Exp: 14 Implementation of clustering Technique K-means

Aim: To implement a k-mean clustering technique using python

Explanation:

1. Import K-mean from sklearn
2. Assign x and Y
3. call the function K-means()

Code:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.dataset - Samples - generator import
pyplot as plt
~~from sklearn.datasets - samples~~
x, y = make_blobs (n_Samples=300, centers=4, cluster_std
        =0.60, random_State=0)
plt.scatter(x[:,0], x[:,1])
wcss = []
for i in range(1,11):
K-means = K-means(n_clusters=i, init='k-means'
    max_iter=300, n_init=10, random_State=0)
K.means.fit(x)
wcss.append(k-mean.inertia_)
plt.plot(range(1,11),wcss)
plt.title('Elbow method')
plt.xlabel('Number of cluster')
plt.Ylabel('wcss')
plt.show()
kmeans = KMeans(n_cluster=4, init="k-mean++",
    max_iter=300, n_init=10, random_State=0)
pred_y = kmeans.fit_predict(x)
```

Exp: 13 Implementation of Decision Tree Classification Techniques

Aim: To implement decision tree classification technique for gender classification using python.

Code:

```
from sklearn.tree import
clf = tree.DecisionTreeClassifier()
X = [[181,180,91],[182,90,92],[183,100,92],[184,200,93],[185,300,94],
[186,400,95],[187,500,96],[189,600,97],[190,700,98],[191,800,99],
[192,900,100],[193,1000,101]]
Y = ['male','male','female','male','female','male','female','male',
'female','male','female','male']
clf = clf.fit(X,Y)
prediction f = clf.predict([[181,180,91]]
prediction m = clf.predict([[183,100,92]]
print(prediction f)
print(prediction m)
```

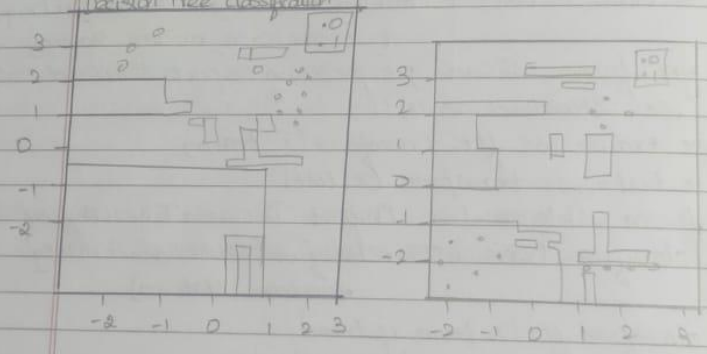
Output:

```
['male']
['female']
```

Result: The program is successfully executed

```
x_set[:,0].min()-1, step=0.01)
ystop=x_set[:,0].max()+1, step=0.01))
plt.contourf(x1,x2,classifier.predict(np.array
                [x1.ravel(),x2.ravel()].T).T)
        .reshape(x1.shape),alpha=0.75,cmap=
ListedColormap(('red','green')))
plt.xlim(x1.min(),x1.max())
plt.ylim(x2.min(),x2.max())
for i,j in enumerate(np.unique(y_set)):
  plt.scatter(x_set[y_set==j,0],x_set[y_set==j,1],
c=ListedColormap(('red','green'))(i),label=j)
plt.title('Decision Tree Classification')
plt.xlabel('Age')
plt.ylabel('Purchase')
plt.legend()
plt.show()
```

Decision Tree Classification



Result: The program is successfully executed.

Exp:12 Decision tree classification

Aim:- To classify the social Network data using Decision tree analysis.

Code:

```
from google.colab import drive
drive.mount("/content/gdrive")
import pandas as pd
import numpy as np
import matplotlib.pylot as plt
dataset = pd.read_csv('/content/gdrive/MyDrive/Social_
                                Network_Ads.csv')

