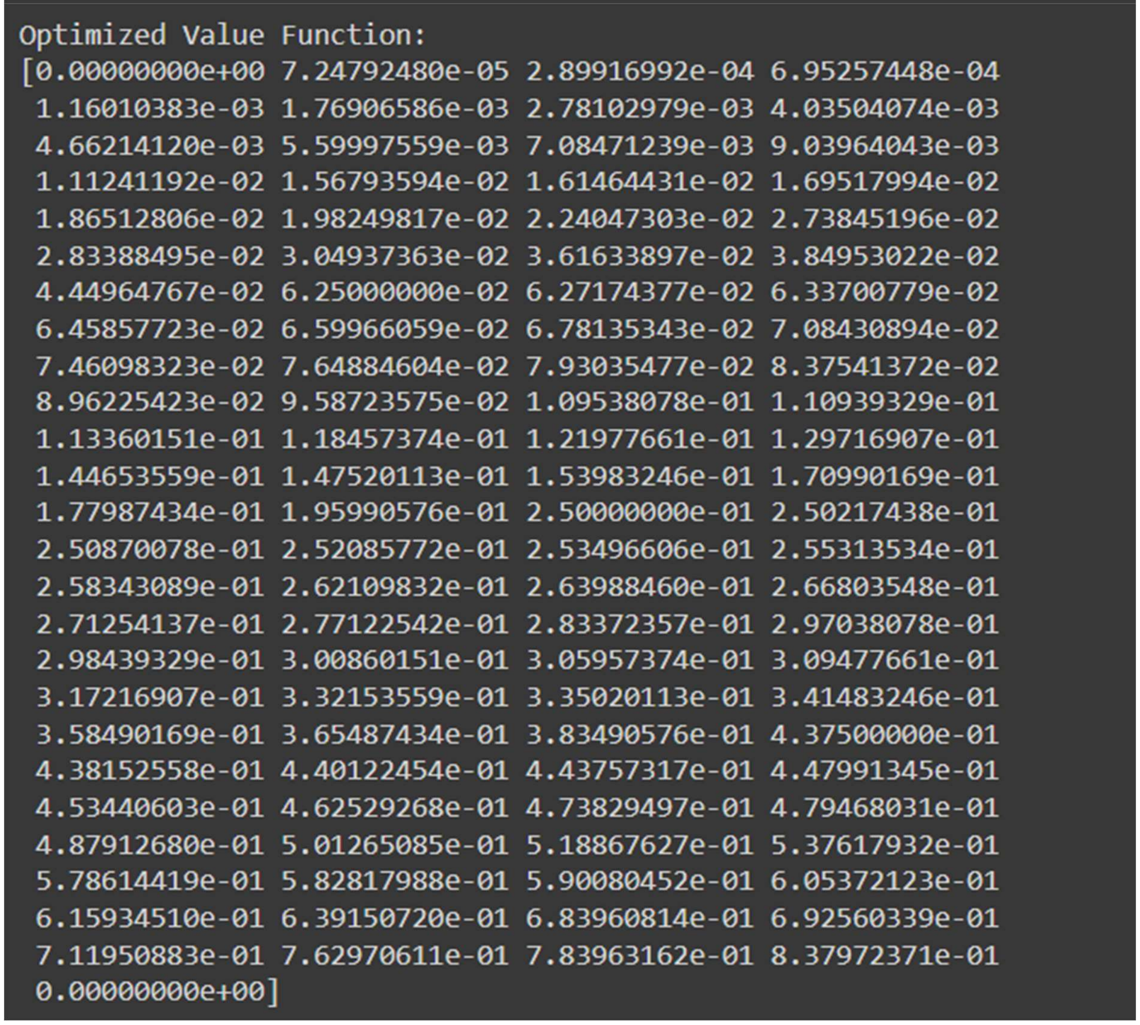
PROGRAM

```
import numpy as np

import sys

import matplotlib.pyplot as plt

if "../" not in sys.path:

  sys.path.append("../")

def value_iteration_for_gamblers(p_h, theta=0.0001, discount_factor=1.0):

    rewards = np.zeros(101)

    rewards[100] = 1

    V = np.zeros(101)
    def one_step_lookahead(s, V, rewards):
        A = np.zeros(101)
        stakes = range(1, min(s, 100-s)+1)
        for a in stakes:
            A[a] = p_h * (rewards[s+a] + V[s+a]*discount_factor) + (1-p_h) * (rewards[s-a] + V[s-
a]*discount_factor)
        return A
    while True:
        delta = 0
        for s in range(1, 100):
            A = one_step_lookahead(s, V, rewards)
            best_action_value = np.max(A)
            delta = max(delta, np.abs(best_action_value - V[s]))
            V[s] = best_action_value
        if delta < theta:
            break
    policy = np.zeros(100)
    for s in range(1, 100):
        A = one_step_lookahead(s, V, rewards)
        best_action = np.argmax(A)
        policy[s] = best_action
    return policy, V
policy, v = value_iteration_for_gamblers(0.25)
print("Optimized Policy:")
print(policy)
print("")
print("Optimized Value Function:")
print(v)
```

OUTPUT



```
Optimized Policy:
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 12. 11. 15. 16. 17.
 18.  6. 20. 21.  3. 23. 24. 25.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.
 11. 12. 38. 11. 10.  9. 42.  7. 44.  5. 46. 47. 48. 49. 50.  1.  2.  3.
  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 11. 10.  9. 17.  7. 19.  5. 21.
 22. 23. 24. 25.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 12. 11.
 10.  9.  8.  7.  6.  5.  4.  3.  2.  1.]
```



```
Optimized Value Function:
[0.00000000e+00 7.24792480e-05 2.89916992e-04 6.95257448e-04
 1.16010383e-03 1.76906586e-03 2.78102979e-03 4.03504074e-03
 4.66214120e-03 5.59997559e-03 7.08471239e-03 9.03964043e-03
 1.11241192e-02 1.56793594e-02 1.61464431e-02 1.69517994e-02
 1.86512806e-02 1.98249817e-02 2.24047303e-02 2.73845196e-02
 2.83388495e-02 3.04937363e-02 3.61633897e-02 3.84953022e-02
 4.44964767e-02 6.25000000e-02 6.27174377e-02 6.33700779e-02
 6.45857723e-02 6.59966059e-02 6.78135343e-02 7.08430894e-02
 7.46098323e-02 7.64884604e-02 7.93035477e-02 8.37541372e-02
 8.96225423e-02 9.58723575e-02 1.09538078e-01 1.10939329e-01
 1.13360151e-01 1.18457374e-01 1.21977661e-01 1.29716907e-01
 1.44653559e-01 1.47520113e-01 1.53983246e-01 1.70990169e-01
 1.77987434e-01 1.95990576e-01 2.50000000e-01 2.50217438e-01
 2.50870078e-01 2.52085772e-01 2.53496606e-01 2.55313534e-01
 2.58343089e-01 2.62109832e-01 2.63988460e-01 2.66803548e-01
 2.71254137e-01 2.77122542e-01 2.83372357e-01 2.97038078e-01
 2.98439329e-01 3.00860151e-01 3.05957374e-01 3.09477661e-01
 3.17216907e-01 3.32153559e-01 3.35020113e-01 3.41483246e-01
 3.58490169e-01 3.65487434e-01 3.83490576e-01 4.37500000e-01
 4.38152558e-01 4.40122454e-01 4.43757317e-01 4.47991345e-01
 4.53440603e-01 4.62529268e-01 4.73829497e-01 4.79468031e-01
 4.87912680e-01 5.01265085e-01 5.18867627e-01 5.37617932e-01
 5.78614419e-01 5.82817988e-01 5.90080452e-01 6.05372123e-01
 6.15934510e-01 6.39150720e-01 6.83960814e-01 6.92560339e-01
 7.11950883e-01 7.62970611e-01 7.83963162e-01 8.37972371e-01
 0.00000000e+00]
```

RESULT

The Python program to elucidate value iteration and policy iteration in Jack's Car Rental problem was developed and executed successfully.

EX.NO: 2
DATE:

ELUCIDATING VALUE ITERATION FOR THE GAMBLER'S PROBLEM

AIM

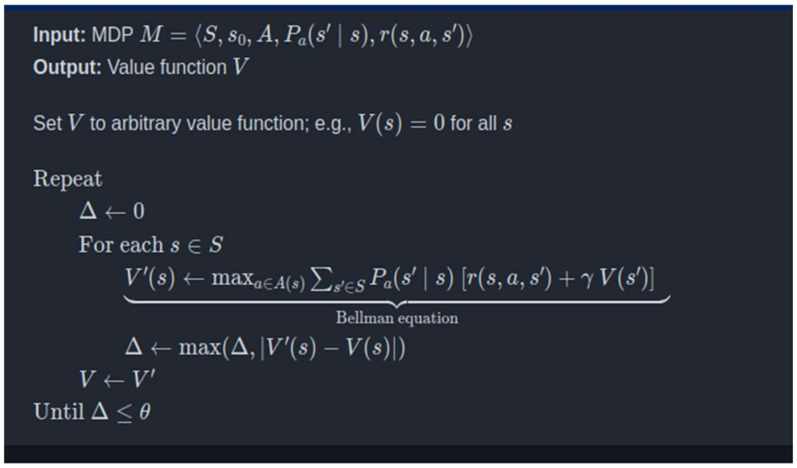
To develop a Python program to elucidate value iteration in Gambler's problem.

PROBLEM SATEMENT

A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake. The game ends when the gambler wins by reaching his goal of $100, or loses by running out of money.
On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars. This problem can be formulated as an undiscounted, episodic, finite MDP.
The state is the gambler's capital, s ∈ {1, 2, . . . , 99}. The actions are stakes, a ∈ {0, 1, . . . , min(s, 100 − s)}. The reward is zero on all transitions except those on which the gambler reaches his goal, when it is +1.
The state-value function then gives the probability of winning from each state. A policy is a mapping from levels of capital to stakes. The optimal policy maximizes the probability of reaching the goal. Let p_h denote the probability of the coin coming up heads. If p_h is known, then the entire problem is known and it can be solved, for instance, by value iteration.

VALUE ITERATIONALGORITHM

Value Iteration is a method for finding the optimal value function **V*** by solving the Bellman equations iteratively. It uses the concept of dynamic programming to maintain a value function **V** that approximates the optimal value function V*, iteratively improving V until it converges to **V*** (or close to it).



```
Input: MDP $M = \langle S, s_0, A, P_a(s' \mid s), r(s,a,s') \rangle$
Output: Value function $V$

Set $V$ to arbitrary value function; e.g., $V(s) = 0$ for all $s$

Repeat
    $\Delta \leftarrow 0$
    For each $s \in S$
        $\underbrace{V'(s) \leftarrow \max_{a \in A(s)} \sum_{s' \in S} P_a(s' \mid s)\, [r(s,a,s') + \gamma\, V(s')]}_{\text{Bellman equation}}$
        $\Delta \leftarrow \max(\Delta, |V'(s) - V(s)|)$
    $V \leftarrow V'$
Until $\Delta \leq \theta$
```

Value Iteration

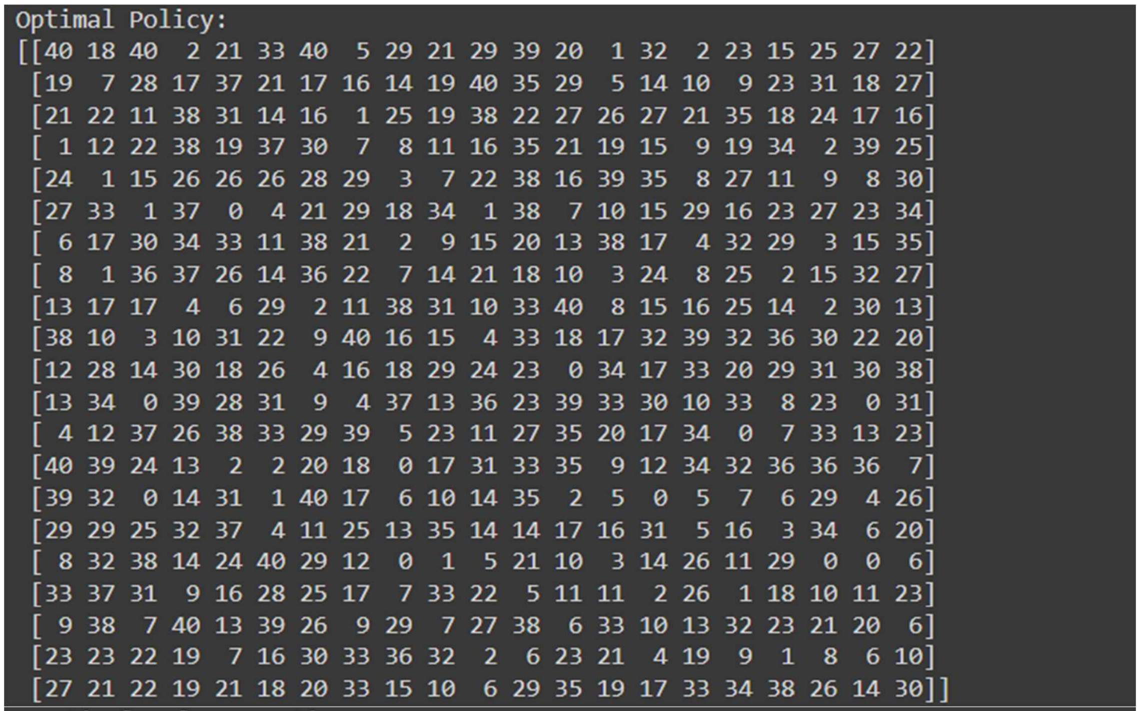
ALGORITHM

1. Initialize by assigning a value of 0 to all states except the goal state, which has a value of 1.
2. For each possible amount of money the gambler has (state), calculate the expected reward for each action (bet amount).
3. For each state, update its value by selecting the action (bet amount) that maximizes the expected reward, considering the probabilities of winning or losing the bet.
4. Repeat the process for all states until the values converge (i.e., no significant changes in value between iterations).
5. Once the values stabilize, derive the optimal betting strategy by choosing the action that leads to the highest value for each state.

