LINKS TO THE COLLAB FILE:

My Statement: Working on dividing a space into 16(4x4) grids with obstacle(or obstruction) at a particular location. Initialized a fixed starting and ending state. Defined 4 actions corresponding to each state; Left, Right, Bottom, Up. Specified reward values for transition from one state to another. Defined 16 random paths from start to end states. Updated the state-action value corresponding to each transition in an episode using SARSA algorithm. Later used Epsilon-Greedy policy, Q-Learning algorithm. Run the agent for many episodes from start to end state, keeping a record of all actions the agent has taken in each of the episode. Also traced the path the agent has taken in each episode.

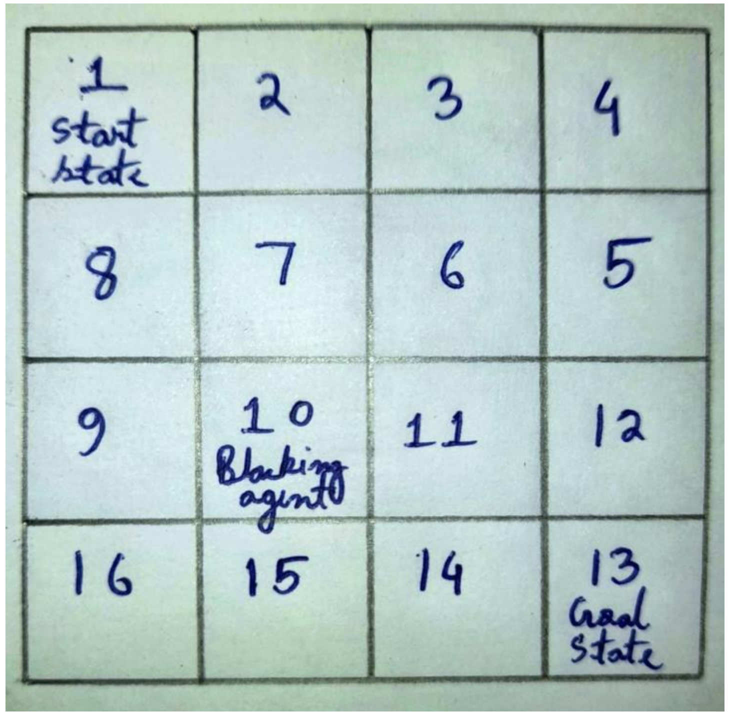
NOTE: In my presentation here, I have provided the screenshots for a particular execution of the cell. Hence all these values will vary if I run the program another time, though all the concepts used remain same.

File 1:(using SARSA algorithm)
https://colab.research.google.com/drive/1kd5_Av4B2Ldj-zwjmoZBjT22ehiHVHCj?usp=sharing

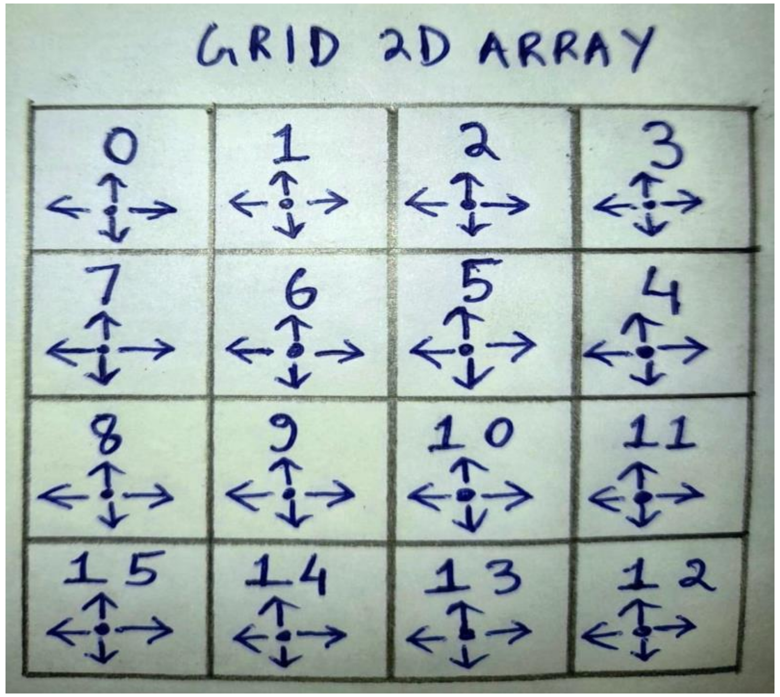
File 2:(using Q-Learning algorithm)
https://colab.research.google.com/drive/1pJVsVFXnYw6JgkhZ63fnXzIiQVmpwKpz?usp=sharing

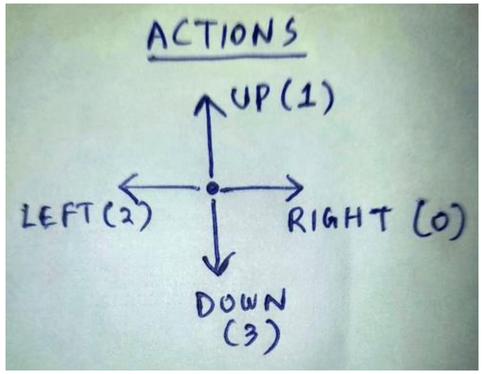
STEP 1:

- Created a (4 x 4) Gridworld with a blocking agent/obstruction at a particular location. Used a (4 x 4) matrix to denote the various states(16 states).
- Denoted the upper most left corner as the starting state('0') and bottomost right corner as the end state ('12'). Now for each state there are 4 possible actions; RIGHT(denoted by 0), UP(denoted by 1), LEFT(denoted by 2) and DOWN(denoted by 3)
- Placed an obstruction at the state 10(denoted as 9 in my notation). Assigned a reward of +1000 for transition into the goal state(denoted by '12'); -50 reward for transitioning into the state '10' and 0 reward for transitioning into all the other states.
- Created a (16 x 4) matrix named as 'Grid', where 16 denotes the number of states and 4 is no. of possible actions per state.



ACTUAL GRID-WORLD

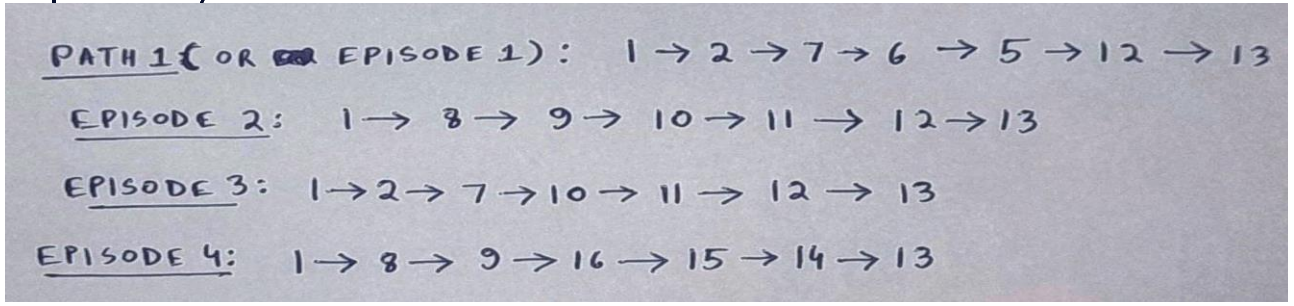




The actual Grid(2-D) Array. It is a (16 x 4) array with 16 states and 4 actions corresponding to each state.

