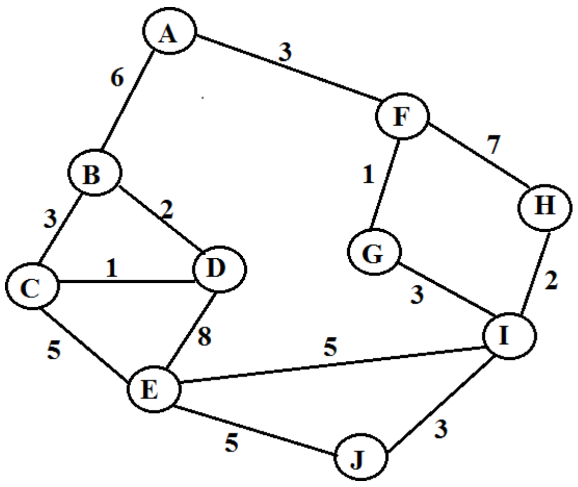
1) Implement A* Algorithm

Algorithm:

Find the most cost effective path to read from start state A to final state J using A*Algorithm.
Step 1: Place the starting node into OPEN and find its f (n) value.

Step 2: Remove the node from OPEN, having smallest f (n) value. If it is a goal nodethen stop and return success.
Step 3: Else remove the node from OPEN, find all its successors.

Step 4: Find the f (n) value of all successors; place them into OPEN and place the removednode into CLOSE.
Step 5: Go to Step-2. Step 6: Exit.



```
class Graph:

  def __init__(self, adjac_lis):

      self.adjac_lis = adjac_lis  # Initialize the adjacency list when
creating the graph object
```

```
    def get_neighbours(self, v):
        return self.adjac_lis[v]  # Return the neighbors of a node v based on the adjacency list

    def h(self, n):
        H = {'A': 1, 'B': 1, 'C': 1, 'D': 1}  # Define a heuristic function H
        return H[n]  # Return the heuristic value for node n

    def a_star_algorithm(self, start, stop):
        open_lst = set([start])  # Initialize an open set with the start node
        closed_lst = set([])  # Initialize a closed set as empty
        dist = {}  # Initialize a dictionary to store the distance from start to each node
        dist[start] = 0  # Distance from start to itself is 0
        prenode = {}  # Initialize a dictionary to store predecessors for path reconstruction
        prenode[start] = start  # Predecessor of start is start itself

        while len(open_lst) > 0:  # Loop until the open set is not empty
            n = None  # Initialize n as None for now
            for v in open_lst:  # Loop through nodes in the open set
                if n == None or dist[v] + self.h(v) < dist[n] + self.h(n):
```

```
                n = v  # Update n if a shorter path to n is found among open
nodes

            if n == None:  # If n is still None, no path is found
                print("path does not exist")
                return None  # Return None indicating no path exists

            if n == stop:  # If the goal is reached, reconstruct the path
                reconst_path = []
                while prenode[n] != n:
                    reconst_path.append(n)
                    n = prenode[n]
                reconst_path.append(start)
                reconst_path.reverse()
                print("path found: {}".format(reconst_path))
                return reconst_path  # Return the reconstructed path


            for (m, weight) in self.get_neighbours(n):  # Loop through
neighbors of node n
                if m not in open_lst and m not in closed_lst:  # If neighbor not
in open or closed set
                    open_lst.add(m)  # Add it to the open set
                    prenode[m] = n  # Set its predecessor to n
                    dist[m] = dist[n] + weight  # Update its distance from start
                else:
                    if dist[m] > dist[n] + weight:  # If a shorter path to m is
found
                        dist[m] = dist[n] + weight  # Update the distance
                        prenode[m] = n  # Update its predecessor
                        if m in closed_lst:  # If m was in the closed set
                            closed_lst.remove(m)  # Remove it from closed set
                            open_lst.add(m)  # Add it to open set
            open_lst.remove(n)  # Remove n from open set as it has been
evaluated
            closed_lst.add(n)  # Add n to closed set as it's fully evaluated

        print("Path does not exist")  # If the while loop ends without
finding the goal, no path exists
        return None

# Example adjacency list
adjac_lis = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
```

```
graph1 = Graph(adjac_lis)  # Create the graph object

graph1.a_star_algorithm('A', 'D')  # Run A* algorithm to find the path from 'A' to 'D'.
```