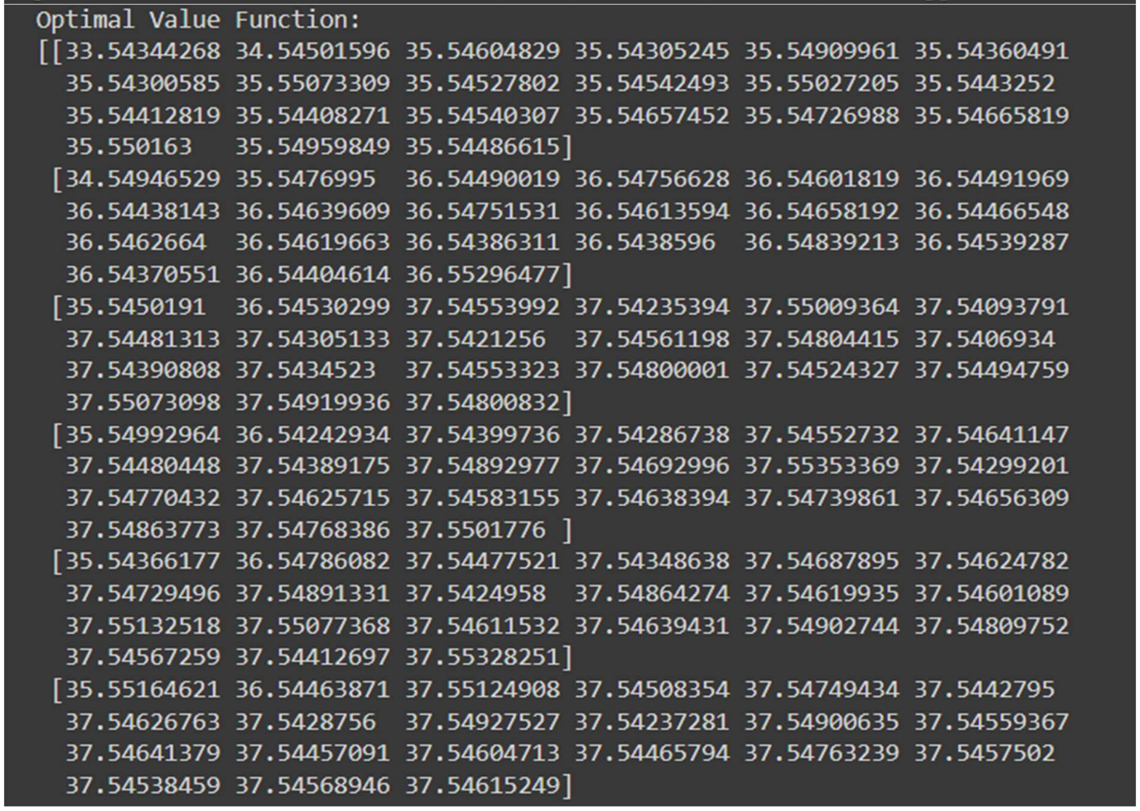
PROGRAM

```
import numpy as np

class JackCarRentalPolicyIteration:

    def __init__(self, max_cars, transition_probs, rewards, discount_factor=0.9):

        self.max_cars = max_cars

        self.transition_probs = transition_probs

        self.rewards = rewards

        self.discount_factor = discount_factor

        self.num_actions = 2 * max_cars + 1

        self.policy = np.zeros((max_cars + 1, max_cars + 1), dtype=int)

        self.value_function = np.zeros((max_cars + 1, max_cars + 1))


    def policy_evaluation(self):

        delta = np.inf

        while delta > 1e-6:

            delta = 0

            new_value_function = np.zeros_like(self.value_function)

            for s1 in range(self.max_cars + 1):

                for s2 in range(self.max_cars + 1):

                    action = self.policy[s1, s2]

                    move_cars = action - self.max_cars

                    new_s1 = np.clip(s1 - move_cars, 0, self.max_cars)
                new_s2 = np.clip(s2 + move_cars, 0, self.max_cars)
                transition_prob = self.transition_probs[action][s1, s2]
                reward = self.rewards[s1, s2]
                        new_value_function[s1, s2] = np.sum(transition_prob * (reward +
self.discount_factor * self.value_function))
                 delta = max(delta, np.abs(new_value_function[s1, s2] - self.value_function[s1,
s2]))
            self.value_function = new_value_function


    def policy_improvement(self):
        policy_stable = True
        new_policy = np.zeros_like(self.policy)
        for s1 in range(self.max_cars + 1):
            for s2 in range(self.max_cars + 1):
                action_values = np.zeros(self.num_actions)
                for action in range(self.num_actions):
                    move_cars = action - self.max_cars
                    new_s1 = np.clip(s1 - move_cars, 0, self.max_cars)
                    new_s2 = np.clip(s2 + move_cars, 0, self.max_cars)
                    transition_prob = self.transition_probs[action][s1, s2]
                    reward = self.rewards[s1, s2]
                    action_values[action] = np.sum(transition_prob * (reward + self.discount_factor
* self.value_function))
                best_action = np.argmax(action_values)
                new_policy[s1, s2] = best_action
                if self.policy[s1, s2] != best_action:
                    policy_stable = False
        self.policy = new_policy
        return policy_stable


    def run(self):
        while True:
            self.policy_evaluation()
            if self.policy_improvement():
                break
        return self.policy, self.value_function
if __name__ == "__main__":
    max_cars = 20
    transition_probs = np.zeros((2 * max_cars + 1, max_cars + 1, max_cars + 1, max_cars + 1,
max_cars + 1))
    for action in range(2 * max_cars + 1):
        move_cars = action - max_cars
        for s1 in range(max_cars + 1):
            for s2 in range(max_cars + 1):
                probs = np.random.uniform(0.1, 0.2, (max_cars + 1, max_cars + 1))
                probs /= probs.sum()
                transition_probs[action][s1, s2] = probs
      rewards = np.minimum(np.arange(max_cars + 1)[:, None], 2) +
np.minimum(np.arange(max_cars + 1), 2)
    pi = JackCarRentalPolicyIteration(max_cars, transition_probs, rewards)
    optimal_policy, optimal_value_function = pi.run()
    print("Optimal Policy:")
    print(optimal_policy)
    print("Optimal Value Function:")
    print(optimal_value_function)
```

OUTPUT



```
Optimal Policy:
[[40 18 40  2 21 33 40  5 29 21 29 39 20  1 32  2 23 15 25 27 22]
 [19  7 28 17 37 21 17 16 14 19 40 35 29  5 14 10  9 23 31 18 27]
 [21 22 11 38 31 14 16  1 25 19 38 22 27 26 27 21 35 18 24 17 16]
 [ 1 12 22 38 19 37 30  7  8 11 16 35 21 19 15  9 19 34  2 39 25]
 [24  1 15 26 26 26 28 29  3  7 22 38 16 39 35  8 27 11  9  8 30]
 [27 33  1 37  0  4 21 29 18 34  1 38  7 10 15 29 16 23 27 23 34]
 [ 6 17 30 34 33 11 38 21  2  9 15 20 13 38 17  4 32 29  3 15 35]
 [ 8  1 36 37 26 14 36 22  7 14 21 18 10  3 24  8 25  2 15 32 27]
 [13 17 17  4  6 29  2 11 38 31 10 33 40  8 15 16 25 14  2 30 13]
 [38 10  3 10 31 22  9 40 16 15  4 33 18 17 32 39 32 36 30 22 20]
 [12 28 14 30 18 26  4 16 18 29 24 23  0 34 17 33 20 29 31 30 38]
 [13 34  0 39 28 31  9  4 37 13 36 23 39 33 30 10 33  8 23  0 31]
 [ 4 12 37 26 38 33 29 39  5 23 11 27 35 20 17 34  0  7 33 13 23]
 [40 39 24 13  2  2 20 18  0 17 31 33 35  9 12 34 32 36 36 36  7]
 [39 32  0 14 31  1 40 17  6 10 14 35  2  5  0  5  7  6 29  4 26]
 [29 29 25 32 37  4 11 25 13 35 14 14 17 16 31  5 16  3 34  6 20]
 [ 8 32 38 14 24 40 29 12  0  1  5 21 10  3 14 26 11 29  0  0  6]
 [33 37 31  9 16 28 25 17  7 33 22  5 11 11  2 26  1 18 10 11 23]
 [ 9 38  7 40 13 39 26  9 29  7 27 38  6 33 10 13 32 23 21 20  6]
 [23 23 22 19  7 16 30 33 36 32  2  6 23 21  4 19  9  1  8  6 10]
 [27 21 22 19 21 18 20 33 15 10  6 29 35 19 17 33 34 38 26 14 30]]
```



```
 Optimal Value Function:
 [[33.54344268 34.54501596 35.54604829 35.54305245 35.54909961 35.54360491
   35.54300585 35.55073309 35.54527802 35.54542493 35.55027205 35.5443252
   35.54412819 35.54408271 35.54540307 35.54657452 35.54726988 35.54665819
   35.550163   35.54959849 35.54486615]
  [34.54946529 35.5476995  36.54490019 36.54756628 36.54601819 36.54491969
   36.54438143 36.54639609 36.54751531 36.54613594 36.54658192 36.54466548
   36.5462664  36.54619663 36.54386311 36.5438596  36.54839213 36.54539287
   36.54370551 36.54404614 36.55296477]
  [35.5450191  36.54530299 37.54553992 37.54235394 37.55009364 37.54093791
   37.54481313 37.54305133 37.5421256  37.54561198 37.54804415 37.5406934
   37.54390808 37.5434523  37.54553323 37.54800001 37.54524327 37.54494759
   37.55073098 37.54919936 37.54800832]
  [35.54992964 36.54242934 37.54399736 37.54286738 37.54552732 37.54641147
   37.54480448 37.54389175 37.54892977 37.54692996 37.55353369 37.54299201
   37.54770432 37.54625715 37.54583155 37.54638394 37.54739861 37.54656309
   37.54863773 37.54768386 37.5501776 ]
  [35.54366177 36.54786082 37.54477521 37.54348638 37.54687895 37.54624782
   37.54729496 37.54891331 37.5424958  37.54864274 37.54619935 37.54601089
   37.55132518 37.55077368 37.54611532 37.54639431 37.54902744 37.54809752
   37.54567259 37.54412697 37.55328251]
  [35.55164621 36.54463871 37.55124908 37.54508354 37.54749434 37.5442795
   37.54626763 37.5428756  37.54927527 37.54237281 37.54900635 37.54559367
   37.54641379 37.54457091 37.54604713 37.54465794 37.54763239 37.5457502
   37.54538459 37.54568946 37.54615249]
```

RESULT

The Python program to elucidate value iteration in Gambler’s problem was developed and executed successfully.

EX.NO: 3 **RANDOM WALK USING MARKOV PROCESS**
DATE:

AIM

To generate random walk using Markov process.

RANDOM WALK

A random walk is a process for traversing a graph where at every step an outgoing edge chosen uniformly at random is followed. A Markov chain is similar except the outgoing edge is chosen according to an arbitrary fixed distribution**.**

MARKOV PROCESS

Markov process is a stochastic model that undergoes transitions from one state to another in a probabilistic manner. This process has the Markov property, which states that the future state depends only on the current state and not on the sequence of events that preceded it.

PROBLEM STATEMENT

States are represented by 'A', 'B', 'C', and 'D', and the transition probabilities between these states are defined in the transition_probabilities dictionary. For each state, the dictionary specifies the probability of transitioning to other states. The generate_random_walks function takes a starting state and the number of steps as input. It simulates a random walk by choosing the next state based on the transition probabilities defined in the transition_probabilities dictionary. The random.choices function is used to select the next state based on the provided weights (transition probabilities). The program then generates multiple random walks, each starting from state 'A', and prints the sequence of states traversed in each walk.

