

**Tribhuvan University**

**INSTITUTE OF SCIENCE AND TECHNOLOGY**

**AMRIT SCIENCE CAMPUS**

**Lab Report On**

**Artificial Intelligence (CSC 266)**

**SUBMITTED BY:**

**KISMAT BHANDARI**

**SYMBOL NO: 28190/078**

**BSc. CSIT 2078**

**SUBMITTED TO:**

**Abhimanyu Yadav**

**Department of Computer Science & IT**

Table of Contents

[Lab 1: Write a program to Implement Breadth First Search using Python. 2](#_Toc163371847)

[Lab 2: Write a program to Implement Depth First Search using Python. 4](#_Toc163371848)

[Lab 3: Write a Program to Implement Tic-Tac-Toe game using Python. 6](#_Toc163371849)

[Lab 4: Write a program to implement WaterJug problem using Python. 9](#_Toc163371850)

[Lab 5: Write a Program to Implement Travelling Salesman Problem using Python. 12](#_Toc163371851)

[Lab 6: Write a python program to implement Tower of Hanoi. 14](#_Toc163371852)

[Lab 7: Write a program to implement N Queen program in Python. 16](#_Toc163371853)

[Lab 8: Write a program to implement Monkey Banana Problem using Python. 19](#_Toc163371854)

[Lab 9 Write a Program to Implement Naïve Bayes Algorithm using Python. 21](#_Toc163371855)

[Lab 10: Write a program to implement Backward Propagation Algorithm using Python 25](#_Toc163371856)

[Lab 11: Write a Program to Implement Genetics algorithm using Python 28](#_Toc163371857)

[Lab 12: Write a prolog program of family relation. 31](#_Toc163371858)

[Lab 13: Write a prolog program to perform an arithmetic operation. 33](#_Toc163371859)

[Lab 14: Write a prolog program to find factorial of a number. 34](#_Toc163371860)

[Lab 15: Write a prolog program to find the area and perimeter of a rectangle. 35](#_Toc163371861)

[Lab 16. Write a prolog program for Tower Of Hanoi. 36](#_Toc163371862)

[Lab 17. Write a prolog program to find GCD. 37](#_Toc163371863)

[Lab 18. Write a prolog program to find LCM. 38](#_Toc163371864)

[Lab 19: Write a program in prolog to find min and max. 39](#_Toc163371865)

[Lab 20. Write a prolog program for car\_owner problem. 40](#_Toc163371866)

## 

## Lab 1: Write a program to Implement Breadth First Search using Python.

**Algorithm**

* Start from the given source vertex.
* Add the source vertex to a queue.
* While the queue is not empty:
  + Remove a vertex from the front of the queue (dequeue operation).
  + Visit the removed vertex and mark it as visited.
  + Enqueue all adjacent vertices of the removed vertex that have not been visited yet.
* Repeat step 3 until the queue is empty.

**Source Code**

from collections import defaultdict, deque

class Graph:

def \_\_init\_\_(self):

self.graph = defaultdict(list)

def add(self, a, b):

self.graph[a].append(b)

def bfs(self, start):

visited = set()

queue = deque([start])

while queue:

vertex = queue.popleft()

if vertex not in visited:

print(vertex, end=' ')

visited.add(vertex)

queue.extend(self.graph[vertex])

if \_\_name\_\_ == "\_\_main\_\_":

g = Graph()

g.add(0,1)

g.add(0,2)

g.add(1,3)

g.add(1,4)

g.add(2,5)

g.add(2,6)

g.add(3,1)

g.add(4,1)

g.add(5,2)

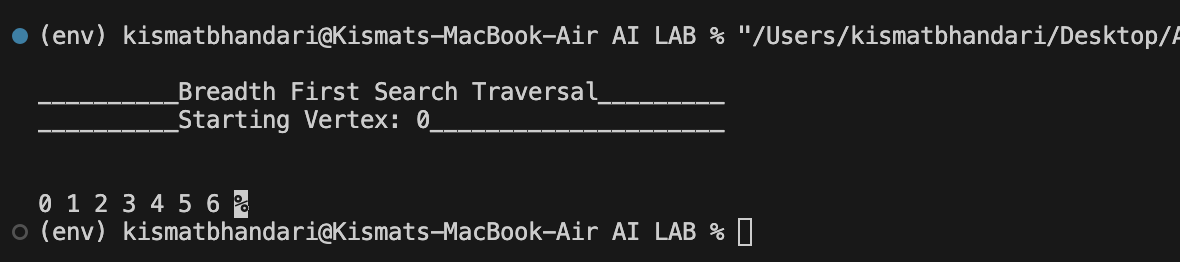
g.add(6,2)

print("\_\_\_\_\_\_\_\_\_\_Breadth First Search Traversal\_\_\_\_\_\_\_\_\_")

print("\_\_\_\_\_\_\_\_\_\_Starting Vertex: 0\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n\n")

g.bfs(0)

Output



## 

## Lab 2: Write a program to Implement Depth First Search using Python.

**Algorithm:**

* Start by putting any one of the graph's vertices on top of a stack.
* Take the top item of the stack and add it to the visited list.
* Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
* Keep repeating steps 2 and 3 until the stack is empty.