STEP 2:

- Defined 16 different random episodes(here paths) from starting state to the end state. Some episode paths were through the block agent location(state 10 or ‘9’ according to my representation). One may define any such no. of episodes of his/her choice.
- Used a ‘Gamma’ value of 0.9 and learning rate (alpha) of 0.1. After end of a particular transition in each episode, updated the ‘Grid’ matrix using SARSA algorithm. Performed this update for each transition of all the 16 defined paths. The 16 paths(or episodes) I defined are as follows:

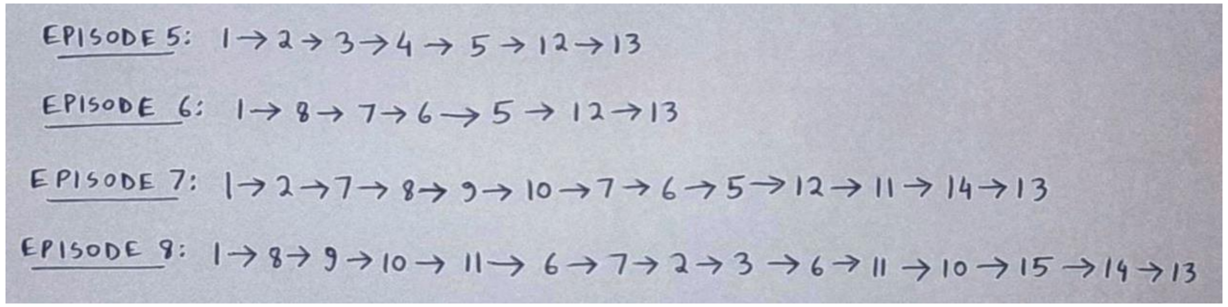


PATH 1(OR EPISODE 1): $1 \to 2 \to 7 \to 6 \to 5 \to 12 \to 13$

EPISODE 2: $1 \to 8 \to 9 \to 10 \to 11 \to 12 \to 13$

EPISODE 3: $1 \to 2 \to 7 \to 10 \to 11 \to 12 \to 13$

EPISODE 4: $1 \to 8 \to 9 \to 16 \to 15 \to 14 \to 13$

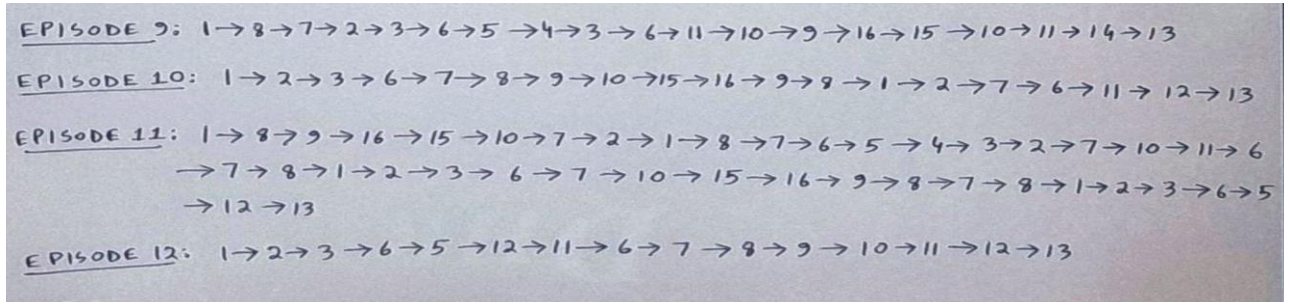


EPISODE 5: $1 \to 2 \to 3 \to 4 \to 5 \to 12 \to 13$

EPISODE 6: $1 \to 8 \to 7 \to 6 \to 5 \to 12 \to 13$

EPISODE 7: $1 \to 2 \to 7 \to 8 \to 9 \to 10 \to 7 \to 6 \to 5 \to 12 \to 11 \to 14 \to 13$

EPISODE 8: $1 \to 8 \to 9 \to 10 \to 11 \to 6 \to 7 \to 2 \to 3 \to 6 \to 11 \to 10 \to 15 \to 14 \to 13$

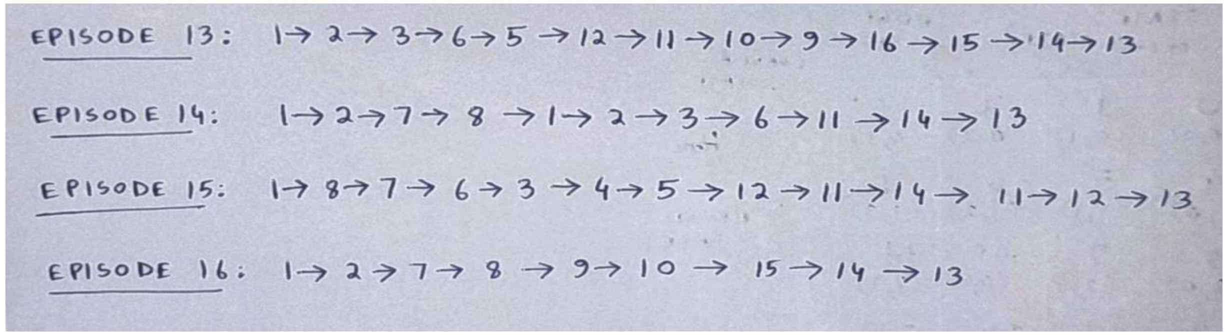


EPISODE 9: $1 \to 8 \to 7 \to 2 \to 3 \to 6 \to 5 \to 4 \to 3 \to 6 \to 11 \to 10 \to 9 \to 16 \to 15 \to 10 \to 11 \to 14 \to 13$

EPISODE 10: $1 \to 2 \to 3 \to 6 \to 7 \to 8 \to 9 \to 10 \to 15 \to 16 \to 9 \to 8 \to 1 \to 2 \to 7 \to 6 \to 11 \to 12 \to 13$

EPISODE 11: $1 \to 8 \to 9 \to 16 \to 15 \to 10 \to 7 \to 2 \to 1 \to 8 \to 7 \to 6 \to 5 \to 4 \to 3 \to 2 \to 7 \to 10 \to 11 \to 6$
$\to 7 \to 8 \to 1 \to 2 \to 3 \to 6 \to 7 \to 10 \to 15 \to 16 \to 9 \to 8 \to 7 \to 8 \to 1 \to 2 \to 3 \to 6 \to 5$
$\to 12 \to 13$

EPISODE 12: $1 \to 2 \to 3 \to 6 \to 5 \to 12 \to 11 \to 6 \to 7 \to 8 \to 9 \to 10 \to 11 \to 12 \to 13$



EPISODE 13: $1 \to 2 \to 3 \to 6 \to 5 \to 12 \to 11 \to 10 \to 9 \to 16 \to 15 \to 14 \to 13$

EPISODE 14: $1 \to 2 \to 7 \to 8 \to 1 \to 2 \to 3 \to 6 \to 11 \to 14 \to 13$

EPISODE 15: $1 \to 8 \to 7 \to 6 \to 3 \to 4 \to 5 \to 12 \to 11 \to 14 \to 11 \to 12 \to 13$

EPISODE 16: $1 \to 2 \to 7 \to 8 \to 9 \to 10 \to 15 \to 14 \to 13$

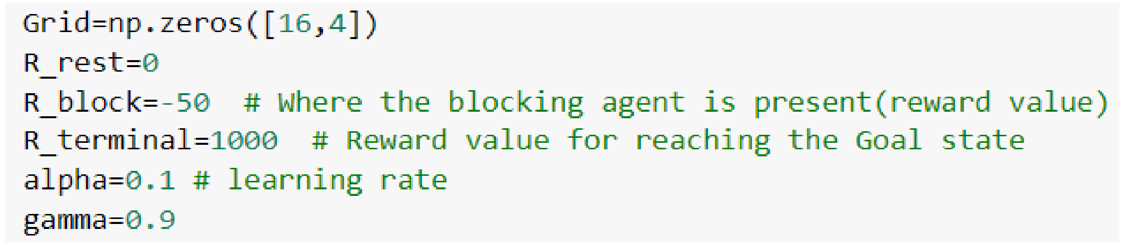
<u>The Updates will be done using SARSA algorithm as follows:</u>

Grid[state][action]=Grid[state][action]+alpha*(reward+(gamma*(Grid[new_state][action(new_state)])-Grid[state][action]) , where new_state -> state reached after transitioning from the old state after taking the particular action and action(new_state) is the action taken in new state.