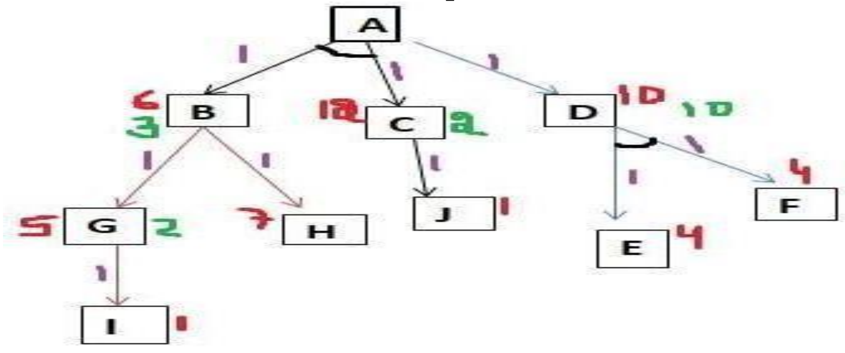
Output:

```
path found:['A', 'B', 'D']

['A', 'B', 'D']
```

2) Implement AO* Algorithm

Algorithm

1. It is an informed search and works as Best First Search.
2. AO* algorithm is based on problem decomposition.
3. It represents an AND-OR graph algorithm that is used to find more than one solution.
4. It is an efficient method to explore a solution path.



```
# Cost to find the AND and OR path

def Cost(H, condition, weight=1):

    cost = {}  # Initialize an empty dictionary to store costs for paths
    if 'AND' in condition:  # Check if 'AND' condition exists in the given condition dictionary
        AND_nodes = condition['AND']  # Retrieve nodes associated with the 'AND' condition
        Path_A = ' AND '.join(AND_nodes)  # Create a string representation of the AND path
        PathA = sum(H[node] + weight for node in AND_nodes)  # Calculate cost for the AND path
        cost[Path_A] = PathA  # Store the cost for the AND path in the dictionary
    if 'OR' in condition:  # Check if 'OR' condition exists in the given condition dictionary
        OR_nodes = condition['OR']  # Retrieve nodes associated with the 'OR' condition
        Path_B = ' OR '.join(OR_nodes)  # Create a string representation of the OR path
        PathB = min(H[node] + weight for node in OR_nodes)  # Calculate cost for the OR path
        cost[Path_B] = PathB  # Store the cost for the OR path in the dictionary

    return cost  # Return the dictionary containing costs for AND and OR paths
```

This function takes in a heuristic dictionary **H**, a condition dictionary **condition** that specifies AND and OR conditions for paths, and an optional weight parameter. It then calculates the costs for both AND and OR paths based on the given conditions and heuristic values.