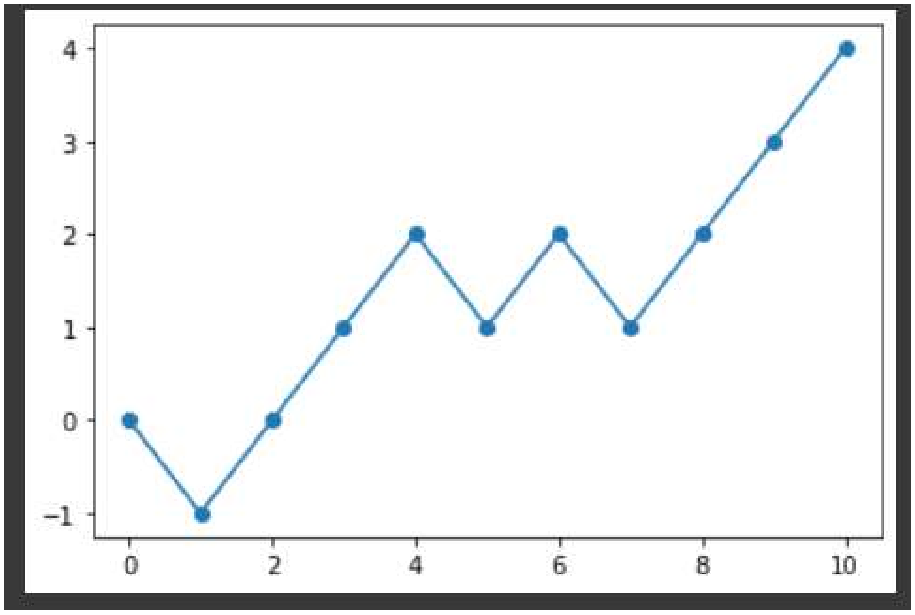
Algorithm:

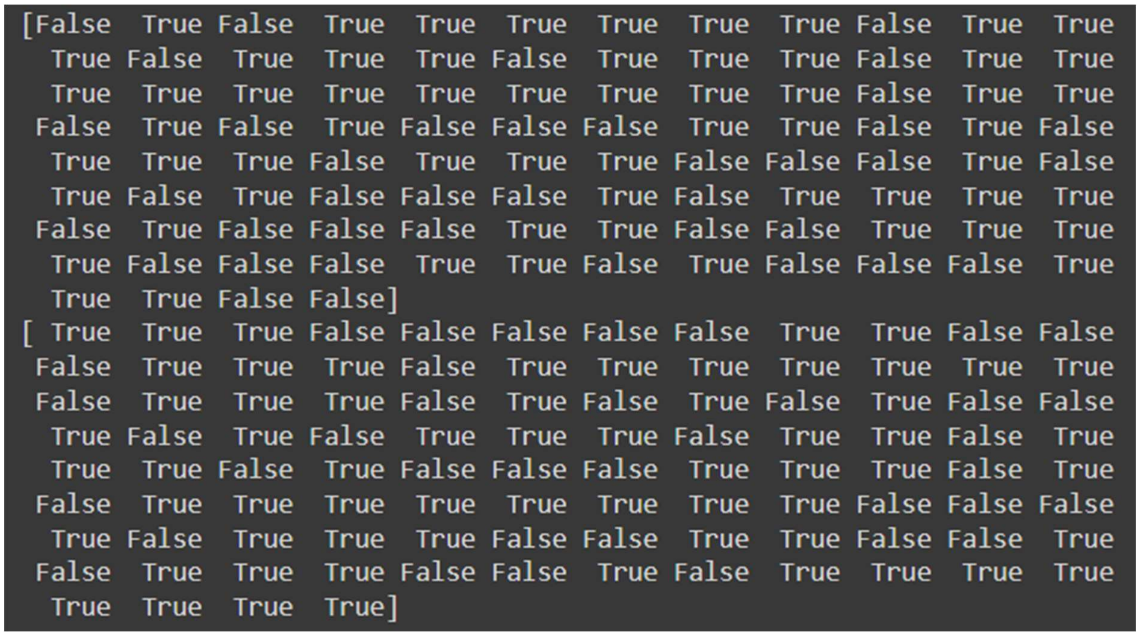
1. Identify all possible states of the system (e.g., positions on a line or graph).
2. Set up a transition matrix that defines the probabilities of moving from one state to another, where each state’s movement follows a Markov process (i.e., the next state depends only on the current state).
3. Choose an initial state to begin the random walk.
4. At each step, use the transition matrix to randomly move to the next state based on the current state's transition probabilities.
5. Continue the random walk by repeating the process for a given number of steps or until reaching a predefined stopping condition (e.g., hitting a boundary or returning to the start).
6. After completing the walk, analyze the results to understand the behavior of the random walk, such as expected time to return to the start or the distribution of states visited.

PROGRAM

```
import random

import numpy as np

import matplotlib.pyplot as plt

numpy.random.seed()

prob = [0.4,0.6]

start = 0

positions = [start]

rr = np.random.random(10)

downp = rr > prob[0]

upp = rr < prob[1]

t=[i for i in range(0,11)]

for idownp, iupp in zip(downp, upp):

  down = idownp

  up = iupp

  positions.append(positions[-1] - down + up)

plt.plot(t,positions,marker='o')

plt.show()

print(upp)

print(downp)
```

OUTPUT





```
[False  True False  True  True  True  True  True  True False  True  True
  True False  True  True  True False  True  True  True False  True  True
  True  True  True  True  True  True  True  True  True False  True  True
 False  True False  True False False False  True  True False  True False
  True  True  True False  True  True  True False False False  True False
  True False  True False False False  True False  True  True  True  True
 False  True False False False  True  True False False  True  True  True
  True False False False  True  True False  True False False False  True
  True  True False False]
[ True  True  True False False False False False  True  True False False
 False  True  True  True False  True  True  True  True  True  True  True
 False  True  True  True False  True False  True False  True False False
  True False  True False  True  True  True False  True  True False  True
  True  True False  True False False False  True  True  True False  True
 False  True  True  True  True  True  True  True  True False False False
  True False  True  True  True False False  True  True False False  True
 False  True  True  True False False  True False  True  True  True  True
  True  True  True  True]
```

RESULT

The Python program to generate random walk using Markov process was developed and executed successfully.

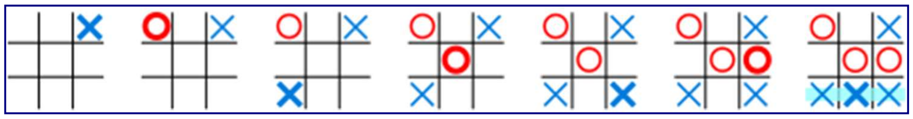
EX.NO: 4a **FRAMING TIC-TAC-TOE IN A RL WORLD USING POLICY ITERATION**
DATE:

AIM

To develop a simple reinforcement learning algorithm for agents to learn the game tic-tac-toe using value function using policy iteration.

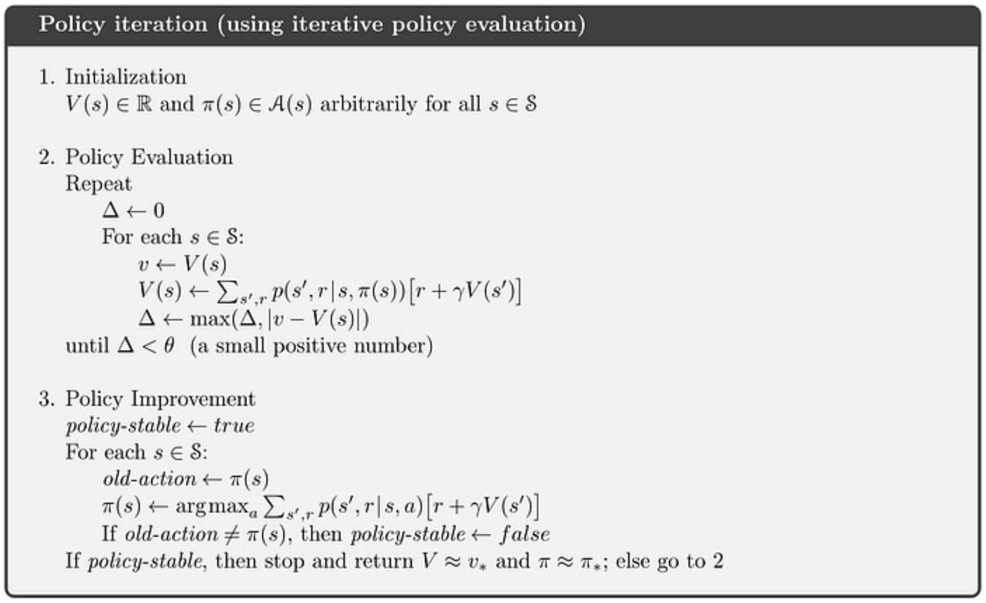
PROBLEM STATEMENT: TIC-TAC-TOE GAME

Tic-tac-toe, noughts and crosses, or Xs and Os is a paper-and-pencil game for two players who take turns marking the spaces in a three-by-three grid with X or O. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner. It is a solved game, with a forced draw assuming best play from both players. Tic-tac-toe is played on a three-by-three grid by two players, who alternately place the marks X and O in one of the nine spaces in the grid. In the following example, the first player (*X*) wins the game in seven steps:



POLICY ITERATIONALGORITHM

A policy is a mapping from states to actions, i.e., given a state, how many cars should Jack move overnight? Now, suppose Jack has some policy π, then given this π, the value of a state (say s) is the expected reward that Jack would get when he starts from s and follows π after that.



```
Policy iteration (using iterative policy evaluation)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
      $old\text{-}action \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
      If $old\text{-}action \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
   If $policy\text{-}stable$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

Policy Iteration

TIC TAC TOE PROBLEM FORMULATION

To formulate this reinforcement learning problem, the most important thing is to be clear about the 3 major components — state, action, and reward. The state of this game is the board state of both the agent and its opponent, so initialize a 3x3 board with zeros indicating available positions and update positions with 1 if player 1 takes a move and -1 if player 2 takes a move. The action is what positions a player can choose based on the current board state. Reward is between 0 and 1 and is only given at the end of the game.

Player Setting

Create a player class to represents agent, and the player is able to:

- Choose actions based on current estimation of the states
- Record all the states of the game
- Update states-value estimation after each game
- Save and load the policy

State-Value update

To update value estimation of states, apply policy iteration which is updated based on the formula below

$$V(s) \leftarrow \sum_{s',r} p(s', r | s, \pi(s))[r + \gamma V(s')]$$

Training

Now agent is able to learn by updating value estimation and our board is all set up, it is time to let two players play against each other. During training, the process for each player is:

- Look for available positions
- Choose action
- Update board state and add the action to player’s states
- Judge if reach the end of the game and give reward accordingly

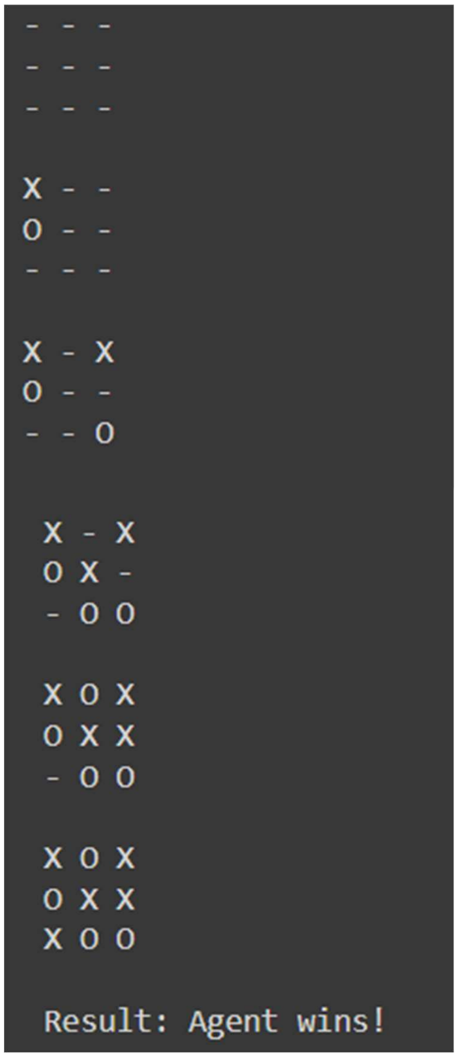
Algorithm:

1. Represent all possible Tic-Tac-Toe board configurations as states.
2. Start with an initial policy (e.g., choose random moves) for each board configuration.
3. For each state, calculate the expected rewards by following the current policy. This includes winning, losing, or drawing the game after a series of moves.
4. Update the value of each state based on future rewards.
5. For each state, try different actions (placing 'X' or 'O' in an empty spot).
6. Update the policy with the action that maximizes the expected reward.
7. Repeat the policy evaluation and improvement steps until the policy becomes stable.
8. Once the policy has converged, the final policy is the optimal strategy that maximizes the chances of winning the game for each board configuration.

PROGRAM

```
#Policy iteration

import random

def check_winner(board):
    win_conditions = [(0,1,2), (3,4,5), (6,7,8), (0,3,6), (1,4,7), (2,5,8), (0,4,8), (2,4,6)]
    for cond in win_conditions:
        if board[cond[0]] == board[cond[1]] == board[cond[2]] != 0:
            return board[cond[0]]
    return 0 if 0 not in board else None


def available_moves(board):
    return [i for i, x in enumerate(board) if x == 0]


class TicTacToePolicyIteration:
    def __init__(self, discount=0.9, epsilon=1e-6):
        self.discount = discount
        self.epsilon = epsilon
        self.values = {tuple(self.int_to_board(b)): 0 for b in range(3**9)}
        self.policy = {tuple(self.int_to_board(b)): random.choice(available_moves(self.int_to_board(b)))
                 for b in range(3**9) if available_moves(self.int_to_board(b))}


    def int_to_board(self, num):
        return [(num // (3**i)) % 3 for i in range(9)]


    def evaluate_policy(self):
        while True:
            delta = 0
            for board in self.values:
                winner = check_winner(list(board))
                if winner is not None:
                    self.values[board] = 1 if winner == 1 else -1 if winner == 2 else 0
                    continue
                move = self.policy[board]
                new_board = list(board)
                new_board[move] = 1  # Simulate player's move
                 reward = 1 if check_winner(new_board) == 1 else -1 if check_winner(new_board)
== 2 else 0
                new_value = reward + self.discount * self.values[tuple(new_board)]
                delta = max(delta, abs(self.values[board] - new_value))
                self.values[board] = new_value
            if delta < self.epsilon:
                break


    def improve_policy(self):
        policy_stable = True
        for board in self.policy:
            old_action = self.policy[board]
            best_action = None
            best_value = -float('inf')
            for move in available_moves(board):
                new_board = list(board)
                new_board[move] = 1  # Simulate player's move
                move_value = self.values[tuple(new_board)]
                if move_value > best_value:
                    best_value = move_value
                    best_action = move
            self.policy[board] = best_action
            if old_action != best_action:
                policy_stable = False
        return policy_stable


    def policy_iteration(self):
        while True:
            self.evaluate_policy()
            if self.improve_policy():
                break


    def get_best_move(self, board):
        return self.policy[tuple(board)]


    def play(self):
        board = [0] * 9
        while True:
            print_board(board)
            if (winner := check_winner(board)) is not None:
                print("Result:", "Draw" if winner == 0 else "Agent wins!" if winner == 1 else "Opponent wins!")
                break
            board[self.get_best_move(board)] = 1
            if (winner := check_winner(board)) is not None:
                print_board(board)
                print("Result:", "Draw" if winner == 0 else "Agent wins!" if winner == 1 else "Opponent wins!")
                break
            board[random.choice(available_moves(board))] = 2


def print_board(board):
    symbols = {0: '-', 1: 'X', 2: 'O'}
    for i in range(0, 9, 3):
        print(f"{symbols[board[i]]} {symbols[board[i+1]]} {symbols[board[i+2]]}")
    print()
game = TicTacToePolicyIteration()
game.policy_iteration()
game.play()
```

OUTPUT