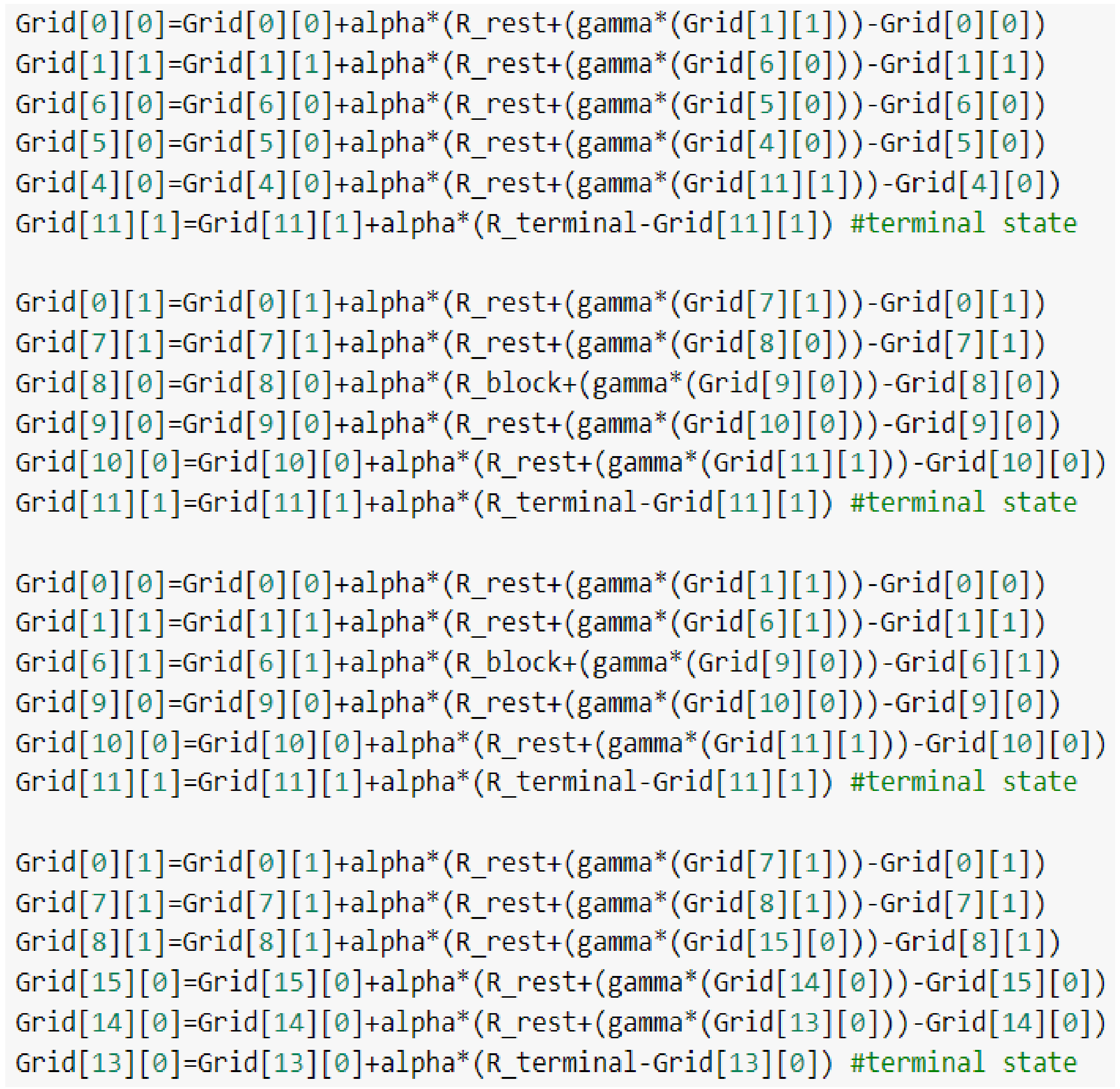
THE ORIGINAL FORM OF SARSA ALGORITHM IS:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \Big]$$

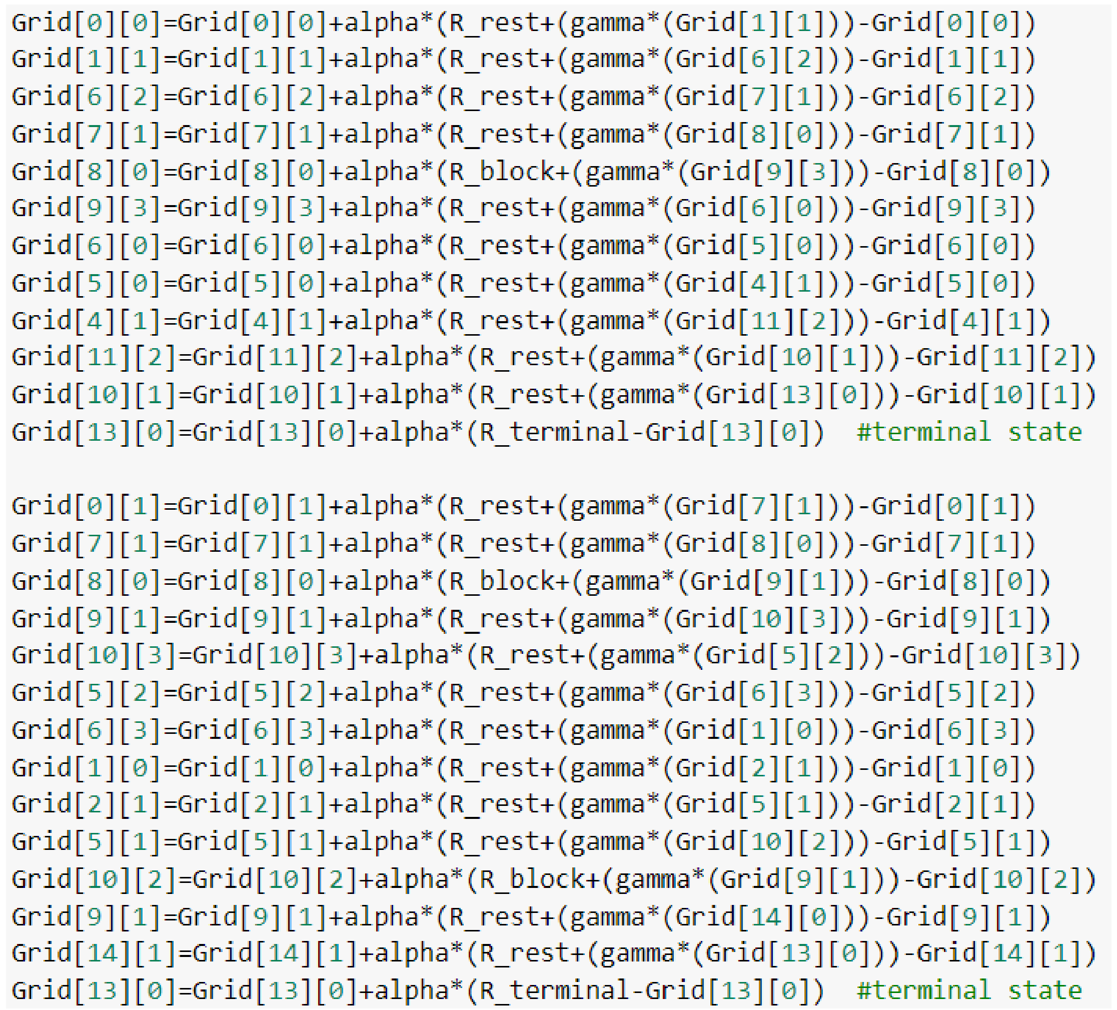
The Grid matrix update for each transition in each state looks like(in Python programming): I have shown it for 6 random paths from amongst the 16 paths defined.



```
Grid=np.zeros([16,4])
R_rest=0
R_block=-50  # Where the blocking agent is present(reward value)
R_terminal=1000  # Reward value for reaching the Goal state
alpha=0.1 # learning rate
gamma=0.9
```