x = dataset.iloc[:,[2,3]].values.
y = dataset.iloc[:,-1].Values
from sklearn.model_selection import train-test-split
x_train x_test, y_train, y_test = train_test_split(x,y
                                test_size=0.25, random_state=0)
you from sklearn.preprocessing import standard scal
se = standard scales()
x_train = se.fit_transform(x_train)
x_test = se.transform (x_test)
from sklearn.tree import Decision Treeclassifier
classifier = Decision Treeclassifier (criterion = 'entropy'
                                random_state = 0)
classifier.fit (x_train, y_train)
y_pred = classifier.predict (x_test)
from sklearn.metrics import confusion_matrix
cm = confusion_matrix (y_test, y_pred)
print (cm)
from matplotlib.colors import Listed colormap
x_set, y_set = x_train, y_train
x1,x2 = np.meshgrid(np.arrange (start = x_set[:,0].min(
1, stop = x_set[:,0].max()+1, step = 0.01).np arrange(s
```

Exp. 11 Implementing Artificial neural network for an application using Python - Regression

Aim: To Implementing artifical neural network for an application using python.

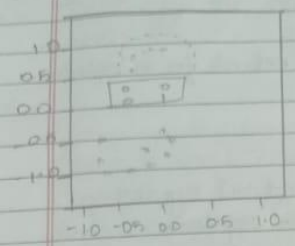
Code:

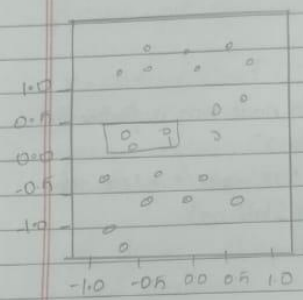
```
from sklearn.neural_network import MLP regressor
from sklearn.datasets import make regression
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%. matplotlib inline
x,y = make_regression(n_samples=1000, noise=0.00, n_feature=100)
x.shape, y.shape = ((1000,100); (1000,))
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size =
        0.2, shuffle=True, random_State=42)
clf = MLP Regressor(max_iter=1000)
clf.fit(x_train, y_train)
```

Output:

R2 Score for test Data = 0.9686558492152

Result: The program is executed Successfully.





Result: The program is successfully executed

Exp. 10 Implementing Artificial Neural Network for an application using Python.

Aim : To implementing artificial neural network for an application in classification using python

Code :
```
Sklearn.model_selection import train_test_split
from sklearn.datasets import make_circles
import from sklearn.neural_network import MLP
                                        classifier

from numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
% matplotlib Inline
x_train, y_train = make_circles(n_samples=100, noise=0.05)
x_test, y_test = make_circles(n_samples=300, noise=0.05)
sns.scatterplot(x_train[:,0]
 x_train[:,1], hue=y_train)
pet.title('Train Data')
pet.show()
if = MLPClassifier(max_iter=1000)
if.fit(x_train, y_train)
y_pred = if.predict(x_test)
fig,ax = pet.subplots(1,2)
sns.Scatterplot x_test[:,0]
x_test[:,1], hue= y_pred, ax=ax[0])
pet.show()
```

Exp No: 9 Fuzzy logic Image Processing.

Aim: To enhance the robustness and accuracy of edge detection in Image by handling uncertainties in pixel intensity transition.

Procedure:

1. Set up the environment
2. Import and Convert image to GrayScale
3. Convert image to double-precision
4. obtain image gradient
5. Define fuzzy inference System for edge detection
6. Specify FIS Rules
7. Evaluate FIS.
8. Plot Results.

Result: The program is executed Successfully.

Exp No:2 Unification and Revolution

Aim: To execute program based on Unification and Revolution
Deduction in Prolog is based on unification and instantiation.
Matching terms are unified.

Procedure:

1. Set up Prolog Environment.
2. Creating a knowledge base file
3. Load the knowledge Base.
4. Define goals for Refutation.
5. Execute queries for each level.
6. Review Results
7. Conclusion
8. Exit prolog.

Output: ?- not(strawberry,pudding)
true
?~wet
true.

Result: Thus the program is successfully executed

Exp. 01 Introduction to Prolog

Aim: To learn PROLOG terminologies and write basic programs.

Procedure:
1. Create a file: Open a text editor and save the following knowledge
2. Open prolog: Launch your prolog.
3. Load Knowledge Base
4. Execute Queries
5. View Result.
6. Exit prototype

Output:
```
?-woman(ma)
true
?-party
true
?-concert
Error: Unknown procedure: concert/0
```

Result: The program is Successfully executed.

Exp. 06 Prolog

Aim: To develop a family tree programming using Prolog with all possible facts, rules and queries

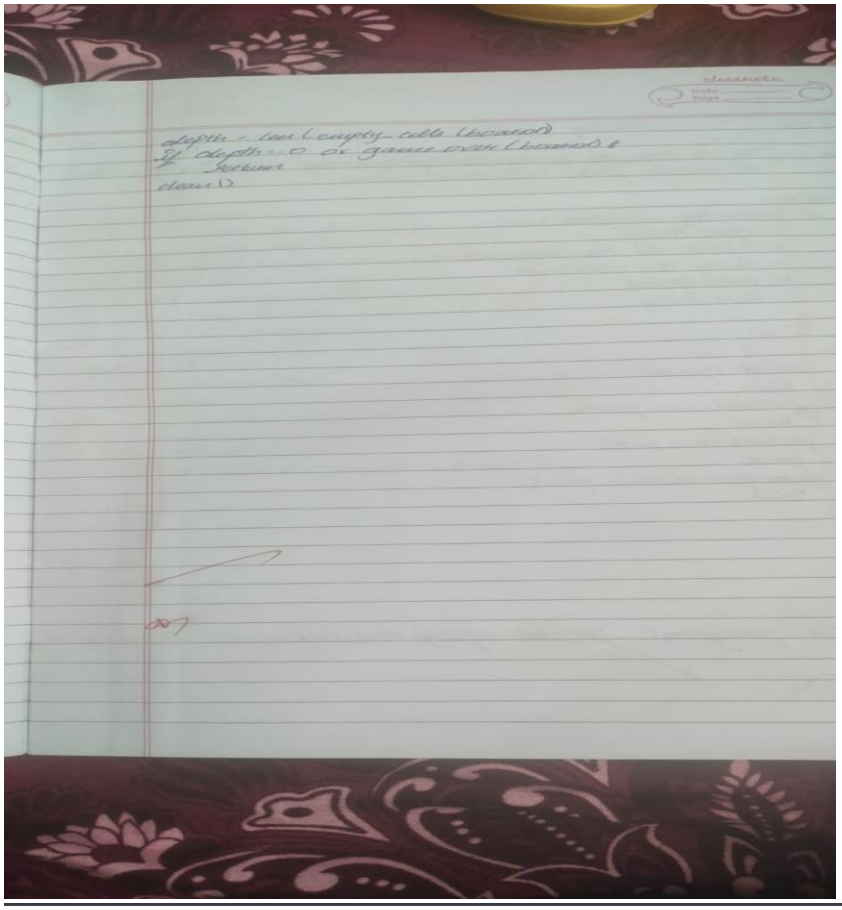
Procedure:

1. Create a file
2. Open Prolog
3. Load the knowledge base
4. Query the database
5. View Result
6. Exit Prolog

Output:

male(peter)
true
father(chris-peter)
true
father(chris-betty)
false
grandfather(Kevin-peter)
true

Result: The program is successfully executed.



Untitled3.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

