BREATH FIRST SEARCH

```
from collections import deque
def bfs(graph,start,goal):
    visited=set()
    queue=deque([[start]])
    if start==goal:
        return[start]
    while queue:
        path=queue.popleft()
        node=path[-1]
        if node not in visited:
            neighbours=graph.get(node,[])
            for neighbour in neighbours:
                new_path=list(path)
                new_path.append(neighbour)
                queue.append(new_path)
                if neighbour==goal:
                    return new_path
            visited.add(node)
    return"path not found"
graph={
    'A':['B','C'],
    'B':['D','E'],
```

```
    'C':['F'],
    'D':[],
    'E':['F'],
    'F':[]
}
start_node='A'
goal_node='F'
result=bfs(graph,start_node,goal_node)
print("path from",start_node,"to",goal_node,"is:",result)
```

OUTPUT

Path from A to F is: ['A', 'C', 'F']

DEPTH FIRST SEARCH

```
def dfs(graph,start,goal):
    stack=[start]
    visited=set()
    while stack:
        node=stack.pop()
        print("visiting",node)
        if node==goal:
            print("Goal found:",node)
            return True
        if node not in visited:
            visited.add(node)
            stack.extend((graph[node]))
    print("Goal not found")
    return False
graph={
    'A':['B','C'],
    'B':['D','E'],
    'C':['F'],
    'D':[],
    'E':['F'],
    'F':[]
 }
```

dfs(graph,'A','F')

OUTPUT

Visiting: A

Visiting: C

Visiting: F

Goal found: F

True

HILL CLIMBING ALGORITHM

```
import random
def objective_function(x):
    """The function we want to maximize."""
    return -(x - 5)**2+10
def hill_climbing(start_x,step_size,iterations):
    current_x = start_x
    current_value=objective_function(current_x)
    for i in range(iterations):
        next_x_pos=current_x + step_size
        next_x_neg=current_x - step_size
        val_pos=objective_function(next_x_pos)
        val_neg=objective_function(next_x_neg)
        if val_pos > current_value:
            current_x, current_value = next_x_pos, val_pos
        elif val_neg > current_value:
            current_x, current_value = next_x_neg, val_neg
        else:
            print(f"Stopping at iteration {i}: peak found.")
            break
    return current_x, current_value
initial_guess = random.uniform(0, 10)
step = 0.1
```

```
max_iter = 100
best_x, best_val = hill_climbing(initial_guess, step, max_iter)
print(f"Initial Guess: {initial_guess: .2f}")
print(f"Best x found: {best_x:.2f}")
print(f"Max value found: {best_val:.2f}")
```

OUTPUT

```
Stopping at iteration 46: Peak found.
Initial Guess: 0.41
Best X found: 5.01
Max Value found: 10.00
```

A* ALGORITHM

```
import heapq
def a_star(graph,start,goal,h):
    open_list = [(h[start], start, [start])]
    visited = set()
    while open_list:
        (f, current, path) = heapq.heappop(open_list)
        if current == goal:
            return path, f - h[goal]
        if current not in visited:
            visited.add(current)
            for neighbor, weight in graph.get(current, {}).items():
                if neighbor not in visited:
                    g_score = (f - h[current]) + weight
                    f_score = g_score + h[neighbor]
                    heapq.heappush(open_list,(f_score, neighbor, path + [neighbor]))
    return None
graph={
'A':{'B': 1, 'C':3},
'B':{'D': 2, 'E':1},
'C':{'E': 5},
'D':{'F': 1},
'E':{'F': 3},
```

```
'F':{}
}
heuristics = {'A':5, 'B':3, 'C':4, 'D':1, 'E':2, 'F':0}
path, cost = a_star(graph, 'A', 'F', heuristics)
print(f"Path: {' -> '.join(path)} | Total Cost: {cost}")
```