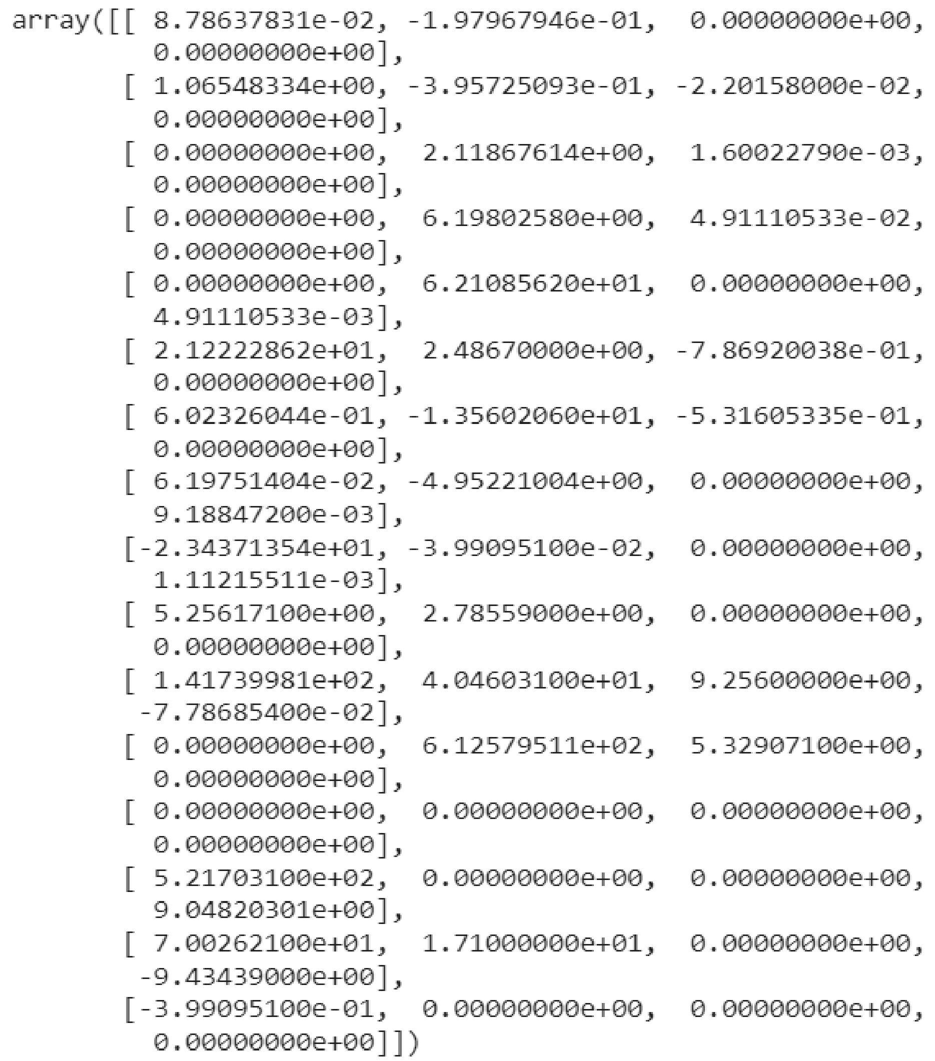


```
Grid[0][0]=Grid[0][0]+alpha*(R_rest+(gamma*(Grid[1][1]))-Grid[0][0])
Grid[1][1]=Grid[1][1]+alpha*(R_rest+(gamma*(Grid[6][0]))-Grid[1][1])
Grid[6][0]=Grid[6][0]+alpha*(R_rest+(gamma*(Grid[5][0]))-Grid[6][0])
Grid[5][0]=Grid[5][0]+alpha*(R_rest+(gamma*(Grid[4][0]))-Grid[5][0])
Grid[4][0]=Grid[4][0]+alpha*(R_rest+(gamma*(Grid[11][1]))-Grid[4][0])
Grid[11][1]=Grid[11][1]+alpha*(R_terminal-Grid[11][1]) #terminal state

Grid[0][1]=Grid[0][1]+alpha*(R_rest+(gamma*(Grid[7][1]))-Grid[0][1])
Grid[7][1]=Grid[7][1]+alpha*(R_rest+(gamma*(Grid[8][0]))-Grid[7][1])
Grid[8][0]=Grid[8][0]+alpha*(R_block+(gamma*(Grid[9][0]))-Grid[8][0])
Grid[9][0]=Grid[9][0]+alpha*(R_rest+(gamma*(Grid[10][0]))-Grid[9][0])
Grid[10][0]=Grid[10][0]+alpha*(R_rest+(gamma*(Grid[11][1]))-Grid[10][0])
Grid[11][1]=Grid[11][1]+alpha*(R_terminal-Grid[11][1]) #terminal state

Grid[0][0]=Grid[0][0]+alpha*(R_rest+(gamma*(Grid[1][1]))-Grid[0][0])
Grid[1][1]=Grid[1][1]+alpha*(R_rest+(gamma*(Grid[6][1]))-Grid[1][1])
Grid[6][1]=Grid[6][1]+alpha*(R_block+(gamma*(Grid[9][0]))-Grid[6][1])
Grid[9][0]=Grid[9][0]+alpha*(R_rest+(gamma*(Grid[10][0]))-Grid[9][0])
Grid[10][0]=Grid[10][0]+alpha*(R_rest+(gamma*(Grid[11][1]))-Grid[10][0])
Grid[11][1]=Grid[11][1]+alpha*(R_terminal-Grid[11][1]) #terminal state

Grid[0][1]=Grid[0][1]+alpha*(R_rest+(gamma*(Grid[7][1]))-Grid[0][1])
Grid[7][1]=Grid[7][1]+alpha*(R_rest+(gamma*(Grid[8][1]))-Grid[7][1])
Grid[8][1]=Grid[8][1]+alpha*(R_rest+(gamma*(Grid[15][0]))-Grid[8][1])
Grid[15][0]=Grid[15][0]+alpha*(R_rest+(gamma*(Grid[14][0]))-Grid[15][0])
Grid[14][0]=Grid[14][0]+alpha*(R_rest+(gamma*(Grid[13][0]))-Grid[14][0])
Grid[13][0]=Grid[13][0]+alpha*(R_terminal-Grid[13][0]) #terminal state
```



```
Grid[0][0]=Grid[0][0]+alpha*(R_rest+(gamma*(Grid[1][1]))-Grid[0][0])
Grid[1][1]=Grid[1][1]+alpha*(R_rest+(gamma*(Grid[6][2]))-Grid[1][1])
Grid[6][2]=Grid[6][2]+alpha*(R_rest+(gamma*(Grid[7][1]))-Grid[6][2])
Grid[7][1]=Grid[7][1]+alpha*(R_rest+(gamma*(Grid[8][0]))-Grid[7][1])
Grid[8][0]=Grid[8][0]+alpha*(R_block+(gamma*(Grid[9][3]))-Grid[8][0])
Grid[9][3]=Grid[9][3]+alpha*(R_rest+(gamma*(Grid[6][0]))-Grid[9][3])
Grid[6][0]=Grid[6][0]+alpha*(R_rest+(gamma*(Grid[5][0]))-Grid[6][0])
Grid[5][0]=Grid[5][0]+alpha*(R_rest+(gamma*(Grid[4][1]))-Grid[5][0])
Grid[4][1]=Grid[4][1]+alpha*(R_rest+(gamma*(Grid[11][2]))-Grid[4][1])
Grid[11][2]=Grid[11][2]+alpha*(R_rest+(gamma*(Grid[10][1]))-Grid[11][2])
Grid[10][1]=Grid[10][1]+alpha*(R_rest+(gamma*(Grid[13][0]))-Grid[10][1])
Grid[13][0]=Grid[13][0]+alpha*(R_terminal-Grid[13][0])  #terminal state

Grid[0][1]=Grid[0][1]+alpha*(R_rest+(gamma*(Grid[7][1]))-Grid[0][1])
Grid[7][1]=Grid[7][1]+alpha*(R_rest+(gamma*(Grid[8][0]))-Grid[7][1])
Grid[8][0]=Grid[8][0]+alpha*(R_block+(gamma*(Grid[9][1]))-Grid[8][0])
Grid[9][1]=Grid[9][1]+alpha*(R_rest+(gamma*(Grid[10][3]))-Grid[9][1])
Grid[10][3]=Grid[10][3]+alpha*(R_rest+(gamma*(Grid[5][2]))-Grid[10][3])
Grid[5][2]=Grid[5][2]+alpha*(R_rest+(gamma*(Grid[6][3]))-Grid[5][2])
Grid[6][3]=Grid[6][3]+alpha*(R_rest+(gamma*(Grid[1][0]))-Grid[6][3])
Grid[1][0]=Grid[1][0]+alpha*(R_rest+(gamma*(Grid[2][1]))-Grid[1][0])
Grid[2][1]=Grid[2][1]+alpha*(R_rest+(gamma*(Grid[5][1]))-Grid[2][1])
Grid[5][1]=Grid[5][1]+alpha*(R_rest+(gamma*(Grid[10][2]))-Grid[5][1])
Grid[10][2]=Grid[10][2]+alpha*(R_block+(gamma*(Grid[9][1]))-Grid[10][2])
Grid[9][1]=Grid[9][1]+alpha*(R_rest+(gamma*(Grid[14][0]))-Grid[9][1])
Grid[14][1]=Grid[14][1]+alpha*(R_rest+(gamma*(Grid[13][0]))-Grid[14][1])
Grid[13][0]=Grid[13][0]+alpha*(R_terminal-Grid[13][0])  #terminal state
```

After performing update for all the 16 paths(or episodes), the Grid matrix will look like as follows. For all the 16 states and for each action in every state, the value of each element of Grid matrix i.e, each of Grid[state][action] will get updated.

