In [2]:

```python
# Program 1
class Graph:
    def __init__(self,adjac_lis):
        self.adjac_lis = adjac_lis
    def get_neighbours(self,v):
        return self.adjac_lis[v]
    def h(self,n):
        H = {'A':1,'B':1, 'C':1,'D':1}
        return H[n]
    def a_star_algorithm(self,start,stop):
        open_lst = set([start])
        closed_lst = set([])
        dist = {}
        dist[start] = 0
        prenode = {}
        prenode[start] = start
        while len(open_lst) > 0:
            n = None
            for v in open_lst:
                if n == None or dist[v] + self.h(v) < dist[n] + self.h(n):
                    n = v;
            if n == None:
                print("path doesnot exist")
                return None
            if n == stop:
                 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
            for (m, weight) in self.get_neighbours(n):
                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)
                    prenode[m] = n
                    dist[m] = dist[n] + weight
                else:
                    if dist[m] > dist[n] + weight:
                        dist[m] = dist[n] + weight
                        prenode[m] = n
                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)
            open_lst.remove(n)
            closed_lst.add(n)
        print("Path does not exist")
        return None
adjac_lis ={'A':[('B',1),('C',3),('D',7)],'B':[('D',5)],'C':[('D',12)]}
graph1=Graph(adjac_lis)
graph1.a_star_algorithm('A', 'D')
```

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

Out[2]:

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

In [3]:

```
def recAOStar(n):
    global finalPath
    print("Expanding Node:",n)
    and_nodes = []
    or_nodes =[]
    if(n in allNodes):
        if 'AND' in allNodes[n]:
            and_nodes = allNodes[n]['AND']
        if 'OR' in allNodes[n]:
            or_nodes = allNodes[n]['OR']
    if len(and_nodes)==0 and len(or_nodes)==0:
        return

    solvable = False
    marked ={}

    while not solvable:
        if len(marked)==len(and_nodes)+len(or_nodes):
            min_cost_least,min_cost_group_least = least_cost_group(and_nodes,or_nodes,{})
            solvable = True
            change_heuristic(n,min_cost_least)
            optimal_child_group[n] = min_cost_group_least
            continue
        min_cost,min_cost_group = least_cost_group(and_nodes,or_nodes,marked)
        is_expanded = False
        if len(min_cost_group)>1:
            if(min_cost_group[0] in allNodes):
                is_expanded = True
                recAOStar(min_cost_group[0])
            if(min_cost_group[1] in allNodes):
                is_expanded = True
                recAOStar(min_cost_group[1])
        else:
            if(min_cost_group in allNodes):
                is_expanded = True
                recAOStar(min_cost_group)
        if is_expanded:
            min_cost_verify, min_cost_group_verify = least_cost_group(and_nodes, or_nodes, {})
            if min_cost_group == min_cost_group_verify:
                solvable = True
                change_heuristic(n, min_cost_verify)
                optimal_child_group[n] = min_cost_group
        else:
            solvable = True
            change_heuristic(n, min_cost)
            optimal_child_group[n] = min_cost_group
        marked[min_cost_group]=1
    return heuristic(n)

def least_cost_group(and_nodes, or_nodes, marked):
    node_wise_cost = {}
    for node_pair in and_nodes:
        if not node_pair[0] + node_pair[1] in marked:
            cost = 0
            cost = cost + heuristic(node_pair[0]) + heuristic(node_pair[1]) + 2
            node_wise_cost[node_pair[0] + node_pair[1]] = cost
    for node in or_nodes:
        if not node in marked:
            cost = 0
            cost = cost + heuristic(node) + 1
            node_wise_cost[node] = cost
    min_cost = 999999
    min_cost_group = None
    for costKey in node_wise_cost:
        if node_wise_cost[costKey] < min_cost:
            min_cost = node_wise_cost[costKey]
            min_cost_group = costKey
    return [min_cost, min_cost_group]

def heuristic(n):
    return H_dist[n]

def change_heuristic(n, cost):
    H_dist[n] = cost
    return

def print_path(node):
    print(optimal_child_group[node], end="")
    node = optimal_child_group[node]
    if len(node) > 1:
        if node[0] in optimal_child_group:
            print("->", end="")
            print_path(node[0])
        if node[1] in optimal_child_group:
            print("->", end="")
            print_path(node[1])
    else:
        if node in optimal_child_group:
            print("->", end="")
            print_path(node)
```

```
H_dist = {
 'A': -1,
 'B': 4,
 'C': 2,
 'D': 3,
 'E': 6,
 'F': 8,
 'G': 2,
 'H': 0,
 'I': 0,
 'J': 0
}
allNodes = {
 'A': {'AND': [('C', 'D')], 'OR': ['B']},
 'B': {'OR': ['E', 'F']},
 'C': {'OR': ['G'], 'AND': [('H', 'I')]},
 'D': {'OR': ['J']}
}
optimal_child_group = {}
optimal_cost = recAOStar('A')
print('Nodes which gives optimal cost are')
print_path('A')
print('\nOptimal Cost is :: ', optimal_cost)
```

```
Expanding Node: A
Expanding Node: B
Expanding Node: C
Expanding Node: D
Nodes which gives optimal cost are
CD->HI->J
Optimal Cost is ::  5
```