```
- - -
- - -
- - -

X - -
O - -
- - -

X - X
O - -
- - O

 X - X
 O X -
 - O O

 X O X
 O X X
 - O O

 X O X
 O X X
 X O O

 Result: Agent wins!
```

RESULT

The Python program to develop a simple reinforcement learning algorithm for agents to learn the game tic-tac-toe using value function using policy iteration was developed and executed successfully.

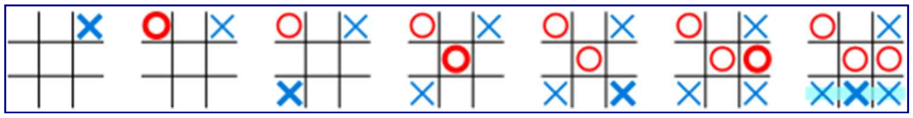
EX.NO: 4b **FRAMING TIC-TAC-TOE IN A RL WORLD USING VALUE ITERATION**
DATE:

AIM

To develop a simple reinforcement learning algorithm for agents to learn the game tic-tac-toe using value function.

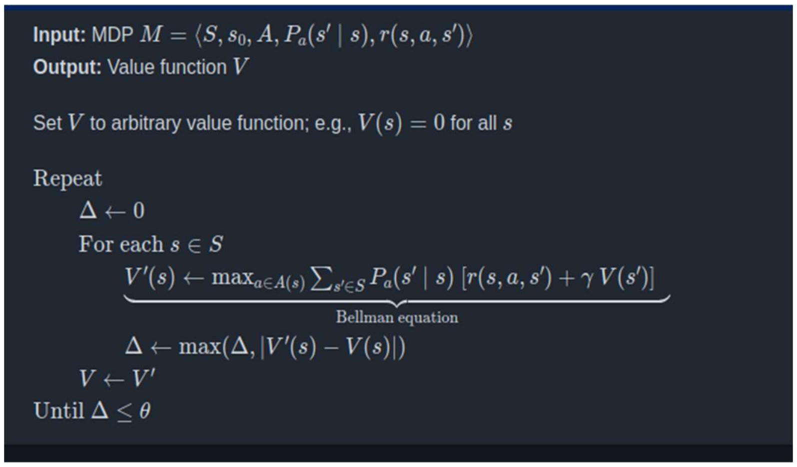
PROBLEM STATEMENT: TIC-TAC-TOE GAME

Tic-tac-toe, noughts and crosses, or Xs and Os is a paper-and-pencil game for two players who take turns marking the spaces in a three-by-three grid with X or O. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner. It is a solved game, with a forced draw assuming best play from both players. Tic-tac-toe is played on a three-by-three grid by two players, who alternately place the marks X and O in one of the nine spaces in the grid. In the following example, the first player (*X*) wins the game in seven steps:



VALUE ITERATIONALGORITHM

Value Iteration is a method for finding the optimal value function **V*** by solving the Bellman equations iteratively. It uses the concept of dynamic programming to maintain a value function **V** that approximates the optimal value function V*, iteratively improving V until it converges to **V*** (or close to it).



```
Input: MDP $M = \langle S, s_0, A, P_a(s' \mid s), r(s,a,s') \rangle$
Output: Value function $V$

Set $V$ to arbitrary value function; e.g., $V(s) = 0$ for all $s$

Repeat
    $\Delta \leftarrow 0$
    For each $s \in S$
        $\underbrace{V'(s) \leftarrow \max_{a \in A(s)} \sum_{s' \in S} P_a(s' \mid s)\ [r(s,a,s') + \gamma\, V(s')]}_{\text{Bellman equation}}$
        $\Delta \leftarrow \max(\Delta, |V'(s) - V(s)|)$
    $V \leftarrow V'$
Until $\Delta \leq \theta$
```

Value Iteration

TIC TAC TOE PROBLEM FORMULATION

To formulate this reinforcement learning problem, the most important thing is to be clear about the 3 major components — state, action, and reward. The state of this game is the board state of both the agent and its opponent, so initialize a 3x3 board with zeros indicating available positions and update positions with 1 if player 1 takes a move and -1 if player 2 takes a move. The action is what positions a player can choose based on the current board state. Reward is between 0 and 1 and is only given at the end of the game.

Player Setting

Create a player class to represents agent, and the player is able to:

- Choose actions based on current estimation of the states
- Record all the states of the game
- Update states-value estimation after each game
- Save and load the policy

State-Value update

To update value estimation of states, apply value iteration which is updated based on the formula below

$$V(S_t) \leftarrow V(S_t) + \alpha \Big[V(S_{t+1}) - V(S_t) \Big]$$

Training

Now agent is able to learn by updating value estimation and our board is all set up, it is time to let two players play against each other. During training, the process for each player is:

- Look for available positions
- Choose action
- Update board state and add the action to player's states
- Judge if reach the end of the game and give reward accordingly

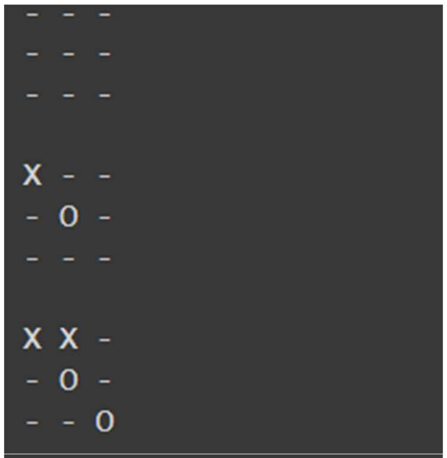
Algorithm:

1. Represent all possible Tic-Tac-Toe board configurations as states.
2. Assign an initial value to each state, typically starting with 0 for all states except terminal states (win, loss, draw).
3. For each possible board configuration, calculate the expected reward for each possible action (placing 'X' or 'O' in an empty space).
4. For each state, update its value by selecting the action that maximizes the expected reward, considering the outcomes of winning, losing, or drawing after the move.
5. Repeat the process for all states until the values converge.
6. Once the values stabilize, derive the optimal policy by selecting the action that leads to the highest value for each state (i.e., the best move for any board configuration).

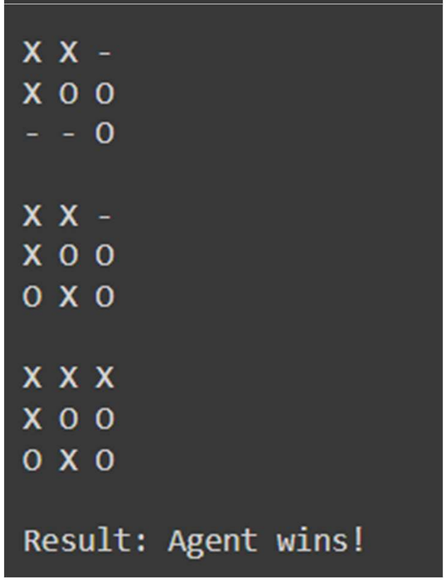
PROGRAM

```
#Value iteration
import random
def check_winner(board):
    win_conditions = [(0,1,2), (3,4,5), (6,7,8), (0,3,6), (1,4,7), (2,5,8), (0,4,8), (2,4,6)]
    for cond in win_conditions:
        if board[cond[0]] == board[cond[1]] == board[cond[2]] != 0:
            return board[cond[0]]
    return 0 if 0 not in board else None
def available_moves(board):
    return [i for i, x in enumerate(board) if x == 0]
class TicTacToeValueIteration:
    def __init__(self, discount=0.9, epsilon=1e-6):
        self.discount = discount
        self.epsilon = epsilon
        self.values = {tuple(self.int_to_board(b)): 0 for b in range(3**9)}

    def int_to_board(self, num):
        return [(num // (3**i)) % 3 for i in range(9)]

    def value_iteration(self):
        while True:
            delta = 0
            new_values = self.values.copy()
            for board in self.values:
                winner = check_winner(list(board))
                if winner is not None:
                    new_values[board] = 1 if winner == 1 else -1 if winner == 2 else 0
                    continue
                best_value = -float('inf')
                for move in available_moves(board):
                    new_board = list(board)
                    new_board[move] = 1  # Simulate player's move
                    reward = 1 if check_winner(new_board) == 1 else -1 if check_winner(new_board) == 2 else 0
                    move_value = reward + self.discount * self.values[tuple(new_board)]
                    best_value = max(best_value, move_value)
                delta = max(delta, abs(self.values[board] - best_value))
                new_values[board] = best_value
            self.values = new_values
            if delta < self.epsilon:
                break


    def get_best_move(self, board):
        best_move = None
        best_value = -float('inf')
        for move in available_moves(board):
            new_board = list(board)
            new_board[move] = 1  # Simulate player's move
            move_value = self.values[tuple(new_board)]
            if move_value > best_value:
                best_value = move_value
                best_move = move
        return best_move

    def play(self):
        board = [0] * 9
        while True:
            print_board(board)
            if (winner := check_winner(board)) is not None:
                print("Result:", "Draw" if winner == 0 else "Agent wins!" if winner == 1 else "Opponent wins!")
                break
            board[self.get_best_move(board)] = 1
            if (winner := check_winner(board)) is not None:
                print_board(board)
                print("Result:", "Draw" if winner == 0 else "Agent wins!" if winner == 1 else "Opponent wins!")
                break
            board[random.choice(available_moves(board))] = 2


def print_board(board):
    symbols = {0: '-', 1: 'X', 2: 'O'}
    for i in range(0, 9, 3):
        print(f"{symbols[board[i]]} {symbols[board[i+1]]} {symbols[board[i+2]]}")
    print()


# Run the game
game = TicTacToeValueIteration()
game.value_iteration()
game.play()
```

OUTPUT



```
- - -
- - -
- - -

X - -
- O -
- - -

X X -
- O -
- - O
```



```
X X -
X O O
- - O

X X -
X O O
O X O

X X X
X O O
O X O

Result: Agent wins!
```

RESULT

The Python program to develop a simple reinforcement learning algorithm for agents to learn the game tic-tac-toe using value function using value iteration was developed and executed successfully.

EX.NO: 5 **BLACKJACK WITH FIRST VISIT MONTE CARLO**
DATE:

AIM

To implement First-Visit Monte Carlo Policy evaluation for Blackjack game.

BLACKJACK - PROBLEM STATEMENT

The game begins with two cards dealt to both dealer and player. One of the dealer's cards is face up and the other is face down. If the player has 21 immediately (an ace and a 10-card), it is called a natural. He then wins unless the dealer also has a natural, in which case the game is a draw. If the player does not have a natural, then he can request additional cards, one by one (hits), until he either stops (sticks) or exceeds 21 (goes bust). If he goes bust, he loses; if he sticks, then it becomes the dealer's turn. The dealer hits or sticks according to a fixed strategy without choice: he sticks on any sum of 17 or greater, and hits otherwise. If the dealer goes bust, then the player wins; otherwise, the outcome — win, lose, or draw — is determined by whose final sum is closer to 21. If the player holds an ace that he could count as 11 without going bust, then the ace is said to be usable.

The Pack

The standard 52-card pack is used, but in most casinos several decks of cards are shuffled together. The six-deck game (312 cards) is the most popular. In addition, the dealer uses a blank plastic card, which is never dealt, but is placed toward the bottom of the pack to indicate when it will be time for the cards to be reshuffled. When four or more decks are used, they are dealt from a shoe (a box that allows the dealer to remove cards one at a time, face down, without actually holding one or more packs).

Object of the Game

Each participant attempts to beat the dealer by getting a count as close to 21 as possible, without going over 21.

Card Values/scoring

It is up to each individual player if an ace is worth 1 or 11. Face cards are 10 and any other card is its pip value.

Betting

Before the deal begins, each player places a bet, in chips, in front of them in the designated area. Minimum and maximum limits are established on the betting, and the general limits are from $2 to $500.

The Play

The player to the left goes first and must decide whether to "stand" (not ask for another card) or "hit" (ask for another card in an attempt to get closer to a count of 21, or even hit 21 exactly). Thus, a player may stand on the two cards originally dealt to them, or they may ask the dealer for additional cards, one at a time, until deciding to stand on the total (if it is 21 or under), or goes "bust" (if it is over 21). In the latter case, the player loses and the dealer collects the bet wagered. The dealer then turns to the next player to their left and serves them in the same manner.

The combination of an ace with a card other than a ten-card is known as a "soft hand," because the player can count the ace as a 1 or 11, and either draw cards or not. For example with a "soft 17" (an ace and a 6), the total is 7 or 17. While a count of 17 is a good hand, the player may wish to draw for a higher total. If the draw creates a bust hand by counting the ace as an 11, the player simply counts the ace as a 1 and continues playing by standing or "hitting" (asking the dealer for additional cards, one at a time).

The Dealer's Play

When the dealer has served every player, the dealer's face-down card is turned up. If the total is 17 or more, it must stand. If the total is 16 or under, they must take a card. The dealer must continue to take cards until the total is 17 or more, at which point the dealer must stand. If the dealer has an ace, and counting it as 11 would bring the total to 17 or more (but not over 21), the dealer must count the ace as 11 and stand. The dealer's decisions, then, are automatic on all plays, whereas the player always has the option of taking one or more cards.

Signaling Intentions

When a player's turn comes, they can say "Hit" or can signal for a card by scratching the table with a finger or two in a motion toward themselves, or they can wave their hand in the same motion that would say to someone "Come here!" When the player decides to stand, they can say "Stand" or "No more," or can signal this intention by moving their hand sideways, palm down and just above the table.

Splitting Pairs

If a player's first two cards are of the same denomination, such as two jacks or two sixes, they may choose to treat them as two separate hands when their turn comes around. The amount of the original bet then goes on one of the cards, and an equal amount must be placed as a bet on the other card. The player first plays the hand to their left by standing or hitting one or more times; only then is the hand to the right played. The two hands are thus treated separately, and the dealer settles with each on its own merits. With a pair of aces, the player is given one card for each ace and may not draw again. Also, if a ten-card is dealt to one of these aces, the payoff is equal to the bet (not one and one-half to one, as with a blackjack at any other time).

Doubling Down

Another option open to the player is doubling their bet when the original two cards dealt total 9, 10, or 11. When the player's turn comes, they place a bet equal to the original bet, and the dealer gives the player just one card, which is placed face down and is not turned up until the bets are settled at the end of the hand. With two fives, the player may split a pair, double down, or just play the hand in the regular way. Note that the dealer does not have the option of splitting or doubling down.

FIRST-VISIT MONTE CARLO POLICY EVALUATION