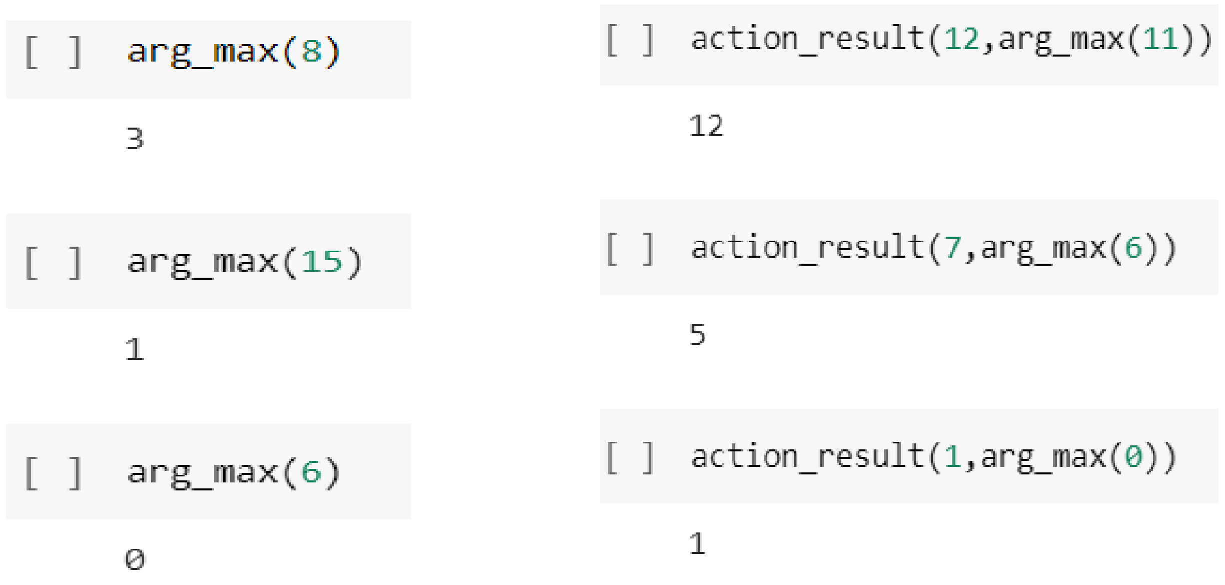


```
array([[ 8.78637831e-02, -1.97967946e-01,  0.00000000e+00,
         0.00000000e+00],
       [ 1.06548334e+00, -3.95725093e-01, -2.20158000e-02,
         0.00000000e+00],
       [ 0.00000000e+00,  2.11867614e+00,  1.60022790e-03,
         0.00000000e+00],
       [ 0.00000000e+00,  6.19802580e+00,  4.91110533e-02,
         0.00000000e+00],
       [ 0.00000000e+00,  6.21085620e+01,  0.00000000e+00,
         4.91110533e-03],
       [ 2.12222862e+01,  2.48670000e+00, -7.86920038e-01,
         0.00000000e+00],
       [ 6.02326044e-01, -1.35602060e+01, -5.31605335e-01,
         0.00000000e+00],
       [ 6.19751404e-02, -4.95221004e+00,  0.00000000e+00,
         9.18847200e-03],
       [-2.34371354e+01, -3.99095100e-02,  0.00000000e+00,
         1.11215511e-03],
       [ 5.25617100e+00,  2.78559000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 1.41739981e+02,  4.04603100e+01,  9.25600000e+00,
        -7.78685400e-02],
       [ 0.00000000e+00,  6.12579511e+02,  5.32907100e+00,
         0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 5.21703100e+02,  0.00000000e+00,  0.00000000e+00,
         9.04820301e+00],
       [ 7.00262100e+01,  1.71000000e+01,  0.00000000e+00,
        -9.43439000e+00],
       [-3.99095100e-01,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00]])
```

<u>STEP 3:</u>

- I have defined function for the state change(or the new state) the agent reaches for a particular action taken in a particular state.("action_result" function)

- I have also defined a function for taking the optimal action from a particular state(making the system ready for applying Q-Learning Algorithm)("arg_max" function)



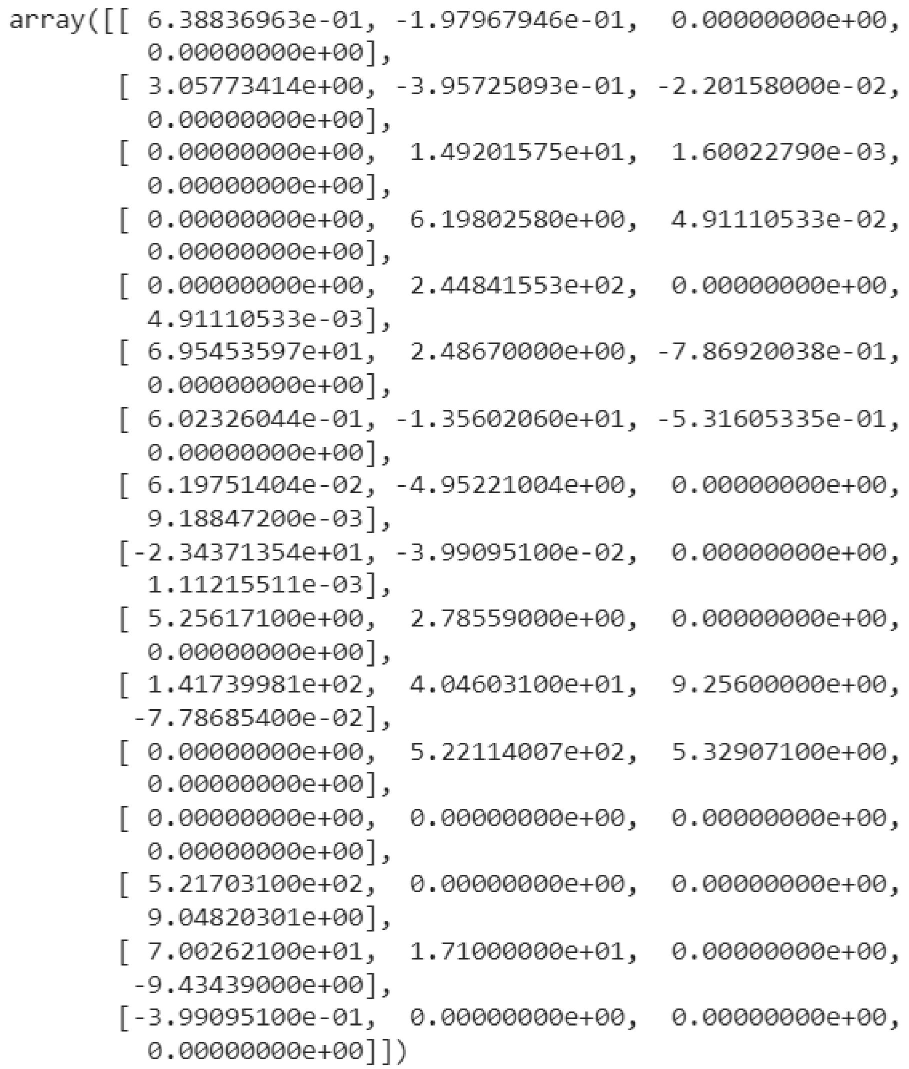
```
[ ]  arg_max(8)

     3

[ ]  arg_max(15)

     1

[ ]  arg_max(6)

     0
```

```
[ ]  action_result(12,arg_max(11))

     12

[ ]  action_result(7,arg_max(6))

     5

[ ]  action_result(1,arg_max(0))

     1
```

STEP 4:

• Finally, applied Q-Learning algorithm for 5 episodes between starting and end states. One may take any such number of episodes. Performed update of the “Grid” matrix for every state transition of each of the 5 episodes I defined using Q-Learning algorithm.

Finally, after performing the Q-Learning updates, the “Grid” Matrix will look like.



