Source Code:

```
#Tower of Hanoi Algorithm
def tower_of_hanoi(n, source, auxiliary, target):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n - 1, source, target, auxiliary)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n - 1, auxiliary, source, target)

# Get the number of disks from the user
n = int(input("Enter the number of disks: "))

# Ensure the user enters a positive integer
if n > 0:
    tower_of_hanoi(n, 'A', 'B', 'C')
else:
    print("Please enter a positive integer for the number of disks.")
```

Output:

```
Enter the number of disks: 3
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

Source Code:

```
#BFS
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def bfs(self, start):
        visited = set()
        queue = [start]

        while queue:
            vertex = queue.pop(0)
            if vertex not in visited:
                print(vertex, end=' ')
                visited.add(vertex)
                queue.extend([neighbor for neighbor in self.graph[vertex] if neighbor not in
visited])

# Create a graph
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)

# Get the start vertex from the user
start_vertex = int(input("Enter the start vertex: "))

# Perform BFS from the user-specified start vertex
print("Breadth-First Traversal (starting from vertex", start_vertex, "):")
g.bfs(start_vertex)
```

Output:

```
Enter the start vertex: 2
Breadth-First Traversal (starting from vertex 2 ):
2 0 3 1
```

Source Code:

```
#DFS
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs(self, vertex, visited):
        visited.add(vertex)
        print(vertex, end=' ')

        for neighbor in self.graph[vertex]:
            if neighbor not in visited:
                self.dfs(neighbor, visited)

# Example usage:
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)

# Get the start vertex from the user
start_vertex = int(input("Enter the start vertex: "))
print("Depth-First Traversal (starting from vertex", start_vertex, "):")
visited = set()
g.dfs(start_vertex, visited)
```

Output:

```
Enter the start vertex: 1
Depth-First Traversal (starting from vertex 1 ):
1 2 0 3
```

Source Code:

```
#Salesman Problem
def take_input():
    global n, cost
    n = int(input("Enter the number of nodes: "))
    cost = 0
    ary = []
    completed = [0] * n

    print("\nEnter the Cost Matrix:")

    for i in range(n):
        row = list(map(int, input().split()))
        ary.append(row)
        completed[i] = 0

    return ary, completed

def min_cost(city):
    global cost
    completed[city] = 1
    print(city + 1, end=" ---> ")
    ncity = least(city)

    if ncity == 999:
        ncity = 0
        print(ncity + 1, end=" ---> ")
        cost += ary[city][ncity]
        return

    min_cost(ncity)

def least(c):
    global cost
    nc = 999
    kmin = 999
```

```
        i = 0
        min = 999

        for i in range(n):
            if ary[c][i] != 0 and completed[i] == 0:
                if ary[c][i] + ary[i][c] < min:
                    min = ary[c][i] + ary[i][c]
                    kmin = ary[c][i]
                    nc = i

        if min != 999:
            cost += kmin

        return nc
    4

    ary, completed = take_input()

    print("\n\nThe Path is:")
    min_cost(0)  # passing 0 because it's the starting vertex

    print("\n\nMinimum cost is", cost)
```

Output:

```
Enter the number of nodes: 4

Enter the Cost Matrix:
0 3 4 6
1 0 3 3
4 1 0 4
2 4 3 0


The Path is:
1 ---> 2 ---> 3 ---> 4 ---> 1 --->

Minimum cost is 12
```

Source Code:

```
#create and load csv file
import pandas as pd
data= {'Name': ['Jai','Princi','Gaurav','Anju','Ravi','Natasha','Riya'],
      'Age': [17,17,18,17,18,17,17],
      'Gender': ['M','F','M','M','M','F','F'],
      'Marks': [90,76,'NaN',74,65,'NaN',71]}

df =pd.DataFrame(data)
df
df = pd.read_csv('1.car driving risk analysis.csv')
df
df.head()
df.tail()



#calculate mean , median,mode,var,sd from dataset
import csv
import random
import statistics

# Create a CSV file with random data
with open('dataset.csv', 'w', newline='') as csvfile:
    csv_writer = csv.writer(csvfile)
    csv_writer.writerow(['Value'])  # Write the header

    for _ in range(100):
        value = random.randint(1, 100)
        csv_writer.writerow([value])

data = []
with open('dataset.csv', 'r') as csvfile:
    csv_reader = csv.DictReader(csvfile)
    for row in csv_reader:
        data.append(int(row['Value']))
    df = pd.read_csv('dataset.csv')

    # Calculate statistics
mean = statistics.mean(data)
median = statistics.median(data)
mode = statistics.mode(data)
variance = statistics.variance(data)
stdev = statistics.stdev(data)
```

```
# Print the results
print(f"Mean: {mean}")
print(f"Median: {median}")
print(f"Mode: {mode}")
print(f"Variance: {variance}")
print(f"Standard Deviation: {stdev}")
```

Output:

	Name	Age	Gender	Marks
0	Jai	17	M	90
1	Princi	17	F	76
2	Gaurav	18	M	NaN
3	Anju	17	M	74
4	Ravi	18	M	65
5	Natasha	17	F	NaN
6	Riya	17	F	71

	speed	risk
0	200	95
1	90	20
2	300	98
3	110	60
4	240	72

	speed	risk
10	290	82
11	185	59
12	310	93
13	95	18
14	30	2