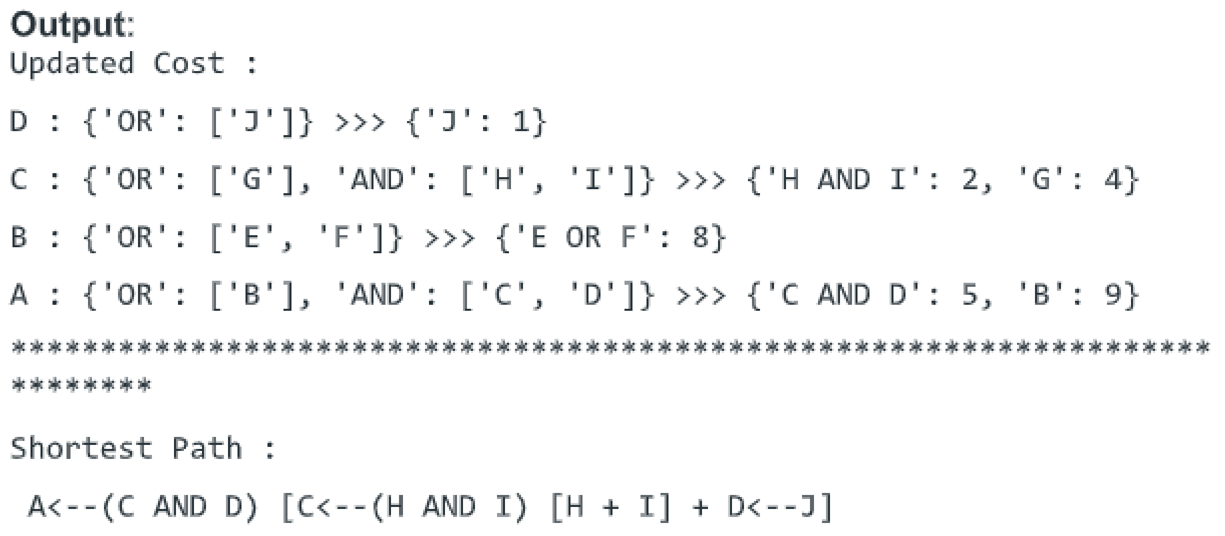
- It checks if 'AND' condition exists, extracts associated nodes, creates a string representation of the path, calculates its cost, and stores it in the dictionary.
- Similarly, it does the same for the 'OR' condition.
- Finally, it returns a dictionary containing the calculated costs for the AND and OR paths.

```
# Update the cost

def update_cost(H, Conditions, weight=1):

    Main_nodes = list(Conditions.keys())  # Get a list of keys (nodes) from the Conditions dictionary

    Main_nodes.reverse()  # Reverse the order of nodes



    least_cost = {}  # Initialize an empty dictionary to track the least cost for each node

    for key in Main_nodes:  # Iterate through the nodes in reversed order

        condition = Conditions[key]  # Get the condition associated with the current node

        print(key, ':', Conditions[key], '>>', Cost(H, condition, weight))  # Display the current node and its condition
        c = Cost(H, condition, weight)  # Calculate the cost for the current node's condition

        H[key] = min(c.values())  # Update the heuristic value of the current node to the minimum cost calculated

        least_cost[key] = Cost(H, condition, weight)  # Store the cost for the current node in the least_cost dictionary


    return least_cost  # Return a dictionary containing costs for each node after updates


# Print the shortest path

def shortest_path(Start, Updated_cost, H):

    Path = Start  # Initialize the path with the starting node

    if Start in Updated_cost.keys():  # Check if the starting node exists in the Updated_cost dictionary

        Min_cost = min(Updated_cost[Start].values())  # Find the minimum cost associated with the starting node

        key = list(Updated_cost[Start].keys())  # Get the keys (paths) associated with the starting node's costs

        values = list(Updated_cost[Start].values())  # Get the values (costs) associated with the starting node's paths

        Index = values.index(Min_cost)  # Find the index of the minimum cost

        # FIND MINIMUM PATH KEY
        Next = key[Index].split()  # Split the key into individual nodes or
paths

        # ADD TO PATH FOR OR PATH

        if len(Next) == 1:  # If the length of Next is 1, it's a single node or
path

            Start = Next[0]  # Update the starting node for the next iteration

            Path += '<--' + shortest_path(Start, Updated_cost, H)  #
Recursively find the shortest path

        # ADD TO PATH FOR AND PATH

        else:  # If the length of Next is more than 1, it represents multiple
nodes or paths

            Path += '<--(' + key[Index] + ') '  # Add the representation of
multiple paths to the path string

            Start = Next[0]  # Update the starting node for the AND path

            Path += '[' + shortest_path(Start, Updated_cost, H) + ' + '  #
Recursively find the shortest path for the AND path

            Start = Next[-1]  # Update the starting node for the remaining
path

            Path += shortest_path(Start, Updated_cost, H) + ']'  #
Recursively find the shortest path for the remaining path



    return Path  # Return the shortest path

Conditions = {
    'A': {'OR': ['B'], 'AND': ['C', 'D']},

    'B': {'OR': ['E', 'F']},

    'C': {'OR': ['G'], 'AND': ['H', 'I']},

    'D': {'OR': ['J']}

}
```

This dictionary represents conditions for different nodes in a graph. Here's an explanation for each line:

- **'A'**: Node 'A' has two conditions:
  - **'OR': ['B']**: Represents an OR condition for node 'A', indicating a possible path from node 'A' to node 'B'.
  - **'AND': ['C', 'D']**: Represents an AND condition for node 'A', indicating that there are paths from node 'A' to both nodes 'C' and 'D' simultaneously.
- **'B'**: Node 'B' has one condition:
  - **'OR': ['E', 'F']**: Represents an OR condition for node 'B', indicating possible paths from node 'B' to either node 'E' or node 'F'.
- **'C'**: Node 'C' has one OR condition and one AND condition:
  - **'OR': ['G']**: Represents an OR condition for node 'C', indicating a possible path from node 'C' to node 'G'.
  - **'AND': ['H', 'I']**: Represents an AND condition for node 'C', indicating paths from node 'C' to both nodes 'H' and 'I' simultaneously.
- **'D'**: Node 'D' has one condition:
  - **'OR': ['J']**: Represents an OR condition for node 'D', suggesting a possible path from node 'D' to node 'J'.

Each node in the dictionary ('A', 'B', 'C', 'D') represents a node in a graph, and the associated conditions ('OR' and 'AND



Output:

```
Updated Cost :

D : {'OR': ['J']} >>> {'J': 1}

C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'H AND I': 2, 'G': 4}

B : {'OR': ['E', 'F']} >>> {'E OR F': 8}

A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 5, 'B': 9}

**********************************************************************
********

Shortest Path :

 A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]
```

3. FOR A GIVEN SET OF TRAINING DATA EXAMPLES STORED IN A .CSV FILE, IMPLEMENT AND DEMONSTRATE THE CANDIDATE-ELIMINATION ALGORITHMTO OUTPUT A DESCRIPTION OF THE SET OF ALL HYPOTHESES CONSISTENT WITH THE TRAINING EXAMPLES.

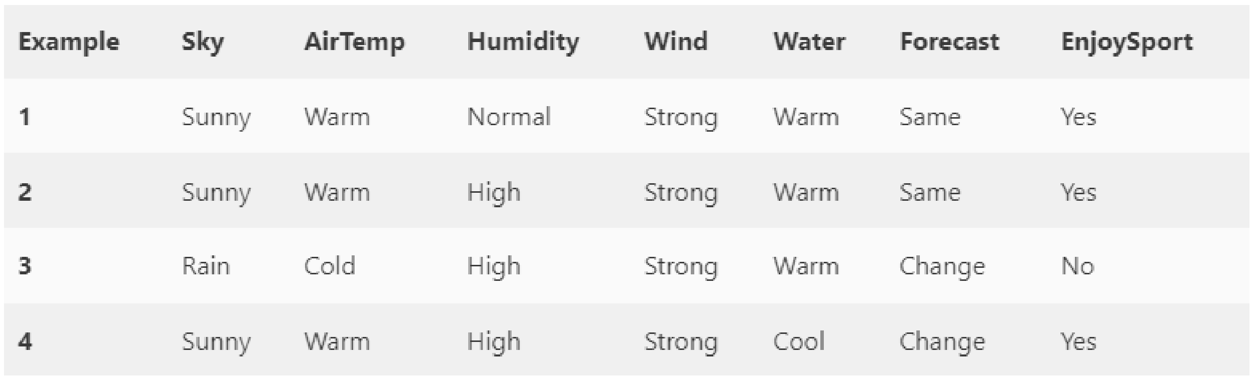
Task: The CANDIDATE-ELIMINATION algorithm computes the version space containingall hypotheses from H that are consistent with an observed sequence of trainingexamples.

Dataset: Enjoy Sports Training Examples:

Ex	**Sky**	**AirTemp**	**Humidity**	**Wind**	**Water**	**Forecast**	**EnjoySport**
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

```
import numpy as np

import pandas as pd

# Loading Data from a CSV File

data = pd.DataFrame(data=pd.read_csv('trainingdata.csv'))

print(data)

# Separating concept features from Target concepts =
np.array(data.iloc[:,0:-1])

print(concepts)

# Isolating target into a separate DataFrame # copying last column to
target array

target = np.array(data.iloc[:,-1])

print(target)


def learn(concepts, target):

    # Initialise S0 with the first instance from concepts

    # .copy() makes sure a new list is created instead of just pointing to
the same memory location

    specific_h = concepts[0].copy()

    print("\nInitialization of specific_h and general_h")

    print(specific_h)
    general_h = [["?" for i in range(len(specific_h))] for i in
range(len(specific_h))]
    print(general_h)


    # The learning iterations
    for i, h in enumerate(concepts):
        # Checking if the hypothesis has a positive target if target[i] ==
"Yes":
        for x in range(len(specific_h)):
            # Change values in S & G only if values change
            if h[x] != specific_h[x]:
                specific_h[x] = '?'
                general_h[x][x] = '?'


        # Checking if the hypothesis has a negative target
        if target[i] == "No":
            for x in range(len(specific_h)):
                # For negative hypothesis change values only in G
                if h[x] != specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else:
                    general_h[x][x] = '?'

        print("\nSteps of Candidate Elimination Algorithm",i+1)
        print(specific_h)
        print(general_h)

        # find indices where we have empty rows, meaning those that are unchanged
        indices = [i for i, val in enumerate(general_h) if val == ['?', '?', '?', '?', '?', '?']]
        for i in indices:
            # remove those rows from general_h
            general_h.remove(['?', '?', '?', '?', '?', '?'])

    # Return final values
    return specific_h, general_h

s_final, g_final = learn(concepts, target)
print("\nFinal Specific_h:", s_final, sep="\n")
print("\nFinal General_h:", g_final, sep="\n")
```

OUTPUT



Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rain	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

```
Change      Yes[['Sunny' 'Warm' 'Normal' 'Strong'
'Warm' 'Same']
 ['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
 ['Rainy' 'Cold' 'High' 'Strong' 'Warm' 'Change']
 ['Sunny' 'Warm' 'High' 'Strong' 'Cool'
'Change']]['Yes' 'Yes' 'No' 'Yes']

Initialization of specific_h and
general_h ['Sunny' 'Warm' 'Normal'
'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?',
'?'], ['?','?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination
Algorithm 1 ['Sunny' 'Warm' 'Normal'
'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?',
'?'], ['?','?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination
Algorithm 2['Sunny' 'Warm' '?'
'Strong' 'Warm' 'Same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?',
'?'], ['?','?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination
Algorithm 3['Sunny' 'Warm' '?'
'Strong' 'Warm' 'Same']
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',
'?', '?','?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]
```

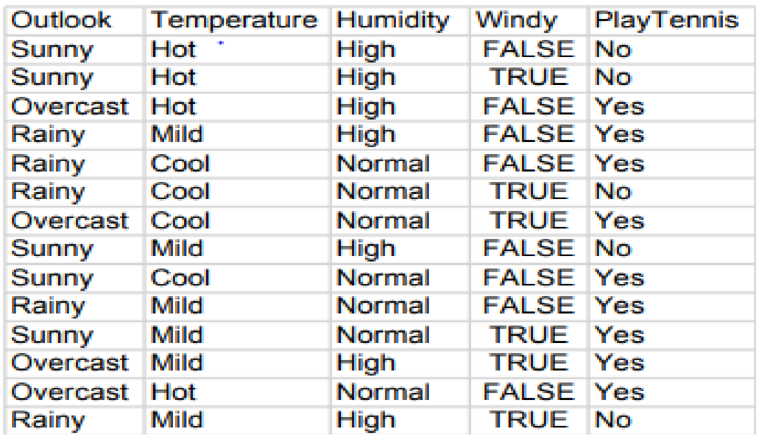
```
Steps of Candidate Elimination
Algorithm 4['Sunny' 'Warm' '?'
'Strong' '?' '?']
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',
'?', '?','?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]


Final Specific_h:
['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final General_h:
[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]
```

4) Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate dataset for building the decision tree and apply this knowledge to classify a new sample.