OUTPUT

Path: A -> B -> D -> F | Total Cost: 4

ALPHA BETA PRUNING ALGORITHM

```
MAX,MIN=1000,-1000
def minimax(depth,nodeIndex,maximizingPlayer,values,alpha,beta):
    if depth==3:
        return values[nodeIndex]
    if maximizingPlayer:
        best=MIN
        for i in range(0,2):
            val=minimax(depth+1,nodeIndex*2+i,False,values,alpha,beta)
            best=max(best,val)
            alpha=max(alpha,best)
            if beta<=alpha:
                break
        return best
    else:
        best=MAX
        for i in range(0,2):
            val=minimax(depth+1,nodeIndex*2+i,True,values,alpha,beta)
            best=min(best,val)
            beta=min(beta,best)
            if beta<=alpha:
                break
        return best
```

```
if __name__== "__main__":
    values=[3,5,6,9,1,2,0,-1]
    print("The optimal value is:",minimax(0,0,True,values,MIN,MAX))
```

OUTPUT

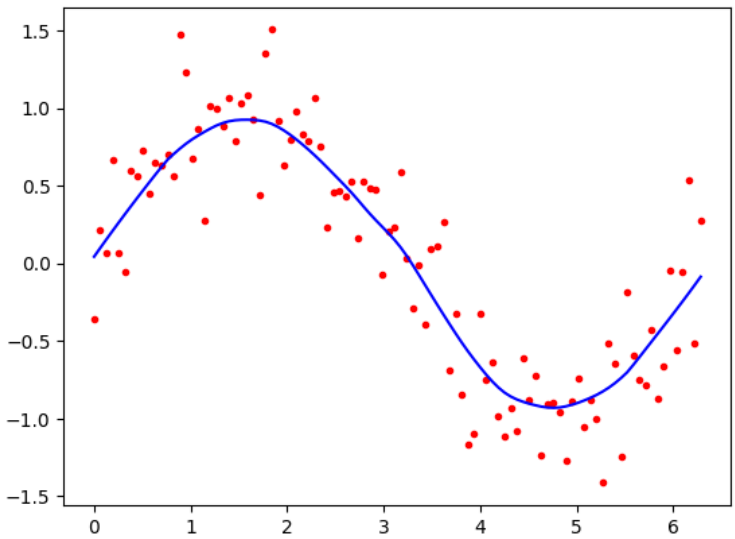
The optimal value is : 5

LINEAR REGRESSION

```
from math import ceil
import numpy as np
from scipy import linalg
import math
import matplotlib.pyplot as plt
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
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)
plt.plot(x,y,"r.")
plt.plot(x,yest,"b-")
plt.show()
```

OUTPUT



CHAT BOT

```
# Chatbot - Simple Question & Answer Program
print("Simple question & answering program")
print()
print("You may ask any of these questions:")
print("hi")
print("how are you")
print("are you working")
print("what is your name")
print("what did you do yesterday")
print("quit")
while True:
    question = input("\nEnter one question from above list: ")
    question = question.lower()
    if question in ["hi"]:
        print("Hello!")
    elif question in ["how are you", "how do you do"]:
        print("I am fine.")
    elif question in ["are you working", "are you doing any job"]:
        print("Yes, I am working in TCS.")
    elif question in ["what is your name"]:
        print("My name is Sudha.")
        name = input("Enter your name: ")
```

```
        print("Nice name & nice to meet you,", name)
    elif question in ["what did you do yesterday"]:
        print("I was doing work at my house.")
    elif question in ["quit"]:
        break
    else:
        print("I don't understand what you said")
```

OUTPUT

```
Simple question & answering program
You may ask any of these questions:
hi
how are you
are you working
what is your name
what did you do yesterday
quit
Enter one question from above list:
```

TOWER OF HANOI

```
def tower_of_hanoi(n, from_rod, to_rod, aux_rod):
    if n == 0:
        return
    tower_of_hanoi(n-1, from_rod, aux_rod, to_rod)
    print("disk", n, "moved from", from_rod, "to", to_rod)
    tower_of_hanoi(n-1, aux_rod, to_rod, from_rod)
if __name__ == "__main__":
    n = 3
    tower_of_hanoi(n, 'A', 'C', 'B')
```