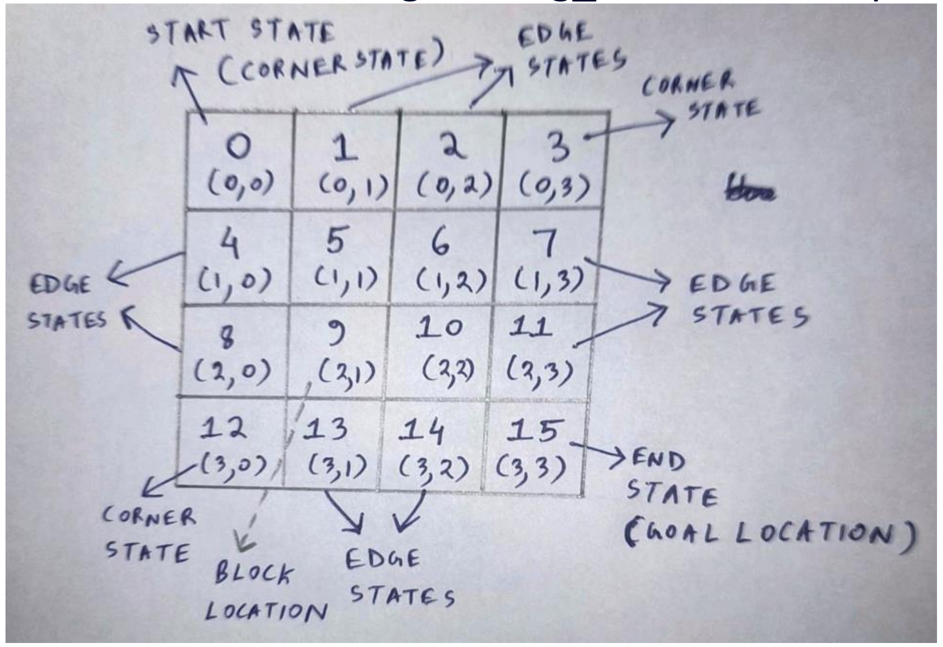
```
array([[ 6.38836963e-01, -1.97967946e-01,  0.00000000e+00,
         0.00000000e+00],
       [ 3.05773414e+00, -3.95725093e-01, -2.20158000e-02,
         0.00000000e+00],
       [ 0.00000000e+00,  1.49201575e+01,  1.60022790e-03,
         0.00000000e+00],
       [ 0.00000000e+00,  6.19802580e+00,  4.91110533e-02,
         0.00000000e+00],
       [ 0.00000000e+00,  2.44841553e+02,  0.00000000e+00,
         4.91110533e-03],
       [ 6.95453597e+01,  2.48670000e+00, -7.86920038e-01,
         0.00000000e+00],
       [ 6.02326044e-01, -1.35602060e+01, -5.31605335e-01,
         0.00000000e+00],
       [ 6.19751404e-02, -4.95221004e+00,  0.00000000e+00,
         9.18847200e-03],
       [-2.34371354e+01, -3.99095100e-02,  0.00000000e+00,
         1.11215511e-03],
       [ 5.25617100e+00,  2.78559000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 1.41739981e+02,  4.04603100e+01,  9.25600000e+00,
        -7.78685400e-02],
       [ 0.00000000e+00,  5.22114007e+02,  5.32907100e+00,
         0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00],
       [ 5.21703100e+02,  0.00000000e+00,  0.00000000e+00,
         9.04820301e+00],
       [ 7.00262100e+01,  1.71000000e+01,  0.00000000e+00,
        -9.43439000e+00],
       [-3.99095100e-01,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00]])
```

STEP 5:

- As the vehicle/process has to be autonomous, the use of SARSA algorithm here is not suitable. So I have created a totally new system which is totally based on use of Epsilon-Greedy policy and Q-Learning Algorithm. I have created a (4 x 4) reward matrix and assigned the reward for transitioning into the goal_location ((3,3) or state 15) as +500, block_location ((2,1) or state 9) as -80 , for all other state transition -1 reward.I represented the starting state by ‘0’. I have taken care of the “corner states” [there are 3 such states namely, state 0(or coordinate(0,0) acc. to my representation), state 3(or (0,3)), state 12(or (3,0))] and the eight “edge states”[namely 1,2,7,11,13,14,4,8 or coordinates (0,1), (0,2), (1,3), (2,3), (3,1), (3,2), (1,0), (2,0) respectively]. I have allowed restricted movements of all these states(2 movements for the corner states and 3 movements for the edge states). I have set the epsilon value as 0.6, gamma value as 0.9 and alpha(learning rate) value as 0.1. I numbered ‘UP’ action as 0, ‘DOWN’ action as 1, ‘LEFT’ action as 2 and ‘RIGHT’ action as 3. Finally created an action_matrix of dimensions (16x4), corresponding to the 16 states and 4 actions in each state, initializing all of its elements to 0. I have used the concept of Uniform Probabilty distribution [namely Uniform(0,1)] to generate numbers and if the number so generated is less than our defined epsilon value, the agent will take any random action from the set of action specified for each state. Else the action will be taken using the arg_max function/policy which forms the main basis of Q-Learning algorithm.

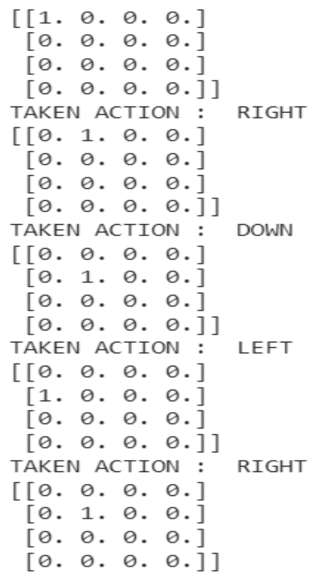
Q-Learning Algorithm:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big]$$

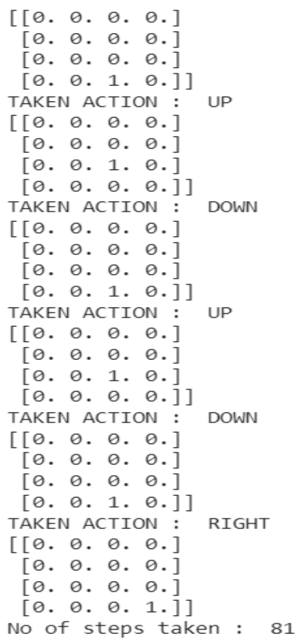


STEP 6:

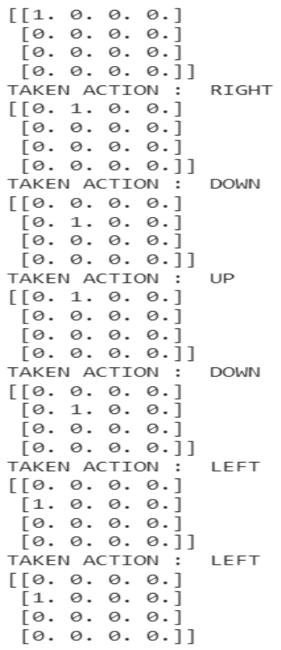
Next, I defined the arg_max criteria in the form of a function named as “choose action”, which mainly decides between the Exploration-exploitation policy and in case the values of to two or more highest valued actions corresponding to a particular state have the same value, then the tie will be broken by randomly choosing any one among the set of actions (or in my representation if two or more greatest valued column elements for a particular row of the action_matrix are same, then the tie will be broken by randomly selecting any one of the column element). Next I have initialized the agent at the starting point (state) (i.e (0,0)) and then using Q-Learning Algorithm instructed the agent to generate paths(or episodes) to the Goal state (i.e (3,3)). I have changed the action_matrix elements in each move of every episode. Also to visualize the generated path and trace the agent’s movement , I have used a matrices containing 0’s and 1(to denote the position in the space where the agent currently is). I displayed the action taken for each move I have also kept a count of the number of steps the agent takes to reach the “Goal state” starting from the “Start state”. This number of steps taken does not follow any regular pattern. For certain episodes, it decreases to a large extent and for certain episodes it shows a sudden increase as compared to the previous episode (or path). In other words the curve of no. of steps per episode vs Episode number is coming out to be “wavy” in nature showing peaks and valleys.