In [4]:

```
# Program 3
import csv

with open("prog3.csv") as f:
    csv_file = csv.reader(f)
    data = list(csv_file)

    specific = data[0][:-1]
    general = [['?' for i in range(len(specific))] for j in range(len(specific))]

    for i in data:
        if i[-1] == "Yes":
            for j in range(len(specific)):
                if i[j] != specific[j]:
                    specific[j] = "?"
                    general[j][j] = "?"

        elif i[-1] == "No":
            for j in range(len(specific)):
                if i[j] != specific[j]:
                    general[j][j] = specific[j]
                else:
                    general[j][j] = "?"

        print("\nStep " + str(data.index(i)+1) + " of Candidate Elimination Algorithm")
        print(specific)
        print(general)
    gh = []
    for i in general:
        for j in i:
            if j != '?':
                gh.append(i)
                break
    print("\nFinal Specific hypothesis:\n", specific)
    print("\nFinal General hypothesis:\n", gh)
```

```
Step 1 of Candidate Elimination Algorithm
['sunny', 'warm', 'normal', 'strong', 'warm', 'same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?',
'?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Step 2 of Candidate Elimination Algorithm
['sunny', 'warm', '?', 'strong', 'warm', 'same']
[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?',
'?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Step 3 of Candidate Elimination Algorithm
['sunny', 'warm', '?', 'strong', 'warm', 'same']
[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'same']]

Step 4 of Candidate Elimination Algorithm
['sunny', 'warm', '?', 'strong', '?', '?']
[['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?',
'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final Specific hypothesis:
 ['sunny', 'warm', '?', 'strong', '?', '?']

Final General hypothesis:
 [['sunny', '?', '?', '?', '?', '?'], ['?', 'warm', '?', '?', '?', '?']]
```

In [5]:

```python
# Program 4
import pandas as pd
from pprint import pprint
from sklearn.feature_selection import mutual_info_classif
from collections import Counter

def id3(df, target_attribute, attribute_names, default_class=None):
    cnt=Counter(x for x in df[target_attribute])
    if len(cnt)==1:
        return next(iter(cnt))

    elif df.empty or (not attribute_names):
         return default_class

    else:
        gainz = mutual_info_classif(df[attribute_names],df[target_attribute],discrete_features=True)
        index_of_max=gainz.tolist().index(max(gainz))
        best_attr=attribute_names[index_of_max]
        tree={best_attr:{}}
        remaining_attribute_names=[i for i in attribute_names if i!=best_attr]

        for attr_val, data_subset in df.groupby(best_attr):
            subtree=id3(data_subset, target_attribute, remaining_attribute_names,default_class)
            tree[best_attr][attr_val]=subtree

        return tree
df=pd.read_csv("prog4.csv")

attribute_names=df.columns.tolist()
print("List of attribut name")

attribute_names.remove("PlayTennis")

for colname in df.select_dtypes("object"):
    df[colname], _ = df[colname].factorize()

print(df)

tree= id3(df,"PlayTennis", attribute_names)
print("The tree structure")
pprint(tree)
```

```
List of attribut name
    outlook  temp  humidity  windy  PlayTennis
0         0     0         0  False           0
1         0     0         0   True           0
2         1     0         0  False           1
3         2     1         0  False           1
4         2     2         1  False           1
5         2     2         1   True           0
6         1     2         1   True           1
7         0     1         0  False           0
8         0     2         1  False           1
9         2     1         1  False           1
10        0     1         1   True           1
11        1     1         0   True           1
12        1     0         1  False           1
13        2     1         0   True           0
The tree structure
{'outlook': {0: {'humidity': {0: 0, 1: 1}},
             1: 1,
             2: {'windy': {False: 1, True: 0}}}}
```