Task: ID3 determines the information gain for each candidate attribute, then selects the one with highest information gain as the root node of the tree. The information gain values for all four attributes are calculated using the following formula:

$$\text{Entropy}(S)=\sum -P(I).\log_2 P(I)$$

$$\text{Gain}(S,A)=\text{Entropy}(S)-\sum[P(S/A).\text{Entropy}(S/A)]$$



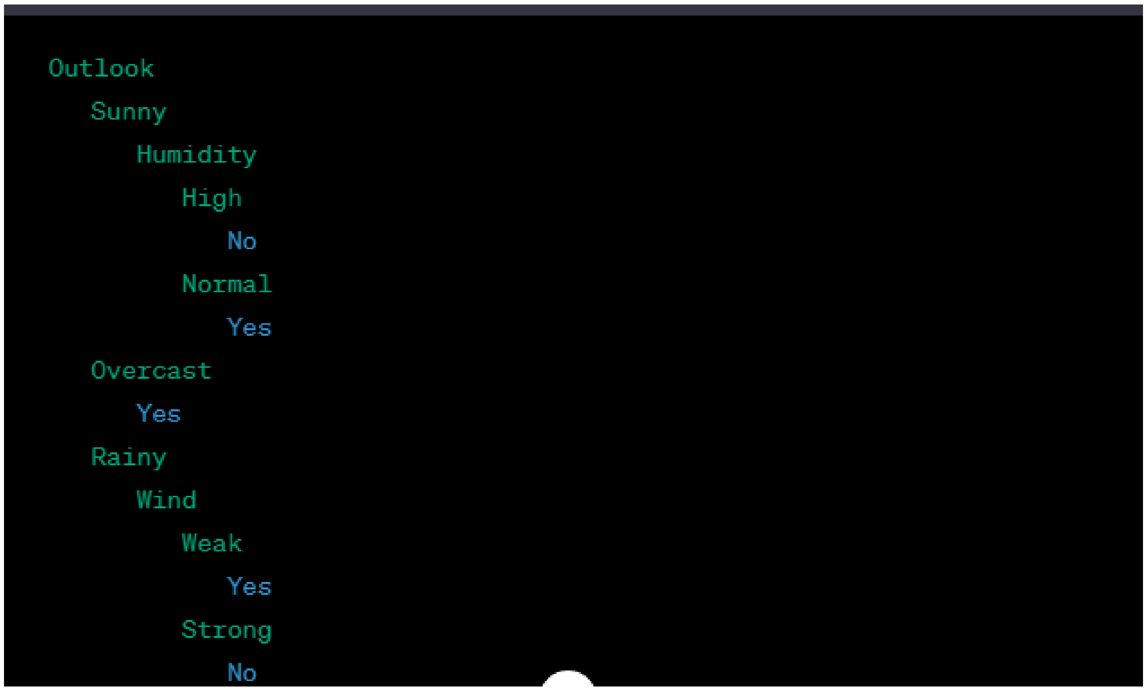
Outlook	Temperature	Humidity	Windy	PlayTennis
Sunny	Hot	High	FALSE	No
Sunny	Hot	High	TRUE	No
Overcast	Hot	High	FALSE	Yes
Rainy	Mild	High	FALSE	Yes
Rainy	Cool	Normal	FALSE	Yes
Rainy	Cool	Normal	TRUE	No
Overcast	Cool	Normal	TRUE	Yes
Sunny	Mild	High	FALSE	No
Sunny	Cool	Normal	FALSE	Yes
Rainy	Mild	Normal	FALSE	Yes
Sunny	Mild	Normal	TRUE	Yes
Overcast	Mild	High	TRUE	Yes
Overcast	Hot	Normal	FALSE	Yes
Rainy	Mild	High	TRUE	No

```
import numpy as np

import math

import csv

# Function to read data from a CSV file

def read_data(filename):

 with open(filename, 'r') as csvfile:

        datareader = csv.reader(csvfile, delimiter=',')

        headers = next(datareader)  # Read the header row

        metadata = []  # List to hold column names

        traindata = []  # List to hold training data

        for name in headers:

            metadata.append(name)  # Store column names in metadata list

        for row in datareader:

            traindata.append(row)  # Store rows of training data

    return (metadata, traindata)  # Return metadata and training data as tuples


# Node class for building the decision tree

class Node:

    def __init__(self, attribute):

        self.attribute = attribute  # Attribute name for the node

        self.children = []  # List to hold child nodes

        self.answer = ""  # Holds the final classification answer

            def __str__(self):
```

```
        return self.attribute  # Returns the attribute name as a string


# Function to create subtables based on column values
def subtables(data, col, delete):
    dict = {}  # Dictionary to hold subtables
    items = np.unique(data[:, col])  # Unique values in the column
    count = np.zeros((items.shape[0], 1), dtype=np.int32)  # Count occurrences of
each value


    # Populate subtables based on unique values
    for x in range(items.shape[0]):
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                count[x] += 1


    # Fill the dictionary with subtables corresponding to each unique value
    for x in range(items.shape[0]):
        dict[items[x]] = np.empty((int(count[x]), data.shape[1]), dtype="|S32")
        pos = 0
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                dict[items[x]][pos] = data[y]
                pos += 1
        if delete:
            dict[items[x]] = np.delete(dict[items[x]], col, 1)  # Delete the column if
needed
    return items, dict


# Function to calculate entropy for a set
def entropy(S):
    items = np.unique(S)  # Unique values in the set
    if items.size == 1:  # If only one unique value, entropy is 0
        return 0
    counts = np.zeros((items.shape[0], 1))  # Count occurrences of each unique
value
    sums = 0
        # Calculate entropy using the formula and counts