```
Mean: 49.32
Median: 49.0
Mode: 41
Variance: 767.9571717171717
Standard Deviation: 27.712040194059544
```

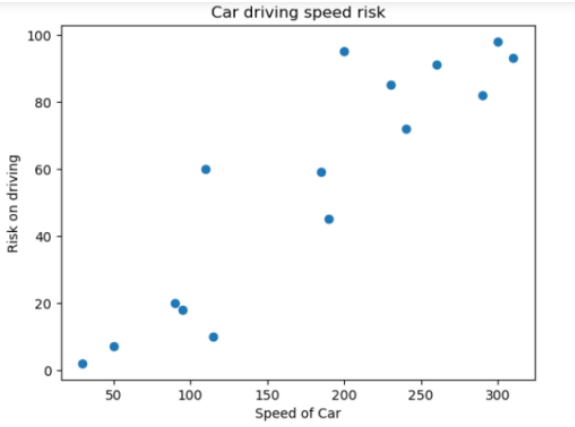
Source Code:

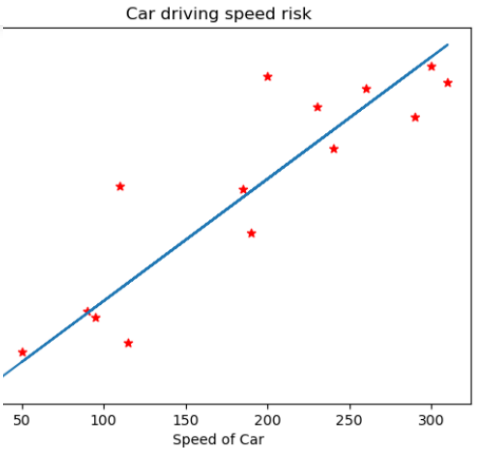
```
#linear Regression and plot the graph
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
df=pd.read_csv('1.car driving risk analysis.csv')
df
x=df[['speed']]  #only 3rd bracket,,two dimension for input means independent variable
y=df['risk']     #only 3rd bracket,,one dimension for outpu means dependent variable
x
plt.scatter(df['speed'],df['risk'])
plt.xlabel('Speed of Car')
plt.ylabel('Risk on driving')
plt.title('Car driving speed risk')
xtest
ytrain
from sklearn.linear_model import LinearRegression
reg=LinearRegression()
reg.fit(xtrain,ytrain)
LinearRegression()
reg.predict(xtest)  #compare with ytest
plt.scatter(df['speed'],df['risk'],marker='*',color='red')
plt.xlabel('Speed of Car')
plt.ylabel('Risk on driving')
plt.title('Car driving speed risk')
plt.plot(df.speed,reg.predict(df[['speed']]))
reg.predict([[180]])
reg.coef_
```

Output:

	speed	risk
0	200	95
1	90	20
2	300	98
3	110	60
4	240	72
5	115	10
6	50	7
7	230	85
8	190	45
9	260	91
10	290	82
11	185	59
12	310	93
13	95	18
14	30	2

	speed
0	200
1	90
2	300
3	110
4	240
5	115
6	50
7	230
8	190
9	260
10	290
11	185
12	310
13	95
14	30






```
array([ 27.15301215,  73.82259334,   3.81822156, 101.04651569,
        97.15738393,  77.7117251 ])
```

```
array([54.37693451])
array([0.38891318])
```

Source Code:

```
#Find S Algorithm
import pandas as pd
import numpy as np
df= pd.read_csv('S algorithm.csv')
x = np.array(df)[:,:-1] #ignore enjoy spot because it's depedent variable
y = np.array(df)[:,-1] #take only last column,it's dependent variable
def train(ind,dep):                    #ind=independent,dep=dependent
    for i,val in enumerate(dep):        #i=index:0,1,2,3 /val=yes
        if val =='yes':
            specific= ind[i].copy()
            break
    for i, val in enumerate(ind):        #i=0(index),val=['sunny' 'warm' 'normal' 'strong'
'warm' 'same']
        #print(i)
        #print(val)
        if dep[i]=='yes':
            for j in range(len(specific)):
                if val[j] !=specific[j]:
                    specific[j]='?'
                else:
                    pass
    return specific
result=train(x,y)
result
day=input("Enter 6 word to check: ")
day=day.split()
check=True
for i in range(len(result)):
    if result[i]=='?' or result[i]==day[i]:
        check=True;
    else:
        check=False;
        break;
if check:
    print("Enjoy spot")
else:
    print("Not Enjoy")
```

Output:

```
array(['sunny', 'warm', '?', 'strong', '?', '?'], dtype=object)
Enter 6 word to check: ss warm
Not Enjoy
```

Source Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load the dataset
data = pd.read_csv('Social_Network_Ads.csv')

# Select features and target variable
X = data.iloc[:, [0, 1]].values  # Assuming columns 'Age' and 'EstimatedSalary' are the relevant features
y = data.iloc[:, 2].values  # Assuming column 'Purchased' is the target variable

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Feature scaling (important for SVM)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the SVM model
svm_classifier = SVC(kernel='linear', random_state=42)
svm_classifier.fit(X_train, y_train)

# Predict on the test set
y_pred = svm_classifier.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)

print(f'Accuracy: {accuracy}')
print('Confusion Matrix:')
print(confusion)
print('Classification Report:')
print(report)
```

Output:

```
Accuracy: 0.8625
Confusion Matrix:
[[50  2]
 [ 9 19]]
Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.96      0.90        52
           1       0.90      0.68      0.78        28

    accuracy                           0.86        80
   macro avg       0.88      0.82      0.84        80
weighted avg       0.87      0.86      0.86        80
```