```
Initialize:
    $\pi \leftarrow$ policy to be evaluated
    $V \leftarrow$ an arbitrary state-value function
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
    (a) Generate an episode using $\pi$
    (b) For each state $s$ appearing in the episode:
            $R \leftarrow$ return following the first occurrence of $s$
            Append $R$ to $Returns(s)$
            $V(s) \leftarrow$ average($Returns(s)$)
```

ALGORITHM

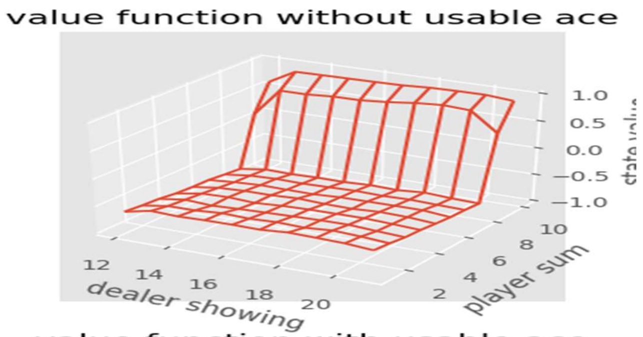
1. Simulate the Blackjack environment
2. Define the policy function which takes the current state and check if the score is greater than or equal to 20, if yes we return 0 else we return 1. i.e If the score is greater than or equal to 20 we stand (0) else we hit (1)
3. Define a function called generate_episode for generating epsiodes
4. Perform First Visit MC Prediction
5. Define the function plot_blackjack for plotting the value function and we can see how our value function is attaining the convergence.

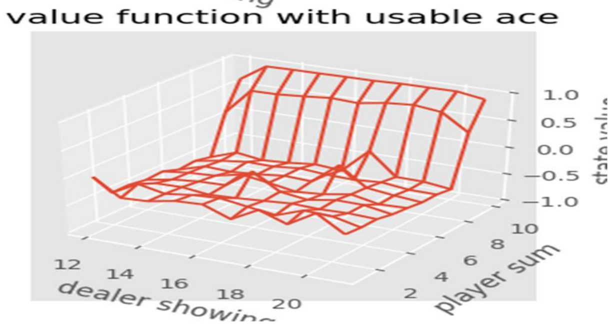
PROGRAM

```
import numpy as np

import random

import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import Axes3D

from collections import defaultdict
```

```
card_values = {'2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, '10': 10, 'J': 10, 'Q': 10, 'K': 10,
'A': 11}
def deal_card():
    return random.choice(list(card_values.keys()))


def get_hand_value(hand):
    value = sum(card_values[card] for card in hand)
    num_aces = hand.count('A')
    while value > 21 and num_aces:
        value -= 10
        num_aces -= 1
    return value


def blackjack_policy(hand):
    return 'hit' if get_hand_value(hand) < 20 else 'stand'


def play_blackjack(policy):
    player_hand = [deal_card(), deal_card()]
    dealer_hand = [deal_card(), deal_card()]
    while policy(player_hand) == 'hit':
        player_hand.append(deal_card())
        if get_hand_value(player_hand) > 21:
            return -1  # Player busts
    while get_hand_value(dealer_hand) < 17:
        dealer_hand.append(deal_card())
        if get_hand_value(dealer_hand) > 21:
            return 1  # Dealer busts
    player_value = get_hand_value(player_hand)
    dealer_value = get_hand_value(dealer_hand)
    if player_value > dealer_value:
        return 1  # Player wins
    elif player_value < dealer_value:
        return -1  # Dealer wins
    else:
        return 0  # Tie

def first_visit_monte_carlo(num_episodes):
    state_action_returns = defaultdict(list)
    state_action_counts = defaultdict(int)
    Q = defaultdict(float)
    for episode in range(num_episodes):
        player_hand = [deal_card(), deal_card()]
        episode_trace = []
        while True:
            action = blackjack_policy(player_hand)
            episode_trace.append((tuple(player_hand), action))
            if action == 'hit':
                player_hand.append(deal_card())
                if get_hand_value(player_hand) > 21:
                    episode_trace.append((tuple(player_hand), 'bust'))
                    break
            else:
                break
        reward = play_blackjack(blackjack_policy)
        for state, action in episode_trace:
            if action != 'bust':
                state_action_returns[(state, action)].append(reward)
                state_action_counts[(state, action)] += 1
        for state_action, returns in state_action_returns.items():
            Q[state_action] = np.mean(returns)
    return Q, state_action_counts

def plot_3d(Q):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    x = []
    y = []
    z = []
    for (state, action), value in Q.items():
        if action == 'hit':
            x.append(get_hand_value(state))
            y.append(1)  # hit
        elif action == 'stand':
            x.append(get_hand_value(state))
            y.append(0)  # stand
        z.append(value)
    ax.scatter(x, y, z, c='r', marker='o')
    ax.set_xlabel('Hand Value')
    ax.set_ylabel('Action')
    ax.set_zlabel('Value Estimate')
    ax.set_yticks([0, 1])
    ax.set_yticklabels(['Stand', 'Hit'])
    plt.show()

def plot_line(Q):
    hand_values = sorted(set(get_hand_value(state) for state, action in Q.keys()))
    actions = ['stand', 'hit']
    values_hit = [Q.get((tuple([str(value)] * 2), 'hit'), 0) for value in hand_values]
    values_stand = [Q.get((tuple([str(value)] * 2), 'stand'), 0) for value in hand_values]

    plt.figure(figsize=(10, 6))
    plt.plot(hand_values, values_hit, label='Hit', color='blue', marker='o')
    plt.plot(hand_values, values_stand, label='Stand', color='green', marker='o')
    plt.xlabel('Hand Value')
    plt.ylabel('Value Estimate')
    plt.title('State-Action Value Function: Hit vs Stand')
    plt.legend()
    plt.grid(True)
    plt.show()

# Main execution
num_episodes = 1000
Q, state_action_counts = first_visit_monte_carlo(num_episodes)

# Print some of the results
print("State-Action Value Estimates:")
for (state, action), value in sorted(Q.items()):
    print(f"State: {state}, Action: {action}, Value: {value:.2f}")

# Plot the 3D representation
plot_3d(Q)

# Plot the line representation
plot_line(Q)
```

OUTPUT





RESULT

The Python program to implement First-Visit Monte Carlo Policy evaluation for Blackjack game was developed and executed successfully.

EX.NO: 6 EVALUATE WINDY GRID WORLD WITH KING'S MOVES
DATE:

AIM

To evaluate Windy GridWorld environment using SARSA method.

SARSA ALGORITHM FOR A WINDY GRIDWORLD ENVIRONMENT

A standard gridworld Figure 1, with start and goal states, but with one difference: there is a crosswind upward through the middle of the grid. The actions are the standard four up, down, right, and left but in the middle region the resultant next states are shifted upward by a wind, the strength of which varies from column to column. The strength of the wind is given below each column, in number of cells shifted, upward. For example, if you are one cell to the right of the goal, then the action left takes you to the cell just above the goal. Let us treat this as an undiscounted episodic task, with constant rewards of 1 until the goal state is reached. Figure 6.11 shows the result of applying-greedy Sarsa to this task, with = 01, =05, and the initial values Q(sa) = 0 for all sa. The increasing slope of the graph shows that the goal is reached more and more quickly over time. By 8000 time steps, the greedy policy (shown inset) was long since optimal; continued-greedy exploration kept the average episode length at about 17 steps, two more than the minimum of 15. Note that Monte Carlo methods cannot easily be used on this task because termination is not guaranteed for all policies. If a policy was ever found that caused the agent to stay in the same state, then the next episode would never end. Step-by-step learning methods such as Sarsa do not have this problem because they quickly learn during the episode that such policies are poor, and switch to something else

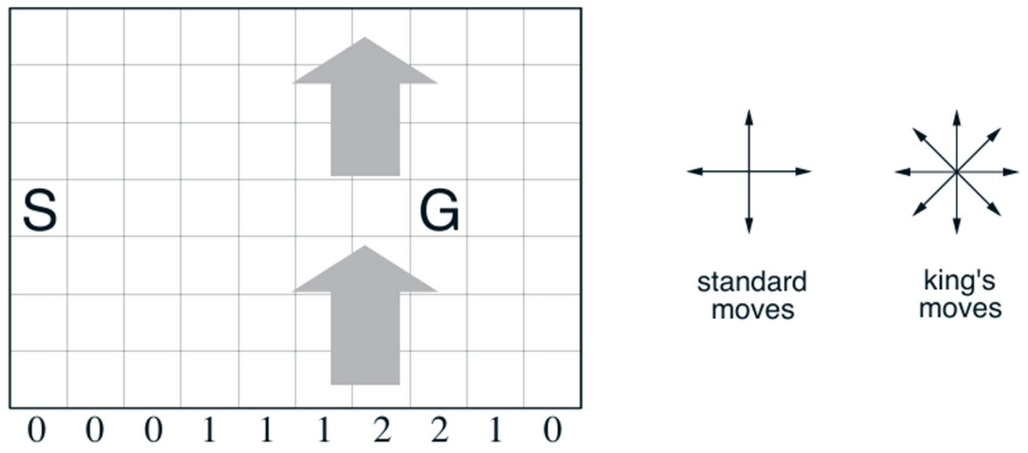


Figure 1: Windy Grid World

```
Initialize $V(s)$ arbitrarily (but set to 0 if $s$ is terminal)
$V_{\text{old}} \leftarrow 0$
Repeat (for each episode):
    Initialize $E(s) = 0$, for all $s \in \mathcal{S}$
    Initialize $S$
    Repeat (for each step of episode):
        $A \leftarrow$ action given by $\pi$ for $S$
        Take action $A$, observe reward, $R$, and next state, $S'$
        $\Delta \leftarrow V(S) - V_{\text{old}}$
        $V_{\text{old}} \leftarrow V(S')$
        $\delta \leftarrow R + \gamma V(S') - V(S)$
        $E(S) \leftarrow (1-\alpha)E(S) + 1$
        For all $s \in \mathcal{S}$:
            $V(s) \leftarrow V(s) + \alpha(\delta + \Delta)E(s)$
            $E(s) \leftarrow \gamma\lambda E(s)$
        $V(S) \leftarrow V(S) - \alpha\Delta$
        $S \leftarrow S'$
    until $S$ is terminal
```

Figure 2 On-policy TD Control Algorithm- SARSA

The SARSA algorithm works by carrying out actions based on rewards received from previous actions. To do this, SARSA stores a table of state (S)-action (A) estimate pairs for each Q-value. This table is known as a Q-table, while the state-action pairs are denoted as Q(S, A). The SARSA process starts by initializing Q(S, A) to arbitrary values Figure 2. In this step, the initial current state (S) is set, and the initial action (A) is selected by using an epsilon-greedy algorithm policy based on current Q-values. An epsilon-greedy policy balances the use of exploitation and exploration methods in the learning process to select the action with the highest estimated reward.

ALGORITHM:

1. START
2. Initialize the rewards, state space, and hyperparameters:
    a. State space S: The set of possible states (e.g., battery levels for the robot).
    b. Action space A: The set of possible actions (e.g., search, wait, recharge).
    c. Initialize the Q-table Q(s,a) with random values or zeros for all state-action pairs.
    d. Set hyperparameters: Learning rate α, discount factor γ, and exploration rate ϵ.
3. Define the reward function, that takes the current state s and action a as inputs and returns the reward r.
4. Define the state transition function, that takes the current state s and action a as inputs and returns next s′ state.
5. Implement SARSA Algorithm for given environment.
6. After completing all the episodes, display the plots and optimal policy.
7. STOP

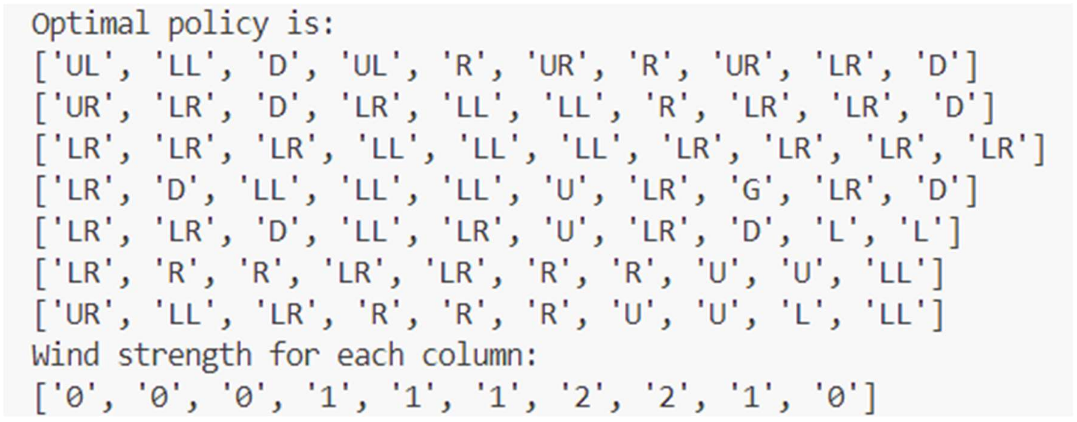
PROGRAM:

```
import numpy as np
import matplotlib
matplotlib.use('Agg')
```

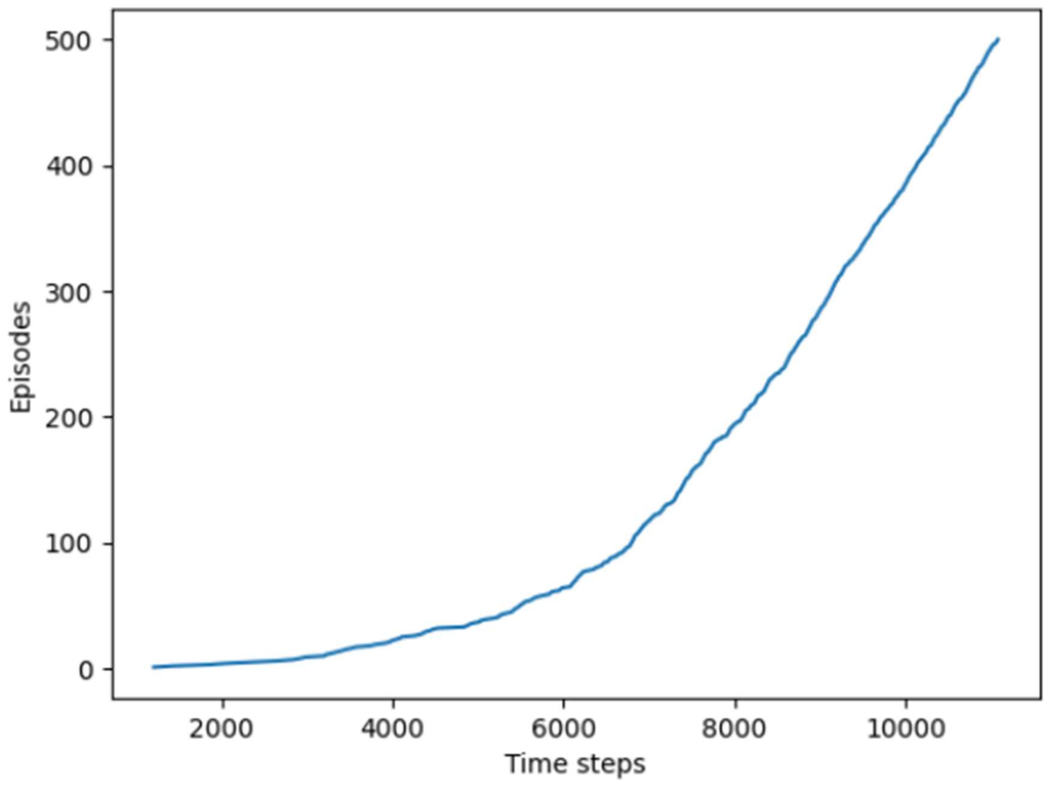
```
import matplotlib.pyplot as plt
# World dimensions
WORLD_HEIGHT = 7
WORLD_WIDTH = 10
# Wind strength for each column
WIND = [0, 0, 0, 1, 1, 1, 2, 2, 1, 0]
# Possible actions (including King's moves)
ACTION_UP = 0
ACTION_DOWN = 1
ACTION_LEFT = 2
ACTION_RIGHT = 3
ACTION_UP_LEFT = 4
ACTION_UP_RIGHT = 5
ACTION_DOWN_LEFT = 6
ACTION_DOWN_RIGHT = 7
# Probability for exploration
EPSILON = 0.1
# Learning rate
ALPHA = 0.5
# Reward for each step
REWARD = -1.0
# Start and Goal positions
START = [3, 0]
GOAL = [3, 7]
# All possible actions
ACTIONS = [ACTION_UP, ACTION_DOWN, ACTION_LEFT, ACTION_RIGHT,
ACTION_UP_LEFT, ACTION_UP_RIGHT, ACTION_DOWN_LEFT,
ACTION_DOWN_RIGHT]

def step(state, action):
    i, j = state
    if action == ACTION_UP:
        return [max(i - 1 - WIND[j], 0), j]
    elif action == ACTION_DOWN:
        return [max(min(i + 1 - WIND[j], WORLD_HEIGHT - 1), 0), j]
    elif action == ACTION_LEFT:
        return [max(i - WIND[j], 0), max(j - 1, 0)]
    elif action == ACTION_RIGHT:
        return [max(i - WIND[j], 0), min(j + 1, WORLD_WIDTH - 1)]
    elif action == ACTION_UP_LEFT:
        return [max(i - 1 - WIND[max(j - 1, 0)], 0), max(j - 1, 0)]
    elif action == ACTION_UP_RIGHT:
        return [max(i - 1 - WIND[min(j + 1, WORLD_WIDTH - 1)], 0), min(j + 1,
WORLD_WIDTH - 1)]
```

```
    elif action == ACTION_DOWN_LEFT:
        return [max(min(i + 1 - WIND[max(j - 1, 0)], WORLD_HEIGHT - 1), 0), max(j - 1, 0)]
    elif action == ACTION_DOWN_RIGHT:
        return [max(min(i + 1 - WIND[min(j + 1, WORLD_WIDTH - 1)], WORLD_HEIGHT -
1), 0), min(j + 1, WORLD_WIDTH - 1)]
    else:
        assert False
def episode(q_value):
    time = 0
    state = START
    if np.random.binomial(1, EPSILON) == 1:
        action = np.random.choice(ACTIONS)
    else:
        values_ = q_value[state[0], state[1], :]
        action = np.random.choice([action_ for action_, value_ in enumerate(values_) if value_
== np.max(values_)])
    while state != GOAL:
        next_state = step(state, action)
        if np.random.binomial(1, EPSILON) == 1:
            next_action = np.random.choice(ACTIONS)
        else:
            values_ = q_value[next_state[0], next_state[1], :]
            next_action = np.random.choice([action_ for action_, value_ in enumerate(values_) if
value_ == np.max(values_)])
        # Q-learning update rule
        q_value[state[0], state[1], action] += \
            ALPHA * (REWARD + np.max(q_value[next_state[0], next_state[1], :]) -
                    q_value[state[0], state[1], action])
        state = next_state
        action = next_action
        time += 1
    return time
def figure_6_3():
    q_value = np.zeros((WORLD_HEIGHT, WORLD_WIDTH, len(ACTIONS)))
    episode_limit = 500
    steps = []
    ep = 0
    while ep < episode_limit:
        steps.append(episode(q_value))
        ep += 1
    steps = np.add.accumulate(steps)
    plt.plot(steps, np.arange(1, len(steps) + 1))
    plt.xlabel('Time steps')
    plt.ylabel('Episodes')
    plt.savefig('figure_6_3.png')
    plt.close()
    optimal_policy = []
    for i in range(0, WORLD_HEIGHT):
        optimal_policy.append([])
        for j in range(0, WORLD_WIDTH):
            if [i, j] == GOAL:
                optimal_policy[-1].append('G')
                continue
            bestAction = np.argmax(q_value[i, j, :])
            if bestAction == ACTION_UP:
                optimal_policy[-1].append('U')
            elif bestAction == ACTION_DOWN:
                optimal_policy[-1].append('D')
            elif bestAction == ACTION_LEFT:
                optimal_policy[-1].append('L')
            elif bestAction == ACTION_RIGHT:
                optimal_policy[-1].append('R')
            elif bestAction == ACTION_UP_LEFT:
                optimal_policy[-1].append('UL')
            elif bestAction == ACTION_UP_RIGHT:
                optimal_policy[-1].append('UR')
            elif bestAction == ACTION_DOWN_LEFT:
                optimal_policy[-1].append('LL')
            elif bestAction == ACTION_DOWN_RIGHT:
                optimal_policy[-1].append('LR')

    print('Optimal policy is:')
    for row in optimal_policy:
        print(row)
    print('Wind strength for each column:\n{}'.format([str(w) for w in WIND]))
if __name__ == '__main__':
    figure_6_3()
```

OUTPUT



```
Optimal policy is:
['UL', 'LL', 'D', 'UL', 'R', 'UR', 'R', 'UR', 'LR', 'D']
['UR', 'LR', 'D', 'LR', 'LL', 'LL', 'R', 'LR', 'LR', 'D']
['LR', 'LR', 'LR', 'LL', 'LL', 'LL', 'LR', 'LR', 'LR', 'LR']
['LR', 'D', 'LL', 'LL', 'LL', 'U', 'LR', 'G', 'LR', 'D']
['LR', 'LR', 'D', 'LL', 'LR', 'U', 'LR', 'D', 'L', 'L']
['LR', 'R', 'R', 'LR', 'LR', 'R', 'R', 'U', 'U', 'LL']
['UR', 'LL', 'LR', 'R', 'R', 'R', 'U', 'U', 'L', 'LL']
Wind strength for each column:
['0', '0', '0', '1', '1', '1', '2', '2', '1', '0']
```



RESULT

Thus, the evaluation of Windy Grid World environment with King's move using SARSA method has been implemented and executed successfully.

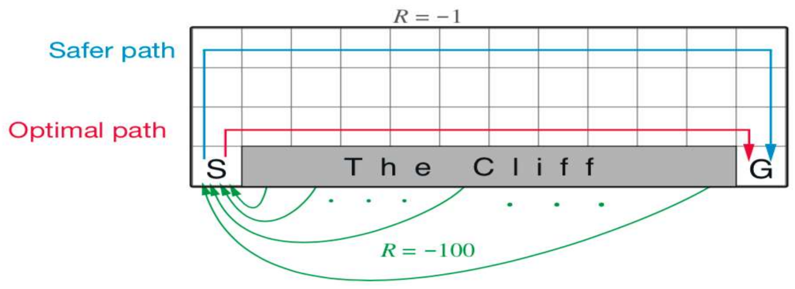
EX.NO: 7 **OFF-POLICY TD CONTROL ALGORITHM FOR CLIFF WALKING**
DATE:

AIM

To implement the Off-Policy TD algorithm known as Q-learning for Cliff Walking.

PROBLEM STATEMENT - CLIFF WALKING

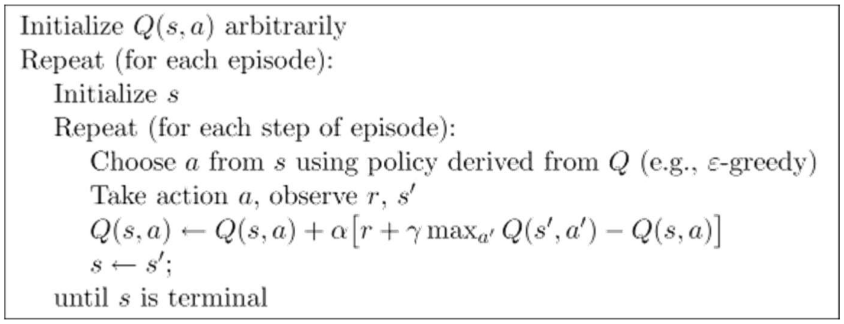
The cliff walking problem is a grid problem with a 4 x 12 board. The agent starts in the bottom left corner and must reach the bottom right corner. The agent must step into the cliff that segregates those tiles.



A) Define the Environment

1. Create a gridworld representation of the environment with states, actions, and rewards.
2. Define the state space, which includes the agent's position on the grid.
3. Define the action space, which includes possible movements (up, down, left, right).
4. Specify rewards and penalties for different states and actions. In cliff-walking, we typically have a large negative reward for falling off the cliff.

B) Initialize Q-Table to values of 0s



```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
   Initialize $s$
   Repeat (for each step of episode):
      Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
      Take action $a$, observe $r$, $s'$
      $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
      $s \leftarrow s'$;
   until $s$ is terminal
```

PROGRAM:

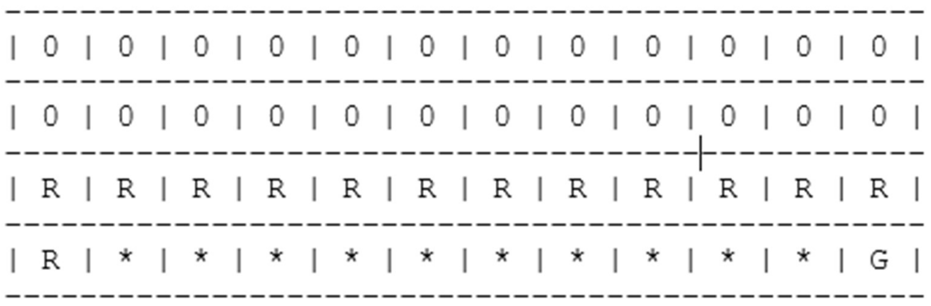
```
import numpy as np
ROWS = 4
COLS = 12
S = (3, 0)
G = (3, 11)
class Cliff:
    def __init__(self):
        self.end = False
        self.pos = S
        self.board = np.zeros([4, 12])
        # add cliff marked as -1
        self.board[3, 1:11] = -1

    def nxtPosition(self, action):
        if action == "up":
            nxtPos = (self.pos[0] - 1, self.pos[1])
        elif action == "down":
            nxtPos = (self.pos[0] + 1, self.pos[1])
        elif action == "left":
            nxtPos = (self.pos[0], self.pos[1] - 1)
        else:
            nxtPos = (self.pos[0], self.pos[1] + 1)
        # check legitimacy
        if nxtPos[0] >= 0 and nxtPos[0] <= 3:
            if nxtPos[1] >= 0 and nxtPos[1] <= 11:
                self.pos = nxtPos
        if self.pos == G:
            self.end = True
            print("Game ends reaching goal")
        if self.board[self.pos] == -1:
            self.end = True
            print("Game ends falling off cliff")
        return self.pos

    def giveReward(self):
        # give reward
        if self.pos == G:
            return -1
        if self.board[self.pos] == 0:
            return -1
        return -100

    def show(self):
        for i in range(0, ROWS):
            print('-------------------------------------------------')
            out = '| '
            for j in range(0, COLS):
                if self.board[i, j] == -1:
                    token = '*'
                if self.board[i, j] == 0:
                    token = '0'
                if (i, j) == self.pos:
                    token = 'S'
                if (i, j) == G:
                    token = 'G'
                out += token + ' | '
            print(out)
        print('-------------------------------------------------')
```