OUTPUT

```
disk 1 moved from A to C
disk 2 moved from A to B
disk 1 moved from C to B
disk 3 moved from A to C
disk 1 moved from B to A
disk 2 moved from B to C
disk 1 moved from A to C
```

HANGMAN

```
import string
word = "apple"
guesses = 8
used = []
print("Welcome to the game Hangman!")
print(f"I am thinking of a word that is {len(word)} letters long.")
print("---")
while guesses > 0:
    display = ''.join(c if c in used else '_' for c in word)
    if display == word:
        print("Congratulations, you won!")
        break
    print(f"You have {guesses} guesses left.")
    available = [c for c in string.ascii_lowercase if c not in used]
    print("Available Letters:", '.'.join(available))
    letter = input("Please guess a letter: ").lower(
    if letter in used:
        print(f"Oops! You've already guessed that letter: {display}")
        continue
    used.append(letter)
    if letter in word:
        new_display = ''.join(c if c in used else '_' for c in word)
```

```
        print(f"Good guess: {new_display}")
    else:
        guesses -= 1
        new_display = ''.join(c if c in used else '_' for c in word)
        print(f"Oops! That letter is not in my word: {new_display}")
print("---")
if display != word:
    print(f"Sorry, you ran out of guesses. The word was {word}")
```

OUTPUT

Welcome to the game Hangman!

I am thinking of a word that is 5 letters long.

You have 8 guesses left.

Available Letters: a.b.c.d.e.f.g.h.i.j.k.l.m.n.o.p.q.r.s.t.u.v.w.x.y.z

Please guess a letter:Good guess: a____

You have 8 guesses left.

Available Letters: b.c.d.e.f.g.h.i.j.k.l.m.n.o.p.q.r.s.t.u.v.w.x.y.z

Please guess a letter:Good guess: app__

You have 8 guesses left.

Available Letters: b.c.d.e.f.g.h.i.j.k.l.m.n.o.q.r.s.t.u.v.w.x.y.z

Please guess a letter:Good guess: appl_

You have 8 guesses left.

Available Letters: b.c.d.e.f.g.h.i.j.k.m.n.o.q.r.s.t.u.v.w.x.y.z

Please guess a letter: Good guess: apple

Congratulations, you won!

If wrong guess increases:

Oops! That letter is not in my word: _____

If guesses finish:

Sorry, you ran out of guesses. The word was apple

RANDOM FOREST

```
From sklearn.ensemble import RandomForestClassifier
X=[[1,2],[3,4],[5,6],[7,8]]
Y=[0,1,0,1]
forest_classifier = RandomForestClassifier()
forest_classifier.fit(X, y)
predictions = forest_classifier.predict(X)
print(predictions)
```

OUTPUT

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

BUILD SVM MODELS

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
iris = datasets.load_iris()
X = iris.data
y = iris.target
X = X[y != 2]
y = y[y != 2]
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
model = SVC(kernel='linear', C=1.0)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```

OUTPUT

Accuracy: 1.0

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	17
1	1.00	1.00	1.00	13
accuracy			1.00	30
macro avg	1.00	1.00	1.00	30
weighted avg	1.00	1.00	1.00	30

IMPLEMENT CLUSTERING ALGORITHM

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
X, y_true = make_blobs(
    n_samples=300,
    centers=3,
    cluster_std=0.60,
    random_state=0
)
kmeans = KMeans(n_clusters=3, random_state=0)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='viridis')
plt.scatter(
    kmeans.cluster_centers_[:, 0],
    kmeans.cluster_centers_[:, 1],
    s=200,
    c='red',
    label='Centroids'
)
plt.title("K-Means Clustering")
plt.legend()
```

```
plt.show()

print("\nResult:")

print("Thus the K-Means clustering algorithm was implemented successfully and the output was verified.")
```

OUTPUT