In [6]:

```
# Program 5
import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)

# scale units
X = X/np.amax(X, axis=0)
y = y/100

class Neural_Network(object):
    def __init__(self):

        self.inputSize = 2
        self.outputSize = 2
        self.hiddenSize = 4
        self.W1 = np.random.randn(self.inputSize, self.hiddenSize)
        self.W2 = np.random.randn(self.hiddenSize, self.outputSize)

    def forward(self, X):
        self.z = np.dot(X, self.W1)
        self.z2 = self.sigmoid(self.z)
        self.z3 = np.dot(self.z2, self.W2)
        o = self.sigmoid(self.z3)
        return o

    def sigmoid(self, s):
        return 1/(1+np.exp(-s))

    def sigmoidPrime(self, s):
        return s * (1 - s)

    def backward(self, X, y, o):
        self.o_error = y - o
        self.o_delta = self.o_error*self.sigmoidPrime(o)
        self.z2_error = self.o_delta.dot(self.W2.T)
        self.z2_delta = self.z2_error*self.sigmoidPrime(self.z2)
        self.W1 += X.T.dot(self.z2_delta)
        self.W2 += self.z2.T.dot(self.o_delta)

    def train (self, X, y):
        o = self.forward(X)
        self.backward(X, y, o)

NN = Neural_Network()
print ("\nInput: \n" + str(X))
print ("\nActual Output: \n" + str(y))
print ("\nPredicted Output: \n" + str(NN.forward(X)))
print ("\nLoss: \n" + str(np.mean(np.square(y - NN.forward(X)))))
```

```
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]

Actual Output:
[[0.92]
 [0.86]
 [0.89]]

Predicted Output:
[[0.49211237 0.37794696]
 [0.50715326 0.34596148]
 [0.48104659 0.40879093]]

Loss:
0.21074179886944763
```

In [8]:

```
# Program 6
# Program 6
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.naive_bayes import GaussianNB

data = pd.read_csv('prog6.csv')
print("The first 5 Values of data is :\n", data.head())
X = data.iloc[:, :-1]
print("\nThe First 5 values of the train data is\n", X.head())
y = data.iloc[:, -1]
print("\nThe First 5 values of train output is\n", y.head())

le_outlook = LabelEncoder()
X.Outlook = le_outlook.fit_transform(X.Outlook)
le_Temperature = LabelEncoder()
X.Temperature = le_Temperature.fit_transform(X.Temperature)
le_Humidity = LabelEncoder()
X.Humidity = le_Humidity.fit_transform(X.Humidity)
le_Windy = LabelEncoder()
X.Windy = le_Windy.fit_transform(X.Windy)

print("\nNow the Train output is\n", X.head())

le_PlayTennis = LabelEncoder()
y = le_PlayTennis.fit_transform(y)
print("\nNow the Train output is\n",y)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.20)
classifier = GaussianNB()
classifier.fit(X_train, y_train)

from sklearn.metrics import accuracy_score
print("Accuracy is:", accuracy_score(classifier.predict(X_test), y_test))
```

```
The first 5 Values of data is :
     Outlook Temperature Humidity  Windy PlayTennis
0     Sunny         Hot     High  False         No
1     Sunny         Hot     High   True         No
2  Overcast         Hot     High  False        Yes
3     Rainy        Mild     High  False        Yes
4     Rainy        Cool   Normal  False        Yes

The First 5 values of the train data is
     Outlook Temperature Humidity  Windy
0     Sunny         Hot     High  False
1     Sunny         Hot     High   True
2  Overcast         Hot     High  False
3     Rainy        Mild     High  False
4     Rainy        Cool   Normal  False

The First 5 values of train output is
 0     No
1     No
2    Yes
3    Yes
4    Yes
Name: PlayTennis, dtype: object

Now the Train output is
    Outlook  Temperature  Humidity  Windy
0        2            1         0      0
1        2            1         0      1
2        0            1         0      0
3        1            2         0      0
4        1            0         1      0

Now the Train output is
 [0 0 1 1 1 0 1 0 1 1 1 1 1 0]
Accuracy is: 0.6666666666666666
```

In [12]:

```
# Program 7
from sklearn import datasets
from sklearn import metrics
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()
X_train,X_test,y_train,y_test = train_test_split(iris.data,iris.target)
model =KMeans(n_clusters=3)
model.fit(X_train,y_train)
model.score
print('K-Mean: ',metrics.accuracy_score(y_test,model.predict(X_test)))

from sklearn.mixture import GaussianMixture
model2 = GaussianMixture(n_components=3)
model2.fit(X_train,y_train)
model2.score
print('EM Algorithm:',metrics.accuracy_score(y_test,model2.predict(X_test)))
```

```
K-Mean:  0.02631578947368421
EM Algorithm: 0.9736842105263158
```