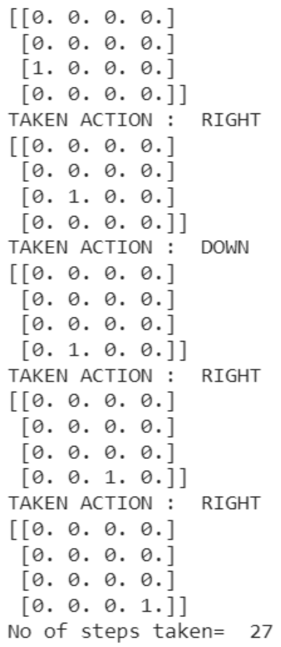
```
[[1. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   RIGHT
[[0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   DOWN
[[0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   LEFT
[[0. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   RIGHT
[[0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```



```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]]
TAKEN ACTION :   UP
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   DOWN
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]]
TAKEN ACTION :   UP
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   DOWN
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]]
TAKEN ACTION :   RIGHT
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 1.]]
No of steps taken :  81
```



```
[[1. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   RIGHT
[[0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   DOWN
[[0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   UP
[[0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   DOWN
[[0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   LEFT
[[0. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   LEFT
[[0. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

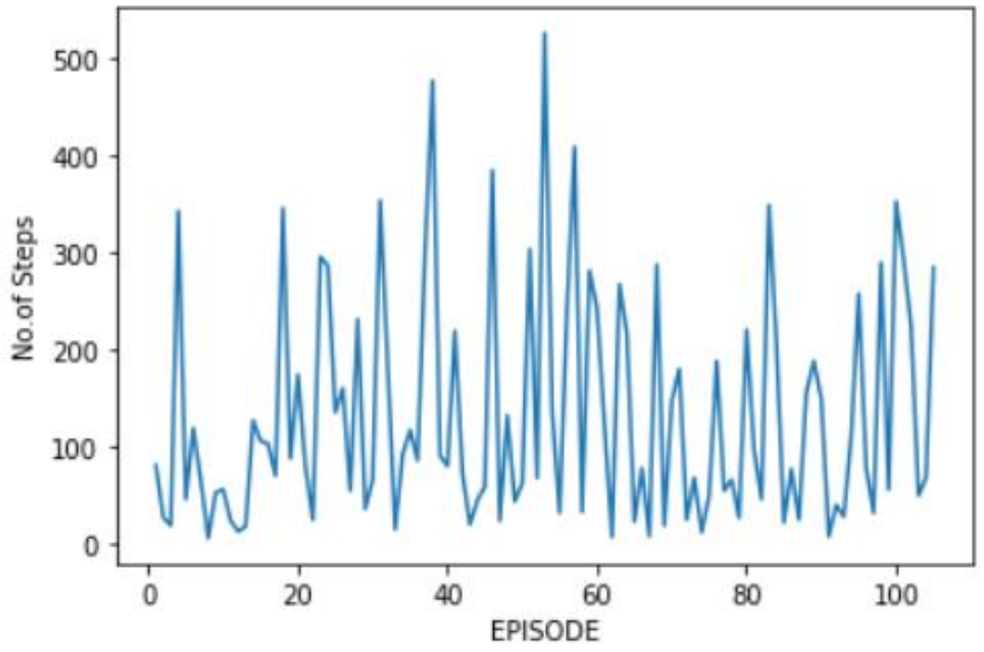


```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   RIGHT
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :   DOWN
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 1. 0. 0.]]
TAKEN ACTION :   RIGHT
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]]
TAKEN ACTION :   RIGHT
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 1.]]
No of steps taken=  27
```

IN THE VERY FIRST EPISODE, FOR INSTANCE, THE AGENT TAKES 81 EPISODES AND IN THE VERY NEXT EPISODE, THE AGENT TAKES 27 STEPS. SNIPPETS OF SOME INSTANCES FROM THE PATH TRACED ARE SHOWN IN THE LEFT HAND SIDE FIGURE.

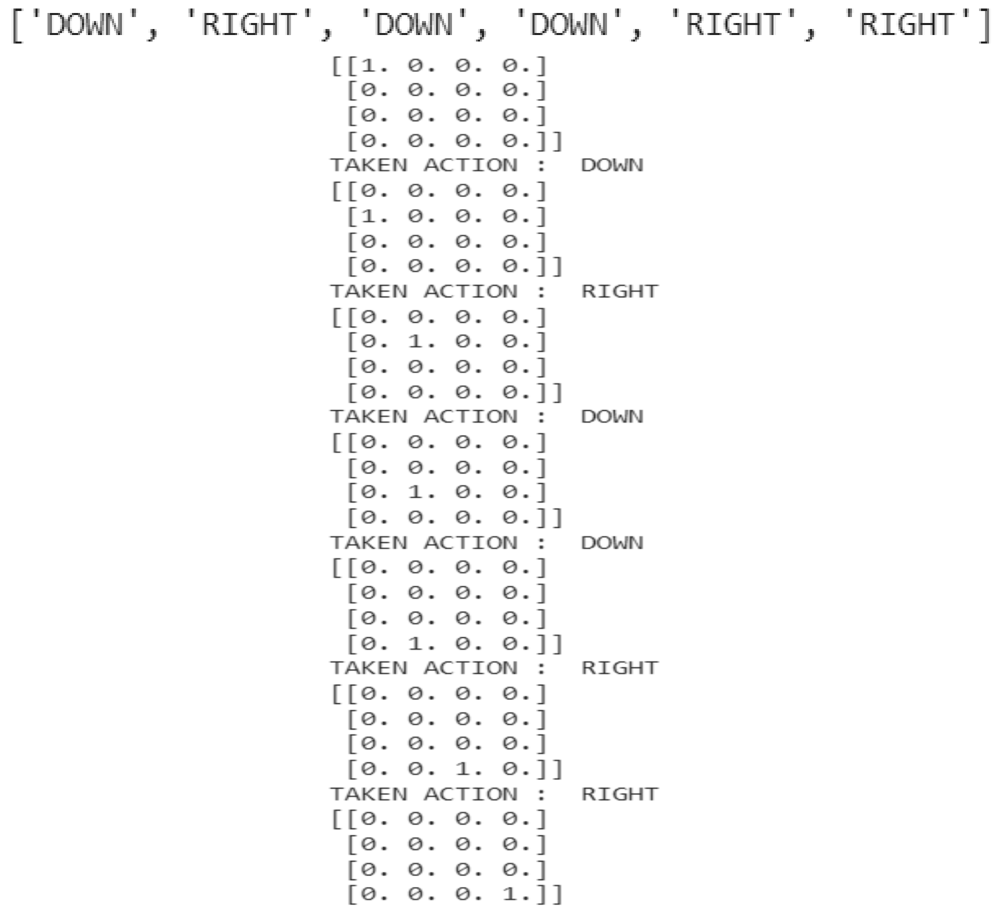
STEP 7:

I observed the trend of the graph for about 105 episodes. For each of the episode, I stored the number of actions taken in a list “period list” and the action sequence in a dictionary “action_set i”, where i denotes the ith episode. For recording the sequence of actions from minimum to maximum, I sorted the list “period list” containing the number of actions from minimum to maximum. After noting the episode in which the agent took the minimum no. of steps, I retrieved the action sequence for that particular episode from the ‘start state’ to the ‘end state’ and also traced the path the agent takes. I did the same for the first 6 least numbered steps episodes, retrieving and tracing the agent’s path in each of the cases.



THE ABOVE CURVE SHOWS THE VARIATION OF THE NO. OF STEPS PER EPISODE VS THE EPISODE NUMBER. THE MAXIMUM NO. OF STEPS WILL NOT REMAIN CONSTANT; RATHER IT WOULD GIVE A DIFFERENT VALUE FOR EVERY SINGLE EXECUTION. THE SAME APPLIES FOR THE MINIMUM NO.OF STEPS AND RATHER EACH AND EVERY EPISODE. THIS NO. OF STEPS WILL ALSO CHANGE FOR DIFFERENT VALUES EPSILON, ALPHA AND GAMMA

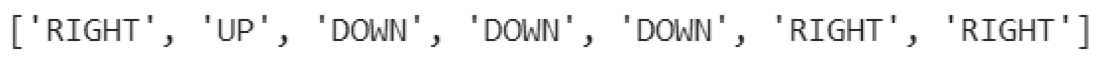
MINIMUM NO.OF STEPS TO REACH THE “END STATE” FROM “START STATE” IS 6. THE CORRESPONDING ACTIONS TAKEN ARE AS FOLLOWS:



```
['DOWN', 'RIGHT', 'DOWN', 'DOWN', 'RIGHT', 'RIGHT']
```