```python
# Example tic-tac-toe board
board = [
    ['X', 'O', 'X'],
    [' ', 'X', 'O'],
    ['O', ' ', ' ']
]

best_move = find_best_move(board)
print("Best move for X is at position:", best_move)
```

```
Best move for X is at position: (2, 2)
```

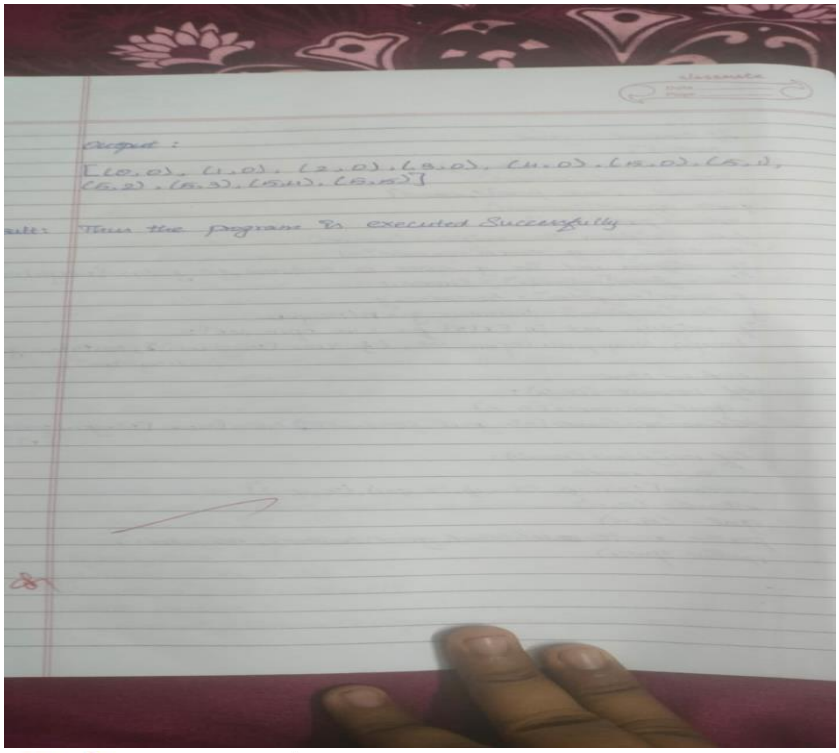
```
    if [player, player, player] in win_state:
        return TRUE
    else:
        return FALSE
def game_over(state):
    return wins(state, HUMAN) or wins(state, COMP)
def empty_cells(state):
    cells = []
    for x, row in enumerate(state):
        for y, cell in enumerate(row):
            if cell = 0:
                cells.append([x,y])
    return cells
def valid_move(x,y):
    return cells
def valid_move(x,y):
    if [x, y] in empty_cells(board):
        return TRUE
    else
        return FALSE
def render(state, c_choice, h_choice):
    chars = {
        -1 : h_choice,
        +1 : c_choice,
        0 : ' ' }
    str_line = "-----"
    print("\n"+str_line)
    for row in state:
        for cell in row:
            symbol = chars[cell]
            print("|{}".format(symbol), end = " ")
        print("|\n"+str_line)
def ai_turn(c_choice, h_choice):
```

Ex.05 Minimax Algorithm

Aim: To implement minimax Algorithm problem using python

Source Code:

```
from math import inf as infinity
from random import choice
import platform
import time
from os.import System
HUMAN = -1
COMP = +1
board = [
[0,0,0],
[0,0,0],
[0,0,0],
]
def evaluate(state):
  if wins(state, COMP):
    score = +1
elif wins(state, HUMAN):
    score = -1
else
    score = 0
return score
def wins(state, player):
    win_state=[
[state[0][0], state[0][1], state[0][2]],
[state[1][0], state[1][1], state[1][2]],
[state[2][0], state[2][1], state[2][2]]
[state[0][0], state[1][0], state[2][0]],
[state[0][1], state[1][1], state[2][1]],
[state[0][2], state[1][2], state[2][2]],
[state[0][0], state[1][1], state[2][2]], ]
```



Output :
[(0,0), (1,0), (2,0), (3,0), (4,0), (5,0), (5,1), (5,2), (5,3), (5,4), (5,5)]

Result: Thus the program is executed successfully

Untitled3.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

```
    [0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0]
]

# Start and goal positions
start = (0, 0)
goal = (4, 4)

# Run A* search
path = a_star(start, goal, grid)

if path:
    print("Path found:", path)
else:
    print("No path found")
```

```
Path found: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (4, 2), (4, 3), (4, 4)]
```

```
if current == goal:
path = []
while current in came_from:
path.append(current)
current = came_from[current]
path.append(current)
return path[::-1]
for neighbor in neighbors(current):
tentative_g = g_score[current] + 1
if neighbor not in g_score or tentative_g < g_score[neighbor]
came_from[neighbor] = current
g_score[neighbor] = tentative_g
f_score[neighbor] = tentative_g + h(neighbor)
if neighbor not in [i[2] for i in open_set]:
heapq.heappush(open_set, (f_score[neighbor], tentative_g,
                                                neighbor))

return None
def heuristic(node):
  goal_position = (5,5)
return abs(node[0] - goal_position[0]) + abs(node[1] - goal_position
                                                [1])