In [13]:

```
# Program 8
from sklearn.datasets import load_iris
iris = load_iris()

from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(iris.data,iris.target,random_state=0)

from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 5)
knn.fit(x_train,y_train)

for i,item in enumerate(x_test):
    prediction = knn.predict([item])
    print("Actual : ", iris['target_names'][y_test[i]])
    print("Prediction : ", iris['target_names'][prediction], " \n")
print("Classification Accuracy : ",knn.score(x_test,y_test))
```

```
Actual :  virginica
Prediction :  ['virginica']

Actual :  versicolor
Prediction :  ['versicolor']

Actual :  setosa
Prediction :  ['setosa']

Actual :  virginica
Prediction :  ['virginica']

Actual :  setosa
Prediction :  ['setosa']

Actual :  virginica
Prediction :  ['virginica']

Actual :  setosa
```

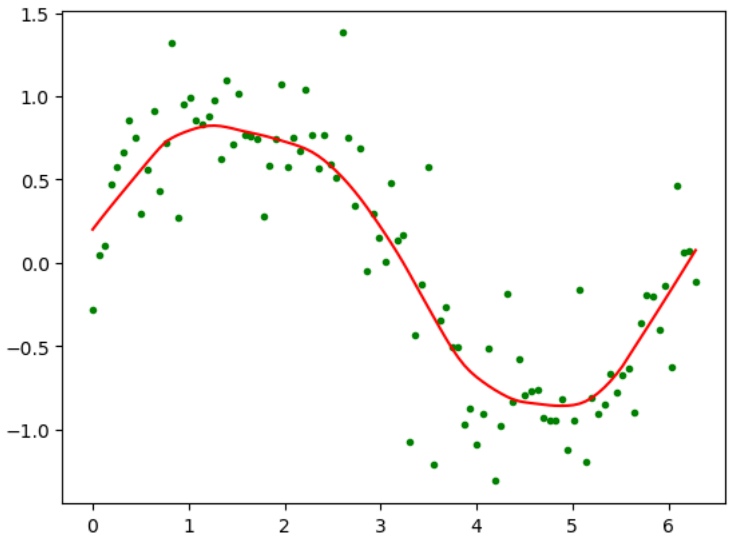
In [14]:

```
# Program 9
from math import ceil
import numpy as np
from scipy import linalg

def lowess(x, y, f, iterations):
    n = len(x)
    r = int(ceil(f * n))
    h = [np.sort(np.abs(x - x[i]))[r] for i in range(n)]
    w = np.clip(np.abs((x[:, None] - x[None, :]) / h), 0.0, 1.0)
    w = (1 - w ** 3) ** 3
    yest = np.zeros(n)
    delta = np.ones(n)
    for iteration in range(iterations):
        for i in range(n):
            weights = delta * w[:, i]
            b = np.array([np.sum(weights * y), np.sum(weights * y * x)])
            A = np.array([[np.sum(weights), np.sum(weights * x)],[np.sum(weights * x), np.sum(weights * x * x)]])
            beta = linalg.solve(A, b)
            yest[i] = beta[0] + beta[1] * x[i]

        residuals = y - yest
        s = np.median(np.abs(residuals))
        delta = np.clip(residuals / (6.0 * s), -1, 1)
        delta = (1 - delta ** 2) ** 2

    return yest

import math
n = 100
x = np.linspace(0, 2 * math.pi, n)
y = np.sin(x) + 0.3 * np.random.randn(n)
f =0.25
iterations=3
yest = lowess(x, y, f, iterations)

import matplotlib.pyplot as plt
plt.plot(x,y,"r.", color="green")
plt.plot(x,yest,"b-",color="red")
```

```
C:\Users\prsur\AppData\Local\Temp\ipykernel_90004\1619400893.py:38: UserWarning: color is redundantly defined by the 'colo
r' keyword argument and the fmt string "r." (-> color='r'). The keyword argument will take precedence.
  plt.plot(x,y,"r.", color="green")
C:\Users\prsur\AppData\Local\Temp\ipykernel_90004\1619400893.py:39: UserWarning: color is redundantly defined by the 'colo
r' keyword argument and the fmt string "b-" (-> color='b'). The keyword argument will take precedence.
  plt.plot(x,yest,"b-",color="red")
```

Out[14]:

```
[<matplotlib.lines.Line2D at 0x1d320484370>]
```



In [ ]:

```

```