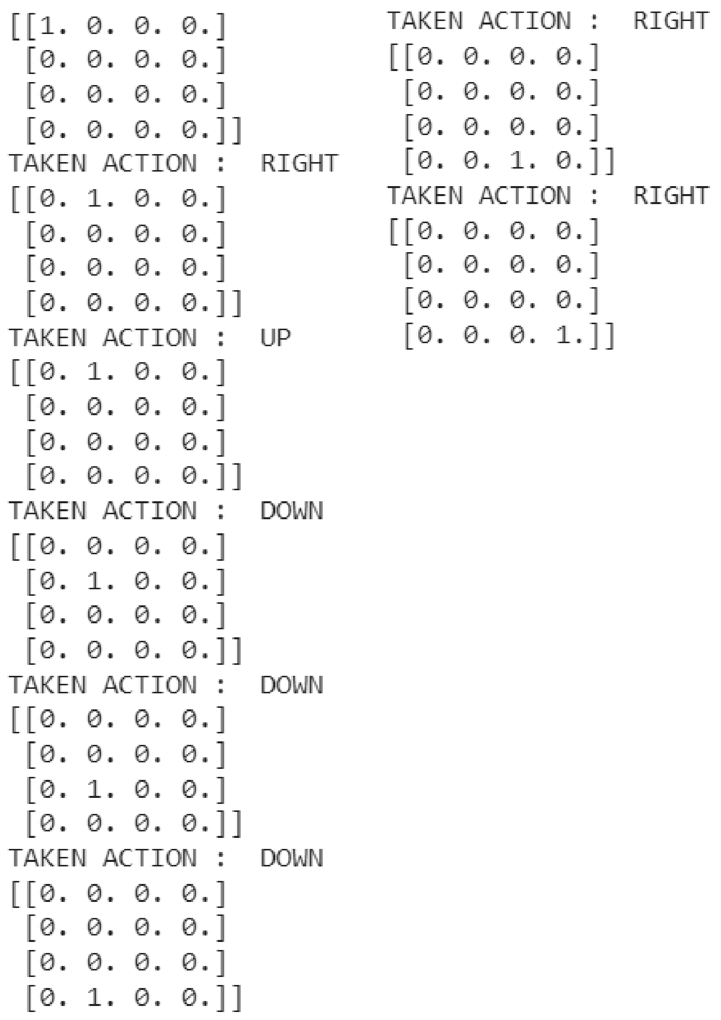
```
[[1. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :  DOWN
[[0. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :  RIGHT
[[0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :  DOWN
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :  DOWN
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 1. 0. 0.]]
TAKEN ACTION :  RIGHT
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]]
TAKEN ACTION :  RIGHT
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 1.]]
```

THE 2ND LEAST NO. OF STEPS THE AGENT TAKES TO REACH THE "END STATE" FROM "START STATE" IS 7. THE SEQUENCE OF ACTIONS TAKEN IS GIVEN BELOW:



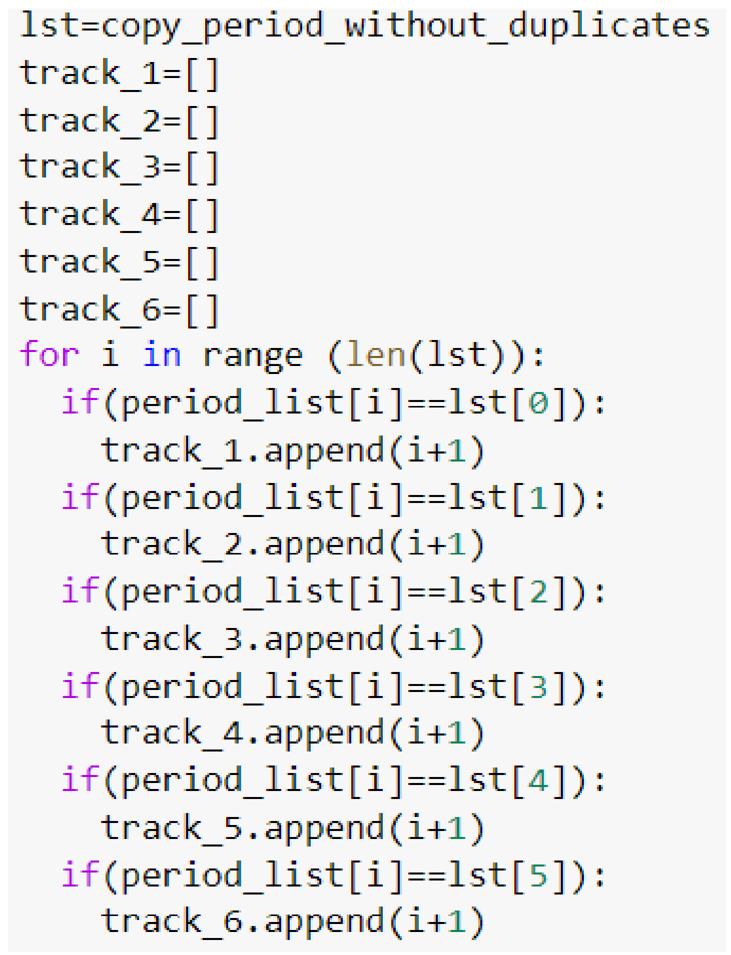
```
['RIGHT', 'UP', 'DOWN', 'DOWN', 'DOWN', 'RIGHT', 'RIGHT']
```

THE PATH FOLLOWED BY THE AGENT IS AS GIVEN BELOW(WHEN NO . OF STEPS TAKEN IS 7)



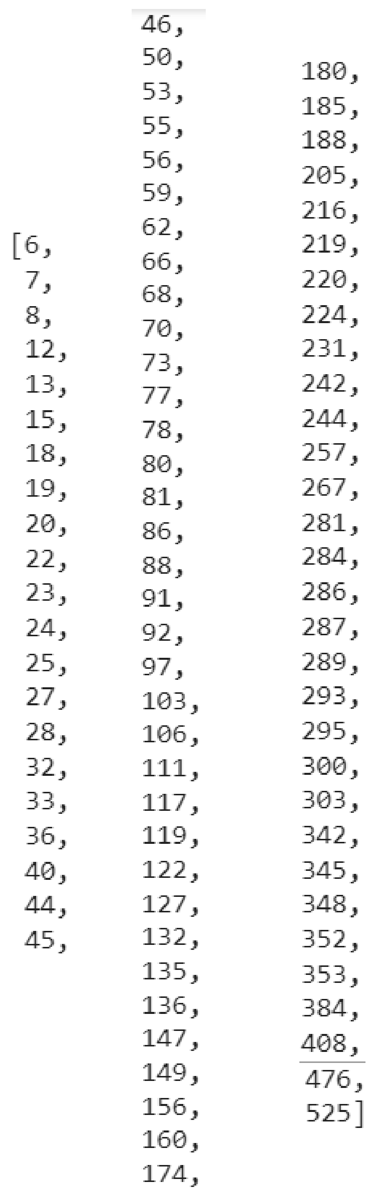
```
[[1. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :  RIGHT
[[0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :  UP
[[0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :  DOWN
[[0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :  DOWN
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]]
TAKEN ACTION :  DOWN
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 1. 0. 0.]]
TAKEN ACTION :  RIGHT
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 0.]]
TAKEN ACTION :  RIGHT
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 1.]]
```

I STORED THE EPISODE NUMBERS FOR THE 1ST 6 LEAST NUMBERED STEPS ACTIONS. DUE TO WHICH I AM ABLE TO RETRIEVE THE ACTION SEQUENCE AND HENCE TRACE THE PATH FROM "START" TO "END" STATE FOR ALL THE SIX SET OF ACTIONS TAKEN



```python
lst=copy_period_without_duplicates
track_1=[]
track_2=[]
track_3=[]
track_4=[]
track_5=[]
track_6=[]
for i in range (len(lst)):
  if(period_list[i]==lst[0]):
    track_1.append(i+1)
  if(period_list[i]==lst[1]):
    track_2.append(i+1)
  if(period_list[i]==lst[2]):
    track_3.append(i+1)
  if(period_list[i]==lst[3]):
    track_4.append(i+1)
  if(period_list[i]==lst[4]):
    track_5.append(i+1)
  if(period_list[i]==lst[5]):
    track_6.append(i+1)
```

THE NUMBER OF STEPS TAKEN FOR ALL THE 105 EPISODES GENERATED. THE ACTION SEQUENCE CAN BE GENERATED AND PATH CAN BE TRACED BY RETRIEVING ANY OF THE EPISODE FROM AMONGST THIS LIST.



```
[6,
 7,
 8,
 12,
 13,
 15,
 18,
 19,
 20,
 22,
 23,
 24,
 25,
 27,
 28,
 32,
 33,
 36,
 40,
 44,
 45,
 46,
 50,
 53,
 55,
 56,
 59,
 62,
 66,
 68,
 70,
 73,
 77,
 78,
 80,
 81,
 86,
 88,
 91,
 92,
 97,
 103,
 106,
 111,
 117,
 119,
 122,
 127,
 132,
 135,
 136,
 147,
 149,
 156,
 160,
 174,
 180,
 185,
 188,
 205,
 216,
 219,
 220,
 224,
 231,
 242,
 244,
 257,
 267,
 281,
 284,
 286,
 287,
 289,
 293,
 295,
 300,
 303,
 342,
 345,
 348,
 352,
 353,
 384,
 408,
 476,
 525]
```