def neighbors(node):
  x,y = node
return [(x+1,y), (x-1,y), (x,y+1), (x,y-1)]
start = (0,0)
goal = (5,5)
path = a_star(start, goal, heuristic, neighbors)
print(path)
```

Exp 04: A* Search

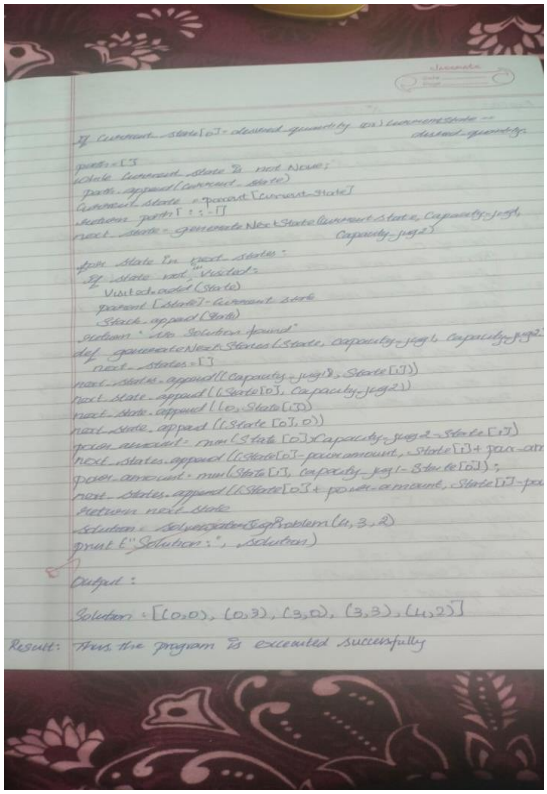
Aim:
To find the shortest path from a start node to a goal node using A* Search.

Algorithm:

Step 1: Create open and closed sets, starts with the initial node
Step 2: Add the start node to the open set with an initial cost of 0.
Step 3: Remove the node with lowest 'f' value (cost + heuristic) from the open set.
Step 4: If the current node is the goal node, reconstruct the path.
Step 5: For each neighbour, calculate 'g', 'h', 'f' values
Step 6: If the neighbour is not in the open set (or) a lower cost path is found, update cost and parent
Step 7: Add the neighbour to open set if it is not already in the closed set.
Step 8: Repeat until the open set is empty or the goal is found.

Program:

```
import heapq
def a_star(start, goal, h, neighbours):
    open_set = []
heapq.heappush(open_set, (0+h(start),0,start))
came_from = {}
g_score = {start:0}
f_score = {start:h(start)}
while open_set:
    _,current_g,current = heapq.heappop(open_set)
```



```
if current_state[0] = desired_quantity or current_state[1] = desired_quantity:
path = []
while current_state is not None:
    path.append(current_state)
    current_state = parent[current_state]
return path[::-1]
next_state = generateNextStates(current_state, capacity_jug1, capacity_jug2)

for state in next_state:
    if state not in visited:
        visited.add(state)
        parent[state] = current_state
        stack.append(state)
return "No Solution found"
def generateNextStates(state, capacity_jug1, capacity_jug2):
    next_states = []
    next_states.append((capacity_jug1, state[1]))
    next_state.append((state[0], capacity_jug2))
    next_state.append((0, state[1]))
    next_state.append((state[0], 0))
    pour_amount = min(state[0], capacity_jug2 - state[1])
    next_states.append((state[0] - pour_amount, state[1] + pour_am
    pour_amount = min(state[1], capacity_jug1 - state[0]);
    next_states.append((state[0] + pour_amount, state[1] - pou
    return next_state
solution = solveWaterJugProblem(4, 3, 2)
print f"Solution:", solution)
```

Output :

Solution = [(0,0), (0,3), (3,0), (3,3), (4,2)]

Result: Thus the program is executed successfully

Untitled3.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

```
# Example usage
x = 4  # Capacity of Jug 1
y = 3  # Capacity of Jug 2
z = 2  # Target amount of water
water_jug_bfs(x, y, z)
```

```
Found solution: Jug 1 = 4L, Jug 2 = 2L
True
```

Ex 03 Water Jug

Aim:
To solve the water jug problem using DFS.

Algorithm:

Step1: Create a stack to store the states of the jug.
Step2: Initialise the stack is not empty, with the initial state.
Step3: While the stack is not empty, do the following
- Pop a state from stack
- If the stack represent the desired quantity, stop and return the solution
- Generate all possible next states from the current state.
- Push the next states on to the stack.

Step4: If the stack becomes empty and no solution is found, the problem is unsolvable.

Program:

```
def solveWaterJugProblem(capacity_jug1,capacity_jug2,
                         desired_quantity):
stack = []
visited = set()
parent = { }
start_state = (0,0)
stack.append(start_state)
visited.add(start_state)
parent[start_state] = None
while stack:
    current_state = stack.pop()
```

Program :

```
def dfs (graph, start):
    stack = [start]
    visited = set()
while stack :
    node = stack.pop()
if node not in visited :
    print (node, end = " ")
    visited.add (node)
    for neighbor in graph[node]:
        if neighbor not in visited
            stack.append(neighbor)
graph = {
'A' : ['B','C']
'B' : ['D','E']
'C' : ['F']
'D' : []
'E' : ['F']
'F' : []
}
print ("DFS traversal starting from node 'A' :")
dfs (graph, 'A')
```

Output :

DFS Traversal starting from node 'A':

ACFBED

Result: Thus, the program has been successfully executed.

Untitled3.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