```
class Agent:
    def __init__(self, exp_rate=0.3, lr=0.1, sarsa=True):
        self.cliff = Cliff()
        self.actions = ["up", "left", "right", "down"]
        self.states = []  # record position and action of each episode
        self.pos = S
        self.exp_rate = exp_rate
        self.lr = lr
        self.sarsa = sarsa
        self.state_actions = {}
        for i in range(ROWS):
            for j in range(COLS):
                self.state_actions[(i, j)] = {}
                for a in self.actions:
                    self.state_actions[(i, j)][a] = 0

    def chooseAction(self):
        # epsilon-greedy
        mx_nxt_reward = -999
        action = ""
        if np.random.uniform(0, 1) <= self.exp_rate:
            action = np.random.choice(self.actions)
        else:
            # greedy action
            for a in self.actions:
                current_position = self.pos
                nxt_reward = self.state_actions[current_position][a]
                if nxt_reward >= mx_nxt_reward:
                    action = a
                    mx_nxt_reward = nxt_reward
        return action

    def reset(self):
        self.states = []
        self.cliff = Cliff()
        self.pos = S

    def play(self, rounds=10):
        for _ in range(rounds):
            while 1:
                curr_state = self.pos
                cur_reward = self.cliff.giveReward()
                action = self.chooseAction()
                # next position
                self.cliff.pos = self.cliff.nxtPosition(action)
                self.pos = self.cliff.pos
                self.states.append([curr_state, action, cur_reward])
                if self.cliff.end:
                    break
            # game end update estimates
            reward = self.cliff.giveReward()
            print("End game reward", reward)
            # reward of all actions in end state is same
            for a in self.actions:
                self.state_actions[self.pos][a] = reward
            if self.sarsa:
                for s in reversed(self.states):
                    pos, action, r = s[0], s[1], s[2]
                    current_value = self.state_actions[pos][action]
                    reward = current_value + self.lr * (r + reward - current_value)
                    self.state_actions[pos][action] = round(reward, 3)
            else:
                for s in reversed(self.states):
                    pos, action, r = s[0], s[1], s[2]
                    current_value = self.state_actions[pos][action]
                    reward = current_value + self.lr * (r + reward - current_value)
                    self.state_actions[pos][action] = round(reward, 3)
                    # update using the max value of S'
                    reward = np.max(list(self.state_actions[pos].values()))  # max
            self.reset()


def showRoute(states):
    board = np.zeros([4, 12])
    # add cliff marked as -1
    board[3, 1:11] = -1
    for i in range(0, ROWS):
        print('-------------------------------------------------')
        out = '| '
        for j in range(0, COLS):
            token = '0'
            if board[i, j] == -1:
                token = '*'
            if (i, j) in states:
                token = 'R'
            if (i, j) == G:
                token = 'G'
            out += token + ' | '
        print(out)
    print('-------------------------------------------------')
```

```
if __name__ == "__main__":
    print("sarsa training ... ")
    ag = Agent(exp_rate=0.1, sarsa=True)
    ag.play(rounds=500)
    # Sarsa
    ag_op = Agent(exp_rate=0)
    ag_op.state_actions = ag.state_actions
    states = []
    while 1:
        curr_state = ag_op.pos
        action = ag_op.chooseAction()
        states.append(curr_state)
        print("current position {} |action {}".format(curr_state, action))
        # next position
        ag_op.cliff.pos = ag_op.cliff.nxtPosition(action)
        ag_op.pos = ag_op.cliff.pos
        if ag_op.cliff.end:
            break
    showRoute(states)
    print("q-learning training ... ")
    ag = Agent(exp_rate=0.1, sarsa=False)
    ag.play(rounds=500)
    # Q-learning
    ag_op = Agent(exp_rate=0)
    ag_op.state_actions = ag.state_actions
    states = []
    while 1:
        curr_state = ag_op.pos
        action = ag_op.chooseAction()
        states.append(curr_state)
        print("current position {} |action {}".format(curr_state, action))
        # next position
        ag_op.cliff.pos = ag_op.cliff.nxtPosition(action)
        ag_op.pos = ag_op.cliff.pos
        if ag_op.cliff.end:
            break
    showRoute(states)
```

OUTPUT

```
current position (3, 0) |action up
current position (2, 0) |action right
current position (2, 1) |action right
current position (2, 2) |action right
current position (2, 3) |action right
current position (2, 4) |action right
current position (2, 5) |action right
current position (2, 6) |action right
current position (2, 7) |action right
current position (2, 8) |action right
current position (2, 9) |action right
current position (2, 10) |action right
current position (2, 11) |action down

Game ends reaching goal
```



```
-------------------------------------------------
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
-------------------------------------------------
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
-------------------------------------------------
| R | R | R | R | R | R | R | R | R | R | R | R |
-------------------------------------------------
| R | * | * | * | * | * | * | * | * | * | * | G |
-------------------------------------------------
```

RESULT

Thus, the implementation of the Off-Policy TD algorithm known as Q-learning for Cliff Walking is executed successfully and output is verified.

EX.NO: 8 **ON-POLICY TD CONTROL ALGORITHM FOR FROZEN LAKE**
DATE:

AIM

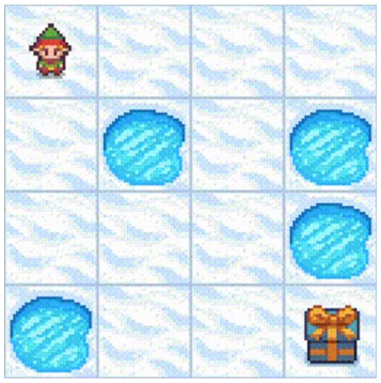
To implement On-Policy TD algorithm known as SARSA for Frozen Lake environment.

ON-POLICY TD CONTROL ALGORITHM: SARSA

State-action-reward-state-action (SARSA) is an on-policy reinforcement learning algorithm used to teach a new Markov decision process policy. It's an algorithm where, in the current state (S), an action (A) is taken and the agent gets a reward (R), and ends up in the next state (S_1), and takes action (A_1) in S_1. Therefore, the tuple (S, A, R, S_1, A_1) stands for the acronym SARSA. It's called an on-policy algorithm because it updates the policy based on actions taken.

PROBLEM STATEMENT

In frozen lake environment, the AI agent must cross the frozen lake from the start to the goal, without falling into the holes.


Frozen Lake Environment

The SARSA algorithm works by carrying out actions based on rewards received from previous actions. To do this, SARSA stores a table of state (S)-action (A) estimate pairs for each Q-value. This table is known as a Q-table, while the state-action pairs are denoted as Q(S, A). Exploitation involves using already known, estimated values to get more previously earned rewards in the learning process. Exploration involves attempting to find new knowledge on actions, which may result in short-term, sub-optimal actions during learning but may yield long-term benefits to find the best possible action and reward. From here, the selected action is taken, and the reward (R) and next state (S1) are observed. Q(S, A) is then updated, and the next action (A1) is selected based on the updated Q-values. Action-value estimates of a state are also updated for each current action-state pair present, which estimates the value of receiving a reward for taking a given action.

The above steps of R through A1 are repeated until the algorithm's given episode ends, which describes the sequence of states, actions and rewards taken until the final (terminal) state is reached. State, action and reward experiences in the SARSA process are used to update Q(S, A) values for each iteration.

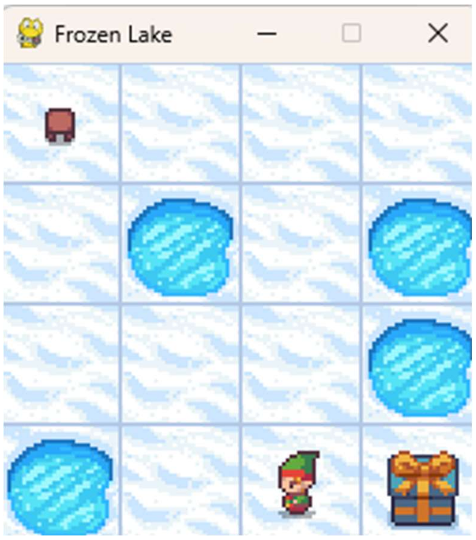
ALGORITHM

1. Initialize the environment and Q-table with zeros.
2. Train using the SARSA algorithm, updating Q-values via the state-action reward dynamics.
3. Evaluate the learned policy periodically to track average rewards.
4. Visualize the agent's performance by rendering its actions in the environment.
5. Analyze results to check if the target average reward was achieved and demonstrate success.

PROGRAM

```
import numpy as np
import gym
from tqdm import tqdm
import time
def epsilon_greedy_policy(Q, state, epsilon):
    if np.random.uniform(0, 1) < epsilon:
        return np.random.choice(len(Q[state]))
    else:
        return np.argmax(Q[state])
def sarsa(env, num_episodes, alpha=0.5, gamma=0.99, epsilon=0.5, eval_every=100):
    Q = np.zeros([env.observation_space.n, env.action_space.n])
    pbar = tqdm(total=num_episodes, dynamic_ncols=True)
    avg_rewards = []  # To track average rewards over episodes
    for episode in range(num_episodes):
        state, _ = env.reset()
        action = epsilon_greedy_policy(Q, state, epsilon)
        done = False
        episode_reward = 0
        while not done:
            next_state, reward, done, _, _ = env.step(action)
            next_action = epsilon_greedy_policy(Q, next_state, epsilon)
            td_target = reward + gamma * Q[next_state, next_action]
            td_error = td_target - Q[state, action]
            Q[state, action] += alpha * td_error
            state, action = next_state, next_action
            episode_reward += reward

        pbar.update(1)

        # Track average rewards
        if episode % eval_every == 0:
            avg_reward = evaluate_policy(env, Q, eval_every)
            avg_rewards.append(avg_reward)
        pbar.set_description(f"Average reward: {avg_reward:.2f}")

    pbar.close()
    return Q, avg_rewards  # Ensure both values are returned

def evaluate_policy(env, Q, num_episodes):
    total_reward = 0
    policy = np.argmax(Q, axis=1)
    for episode in range(num_episodes):
        observation, _ = env.reset()
        done = False
        episode_reward = 0
        while not done:
            action = policy[observation]
            observation, reward, done, _, _ = env.step(action)
            episode_reward += reward
        total_reward += episode_reward
    return total_reward / num_episodes
def demo_agent(env, Q, num_episodes=1):
    policy = np.argmax(Q, axis=1)
    for episode in range(num_episodes):
        observation, _ = env.reset()
        done = False
        print("\nEpisode:", episode + 1)
        while not done:
            env.render()
            action = policy[observation]
            observation, _,done, _, _ = env.step(action)
    env.render()
    env.close()
def main():
    env = gym.make("FrozenLake-v1")
    num_episodes = 10000

    Q_sarsa, avg_rewards = sarsa(env, num_episodes)

    # Find episodes required to reach optimal average reward
    target_avg_reward = 0.8  # Define your target average reward
    optimal_episode = next((i * 100 for i, reward in enumerate(avg_rewards) if reward >=
target_avg_reward), None)

    if optimal_episode is not None:
        print(f"Optimal average reward achieved after {optimal_episode} episodes.")
    else:
        print("Optimal average reward not achieved within the training episodes.")

    avg_reward = evaluate_policy(env, Q_sarsa, num_episodes)
    print(f"Average reward after SARSA: {avg_reward}")

    visual_env = gym.make('FrozenLake-v1', render_mode='human')
    demo_agent(visual_env, Q_sarsa, num_episodes)


if __name__ == '__main__':
    main()
```

OUTPUT:



RESULT:

Hence On - policy TD algorithm for Frozen Lake was successfully implemented.

EX.NO: 9 **ONLINE TABULAR TD ALGORITHM FOR CLIFFWALKING**
DATE:

AIM

To implement online tabular temporal difference algorithm for Cliff Walking.

TRUE ONLINE TEMPORAL DIFFERENCE ALGORITHM

1. **Initialize Q-Values:**
   Initialize the Q-values (action-values) for all state-action pairs arbitrarily, e.g., to zeros.
2. **Set Hyperparameters:**
   Define the learning rate (alpha), discount factor (gamma), and lambda (λ) for eligibility traces.
3. **Initialize Eligibility Traces:**
   Initialize eligibility traces for each state-action pair to zero.
4. **Repeat for each episode:**

a. Initialize the starting state (S).
b. Initialize eligibility traces to zero for the new episode.
c. Repeat for each time step within the episode:
    i. Choose an action (A) based on a policy (e.g., epsilon-greedy).
    ii. Take the action and observe the reward (R) and the new state (S').
    iii. Calculate the TD error (delta):
        - delta = R + gamma * Q(S', A) - Q(S, A)
    iv. Update the eligibility traces for the current state-action pair:
        - E(S, A) = gamma * lambda * E(S, A) + 1
    v. Update the Q-value for the current state-action pair using the True Online TD update
        rule:
        - Q(S, A) = Q(S, A) + alpha * delta * E(S, A)
    vi. Update the eligibility traces for all state-action pairs:
        - E(S, A) = gamma * lambda * E(S, A)
    vii. Set the current state to the new state (S = S').
d. Repeat until the episode ends.