    for x in range(items.shape[0]):
        counts[x] = sum(S == items[x]) / (S.size * 1.0)
    for count in counts:
        sums += -1 * count * math.log(count, 2)  # Entropy formula
    return sums
# Function to calculate gain ratio for a column
def gain_ratio(data, col):
    items, dict = subtables(data, col, delete=False)  # Get subtables for the column
    total_size = data.shape[0]  # Total size of the data
    entropies = np.zeros((items.shape[0], 1))  # Array to hold entropies
    intrinsic = np.zeros((items.shape[0], 1))  # Array to hold intrinsic information
      # Calculate entropies and intrinsic information
    for x in range(items.shape[0]):
        ratio = dict[items[x]].shape[0] / (total_size * 1.0)
        entropies[x] = ratio * entropy(dict[items[x]][:, -1])  # Entropy for each
subtable
        intrinsic[x] = ratio * math.log(ratio, 2)  # Intrinsic information
            total_entropy = entropy(data[:, -1])  # Total entropy of the entire set
    iv = -1 * sum(intrinsic)  # Calculate intrinsic value
        # Calculate gain ratio using entropy and intrinsic value
    for x in range(entropies.shape[0]):
        total_entropy -= entropies[x]
    return total_entropy / iv


# Function to create nodes in the decision tree
def create_node(data, metadata):
    if (np.unique(data[:, -1])).shape[0] == 1:
        node = Node("")
        node.answer = np.unique(data[:, -1])[0]  # Store the answer if only one class
remains
        return node
        gains = np.zeros((data.shape[1] - 1, 1))  # Array to hold gain ratios for each
column
```

```
        # Calculate gain ratio for each column
    for col in range(data.shape[1] - 1):
        gains[col] = gain_ratio(data, col)
        split = np.argmax(gains)  # Find the column with the highest gain ratio
    node = Node(metadata[split])  # Create a node with the split attribute
    metadata = np.delete(metadata, split, 0)  # Remove the split attribute from
metadata
        items, dict = subtables(data, split, delete=True)  # Get subtables based on the
split attribute
        # Recursively create child nodes
    for x in range(items.shape[0]):
        child = create_node(dict[items[x]], metadata)
        node.children.append((items[x], child))  # Append child nodes to the current
node
        return node  # Return the node
# Function to create indentation for tree visualization
def empty(size):
    s = ""
    for x in range(size):
        s += "   "
    return s
# Function to print the decision tree
def print_tree(node, level):
    if node.answer != "":
        print(empty(level), node.answer)

        return

    print(empty(level), node.attribute)

    for value, n in node.children:

        print(empty(level + 1), value)

        print_tree(n, level + 2)

# Read data from a CSV file

metadata, traindata = read_data("tennisdata.csv")

data = np.array(traindata)

# Create the decision tree

node = create_node(data, metadata)

print_tree(node, 0)  # Print the decision tree
```

Output:



```
Outlook
   Sunny
      Humidity
         High
            No
         Normal
            Yes
   Overcast
      Yes
   Rainy
      Wind
         Weak
            Yes
         Strong
            No
```