```
dfs(graph, 'A')
```

```
A B D E F C
```

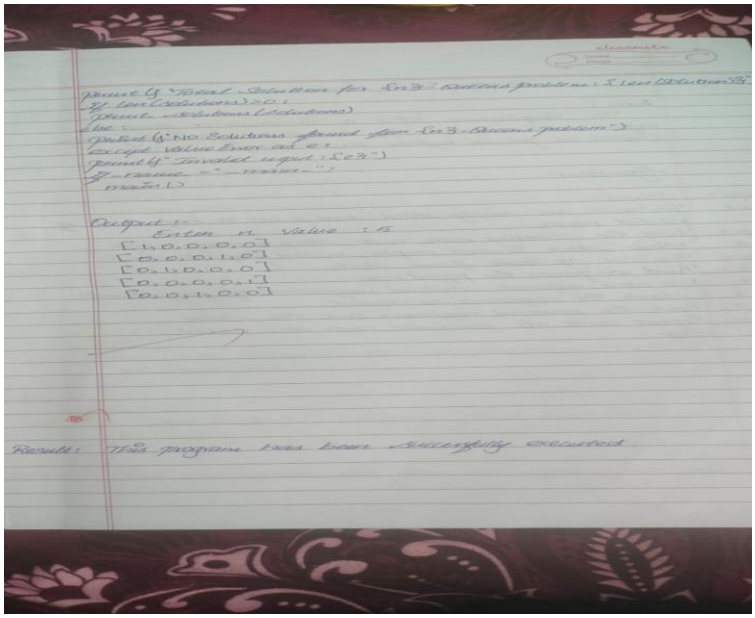
Exp: 02 Depth First Search

Aim:

To Implement DFS to traverse a graph & explore all vertices by visiting as far along each branch as possible before backtracking.

Algorithm:

Step1: Start
Step2: Initialise an empty stack and a list to keep track of visited nodes.
Step3: Push the starting node onto stack & mark visited
Step4: While stack is not empty, repeat Step5 to Step7
Step5: Pop the top node from stack.
Step6: Print (or) process the popped node.
Step7: For each adjacent unvisited neighbour of popped node
Step8: Mark the neighbour as visited.
Step9: Push the unvisited neighbour onto the stack
Step10: Repeat until all reachable node are visited
Step11: Stop.



```
print("Total Solutions for {n}x{n} chessboard problem: {len(solutions)}")
if len(solutions)>0:
print(solutions[0:solutions])
else:
print("No Solutions found for {n}x{n} chessboard problem")
except ValueError as e:
print("Invalid input:{e}")
if __name__ == "__main__":
main()
```

Output:-

```
Enter n Value : 4
[1,0,0,0,0]
[0,0,0,1,0]
[0,1,0,0,0]
[0,0,0,0,1]
[0,0,1,0,0]
```

Result: The program has been successfully executed.

```
. Q . .
. . . Q
Q . . .
. . Q .

--------------------
. . Q .
Q . . .
. . . Q
. Q . .

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

Program :-

```
def solve_nqueens(n):
 def is_valid(board, row, col):
for i in range(row):
 if board[i] == color |
 board[i]-1==col or row or |
 board[i]+1 == col+row:
   return False
return True
def solve(board, row):
 if row==n:
  result.append(board[:])
 return
for col in range(n):
 if is_valid(board, row, col):
   board[row]= col
  solve(board, row+1)
  board[row]=-1
result = []
board=[-1]*n
solve(board, 0)
return result
def print_solutions(solutions):
for sol in solutions:
for row in sol:
print(" ".join("Q" if i==row else ". " for i in range(len(sol)))
print()
print("="*(len(sol)*2-1))
def main():
try:
n = int(input("Enter the value:"))
if n<1:
raise ValueError("N must be positive integer")
solution = solve_nqueens(n)
```

Exp:01

N-Queen Backtracking

Aim:-
To write a python program for N-Queen backtracking problem.

Algorithm:-

Step1: Start
Step2: Create a n×n chessboard with all cells set to 0, representing no queens placed.
Step3: Ensure no queen is in the same row, upper diagonal or lower diagonal for given position
Step4: Try placing a queen in each row of the current column if it is safe using is_safe.
Step5: Move to the next column if placing a queen works else backtrack by removing queen.
Step6: If queens are placed in all columns return success. Display the board.
Step7: If no solutions exist, print solution does not exist".