```
Input: the policy $\pi$ to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in \mathcal{S}^+$)
Repeat (for each episode):
   Initialize $S$
   Repeat (for each step of episode):
      $A \leftarrow$ action given by $\pi$ for $S$
      Take action $A$; observe reward, $R$, and next state, $S'$
      $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
      $S \leftarrow S'$
   until $S$ is terminal
```

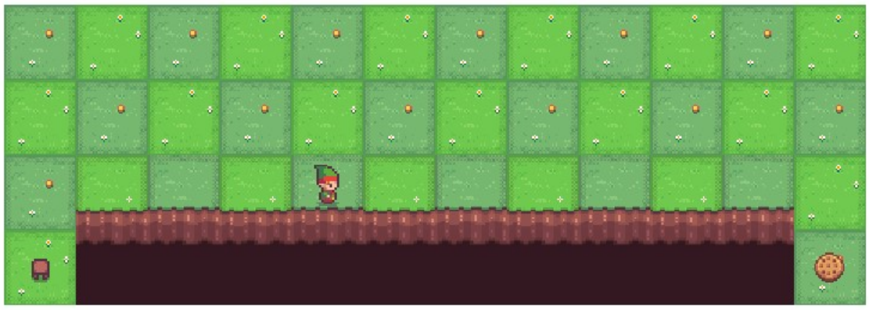
ALGORITHM:

1. Initialize Q-table, learning rate, discount factor, and exploration parameters.
2. For each episode, reset the environment and initialize total reward.
3. Select actions using ϵ\epsilonϵ-greedy policy, perform the action, and observe the next state, reward, and done flag.
4. Update the Q-value for the current state and action based on the reward and the maximum Q-value of the next state.
5. Accumulate the reward, move to the next state, and repeat until the episode ends.
6. Record the total reward, decay the exploration rate, and continue to the next episode.
7. After training, visualize the total rewards and Q-value heatmap.

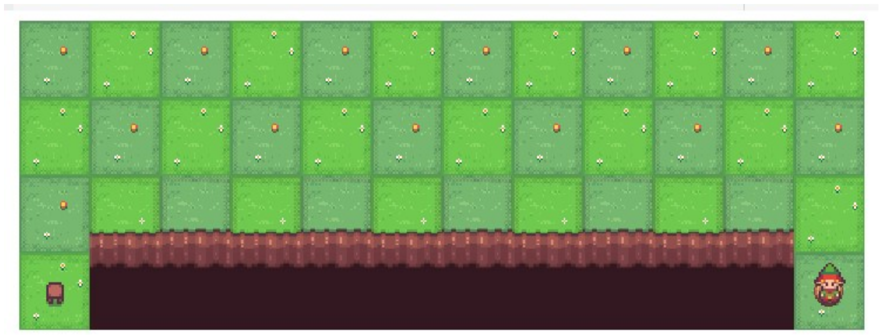
PROGRAM

```
import numpy as np
import gym
import pygame
import imageio
import random

class TDAgent:
    def __init__(self, n_states, n_actions, alpha=0.1, gamma=0.99, epsilon=0.1):
        self.n_states = n_states
        self.n_actions = n_actions
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.Q = np.zeros((n_states, n_actions))

    def choose_action(self, state):
        if np.random.uniform(0, 1) < self.epsilon:
            return np.random.choice(self.n_actions)
        else:
            return np.argmax(self.Q[state, :])

    def update(self, state, action, reward, next_state):
        predict = self.Q[state, action]
        target = reward + self.gamma * np.max(self.Q[next_state, :])
        self.Q[state, action] += self.alpha * (target - predict)

def train(agent, env, episodes, render=False):
    rewards = []
    for episode in range(episodes):
        state = env.reset()
        episode_reward = 0
        while True:
            action = agent.choose_action(state)
            next_state, reward, done, _ = env.step(action)
            agent.update(state, action, reward, next_state)
            state = next_state
            episode_reward += reward

            if done:
                break
        rewards.append(episode_reward)
    return rewards,agent

def record_video(agent, env, out_directory, fps=1):
  images = []
  done = False
  state = env.reset(seed=40)
  img = env.render(mode='rgb_array')
  images.append(img)
  while not done:
    # Take the action (index) that have the maximum expected future reward given that state
    action = agent.choose_action(state)
    state, reward, done, info = env.step(action) # We directly put next_state = state for
recording logic
    img = env.render(mode='rgb_array')
    if reward == -100:
      images = []
    images.append(img)
    wait_time_per_frame = 1000//fps
  imageio.mimsave(out_directory, [np.array(img) for i, img in enumerate(images)],
duration=wait_time_per_frame)

env = gym.make('CliffWalking-v0')
agent = TDAgent(env.observation_space.n, env.action_space.n)
episodes = 500
rewards,trainned_agent = train(agent, env, episodes, render=True)

video_path="/content/replay.gif"
video_fps=10
record_video(trainned_agent, env, video_path, video_fps)

from IPython.display import Image
Image('./replay.gif')
```

OUTPUT



INITIAL STATE



GOAL STATE

RESULT:

Thus, the implementation of online tabular temporal difference algorithm for Cliff Walking is executed successfully and output is verified.

EX.NO: 10 **RECYCLING ROBOT USING Q-LEARNING**

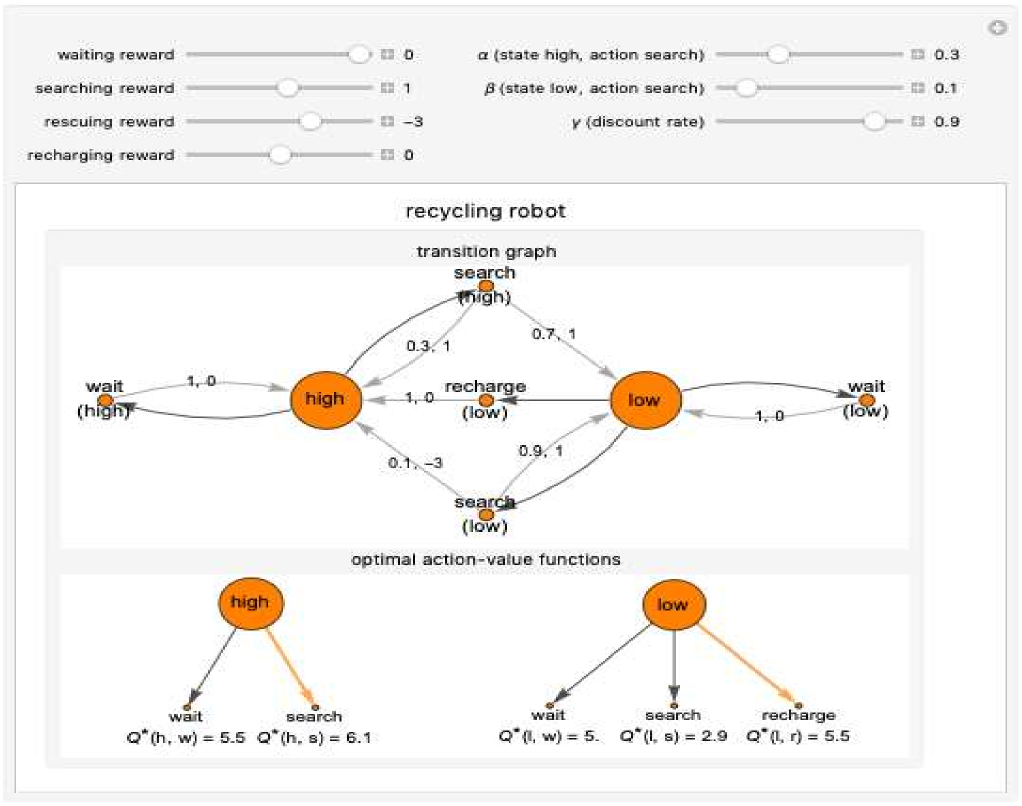
DATE:

AIM

To implement Q-Learning algorithm for Recycling Robot.

PROBLEM SATEMENT

A mobile robot with a rechargeable battery collects empty soda cans in an office area. In the transition graph, the state nodes represent the high and low energy levels. They are connected with black edges to two action nodes (search, wait) and to three action nodes (search, wait, recharge), respectively. On the edges of the graph, the transition probability and the reward for that transition are displayed. The separated diagrams show the action-value functions for the optimal policy, where the values are calculated based on the Bellman optimality equations. The optimal choice is highlighted in both states; in the case of multiple optimal policies in the given state, each of them is highlighted.



waiting reward 0
searching reward 1
rescuing reward −3
recharging reward 0
α (state high, action search) 0.3
β (state low, action search) 0.1
γ (discount rate) 0.9

Q-LEARNING ALGORITHM

Q-Learning is a model-free reinforcement learning algorithm used to find the optimal action-selection policy for an agent interacting with an environment. It is a type of temporal difference learning, which means it updates estimates based on other learned estimates, rather than waiting for the final outcome.

In Q-learning, an agent learns how to act optimally by learning the action-value function (Q-function), which provides a measure of the expected cumulative reward for each action taken in a given state.

```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
   Initialize $S$
   Repeat (for each step of episode):
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
      $S \leftarrow S'$;
   until $S$ is terminal
```

Algorithm:

1. START
2. Initialize the rewards, state space, and hyperparameters:
    a. State space S: The set of possible states (e.g., battery levels for the robot).
    b. Action space A: The set of possible actions (e.g., search, wait, recharge).
    c. Initialize the Q-table Q(s,a) with random values or zeros for all state-action pairs.
    d. Set hyperparameters: Learning rate α, discount factor γ, and exploration rate ϵ.
3. Define the reward function, that takes the current state s and action a as inputs and returns the reward r.
4. Define the state transition function, that takes the current state s and action a as inputs and returns next s′ state.
5. Implement Q-Learning Algorithm for given environment.
6. After completing all the episodes, display the plots and optimal policy.
7. STOP

PROGRAM

```
import numpy as np
import random

class RecyclingEnvironment:
    def __init__(self):
        self.energy_levels = ["high", "low"]
        self.actions = ["search", "wait", "recharge"]
        self.initial_energy_level = "high"
        self.battery_depletion_prob = 0.1

    def reset(self):
        self.current_energy_level = self.initial_energy_level
        return self.current_energy_level
    def step(self, action):
        # Energy level transitions
        if self.current_energy_level == "high":
            if action == "search":
                self.current_energy_level = "low"
                if np.random.uniform(0, 1) < self.battery_depletion_prob:
                    # Robot's battery is depleted while searching
                    self.current_energy_level = "low"
                    reward = -10  # Penalty for battery depletion
                else:
                    reward = random.randint(1, 10)  # Random reward for searching
            elif action == "recharge":
                self.current_energy_level = "high"
                reward = 0  # No reward for recharging
            else:
                reward = 0  # No reward for waiting
        else:
            if action == "recharge":
                self.current_energy_level = "high"
                reward = 0  # No reward for recharging
            else:
                reward = 0  # No action other than recharging is allowed when energy is low

        return self.current_energy_level, reward

class QLearningAgent:
        def __init__(self, states, actions, learning_rate=0.1, discount_factor=0.99,
exploration_prob=0.1):
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_prob = exploration_prob
        self.states = states
        self.actions = actions
        self.q_table = np.zeros((len(states), len(actions)))

    def choose_action(self, state):
        if np.random.uniform(0, 1) < self.exploration_prob:
            return np.random.choice(self.actions)
        else:
            state_idx = self.states.index(state)
            return self.actions[np.argmax(self.q_table[state_idx, :])]

    def update_q_table(self, state, action, reward, next_state):
        state_idx = self.states.index(state)
        next_state_idx = self.states.index(next_state)
        action_idx = self.actions.index(action)
        self.q_table[state_idx, action_idx] += self.learning_rate * \
                (reward + self.discount_factor * np.max(self.q_table[next_state_idx, :]) -
self.q_table[state_idx, action_idx])

# Hyperparameters
learning_rate = 0.1
discount_factor = 0.99
exploration_prob = 0.1
num_episodes = 1000

# Initialize environment and agent
env = RecyclingEnvironment()
states = env.energy_levels
actions = env.actions
agent = QLearningAgent(states, actions, learning_rate, discount_factor, exploration_prob)

# Training loop
for episode in range(num_episodes):
    state = env.reset()
    total_reward = 0

    while True:
        action = agent.choose_action(state)
        next_state, reward = env.step(action)
        agent.update_q_table(state, action, reward, next_state)
        total_reward += reward
        state = next_state

        if state == "low":
            break

    if episode % 100 == 0:
        print(f"Episode {episode}, Total Reward: {total_reward}")

# Evaluation
state = env.reset()
total_reward = 0

while state == "high":
    action = agent.choose_action(state)
    next_state, reward = env.step(action)
    total_reward += reward
    state = next_state

print(f"Evaluation - Total Reward: {total_reward}")
```

OUTPUT

```
Episode 0, Total Reward: 2
Episode 100, Total Reward: 1
Episode 200, Total Reward: 4
Episode 300, Total Reward: 4
Episode 400, Total Reward: 5
Episode 500, Total Reward: 5
Episode 600, Total Reward: 5
Episode 700, Total Reward: 4
Episode 800, Total Reward: 8
Episode 900, Total Reward: 5
Evaluation - Total Reward: 2
```

RESULT

Thus, the Recycling Robot problem has been implemented and executed succesfully using Q-Learning.