**Source Code:**

from collections import defaultdict

class Graph:

def \_\_init\_\_(self):

self.graph = defaultdict(list)

def add(self, u, v):

self.graph[u].append(v)

def dfs\_recursive(self, vertex, visited):

visited.add(vertex)

print(vertex, end=' ')

for neighbor in self.graph[vertex]:

if neighbor not in visited:

self.dfs\_recursive(neighbor, visited)

def dfs(self, start):

visited = set()

self.dfs\_recursive(start, visited)

if \_\_name\_\_ == "\_\_main\_\_":

g = Graph()

g.add(0,1)

g.add(0,2)

g.add(1,3)

g.add(1,4)

g.add(2,5)

g.add(2,6)

g.add(3,1)

g.add(4,1)

g.add(5,2)

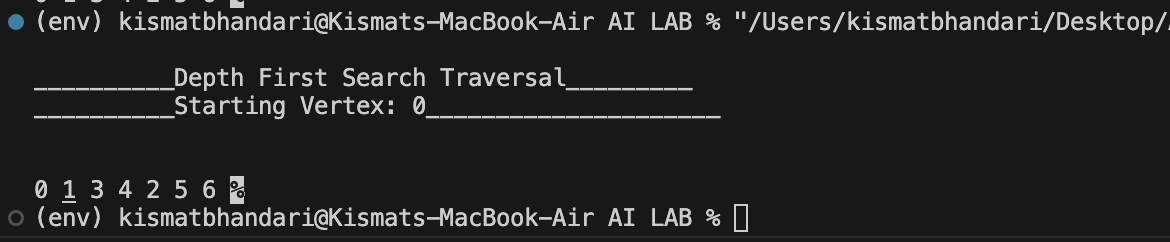
g.add(6,2)

print("\_\_\_\_\_\_\_\_\_\_Breadth First Search Traversal\_\_\_\_\_\_\_\_\_")

print("\_\_\_\_\_\_\_\_\_\_Starting Vertex: 0\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n\n")

g.dfs(0)

Output:

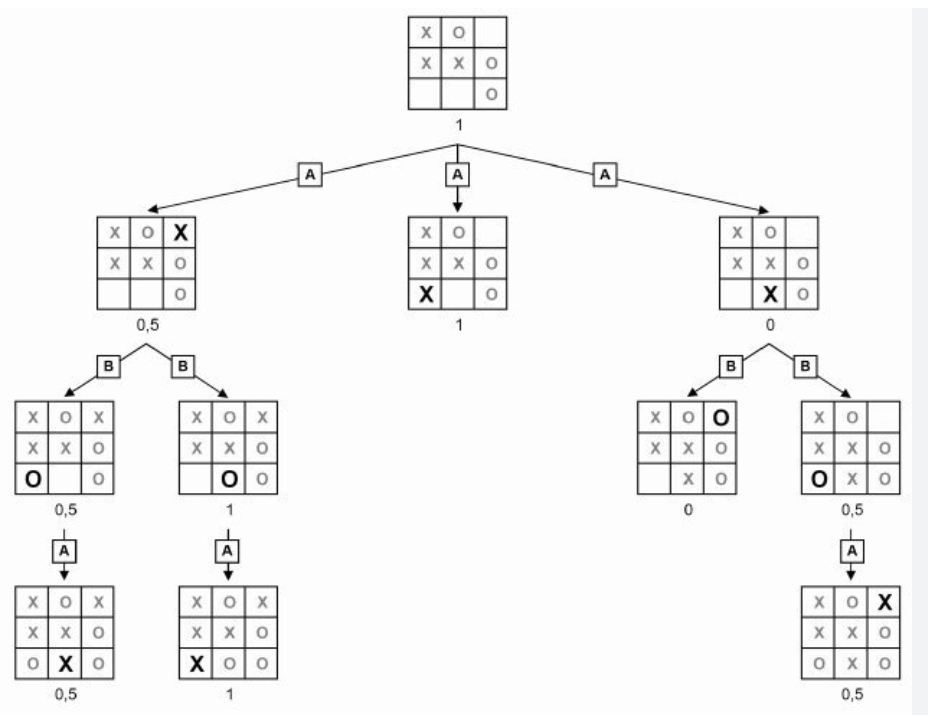


## Lab 3: Write a Program to Implement Tic-Tac-Toe game using Python.

**Introduction:**

Tic-Tac-Toe is a classic two-player game played on a 3x3 grid. The players take turns marking spaces in the grid with their respective symbols (usually "X" and "O") until one player gets three of their symbols in a row (horizontally, vertically, or diagonally), or until the grid is filled without a winner (resulting in a draw).

Space State Representation of Tic-Tac-Toe:



**Source Code:**

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 9)

def check\_winner(board, player):

if all(board[i][i] == player for i in range(3)) or all(board[i][2-i] == player for i in range(3)):

return True

for col in range(3):

if all(board[row][col] == player for row in range(3)):

return True

for row in board:

if all(cell == player for cell in row):

return True

return False

def tic\_tac\_toe():

board = [[" " for \_ in range(3)] for \_ in range(3)]

player = "X"

print("Tic-Tac-Toe")

print\_board(board)

for \_ in range(9):

row, col = map(int, input(f"Player {player}, enter your move (row[1-3] col[1-3]): ").split())

row -= 1

col -= 1

if board[row][col] != " ":

print("Cell already occupied. Retry")

continue

board[row][col] = player

print\_board(board)

if check\_winner(board, player):

print(f"Player {player} wins!")

return

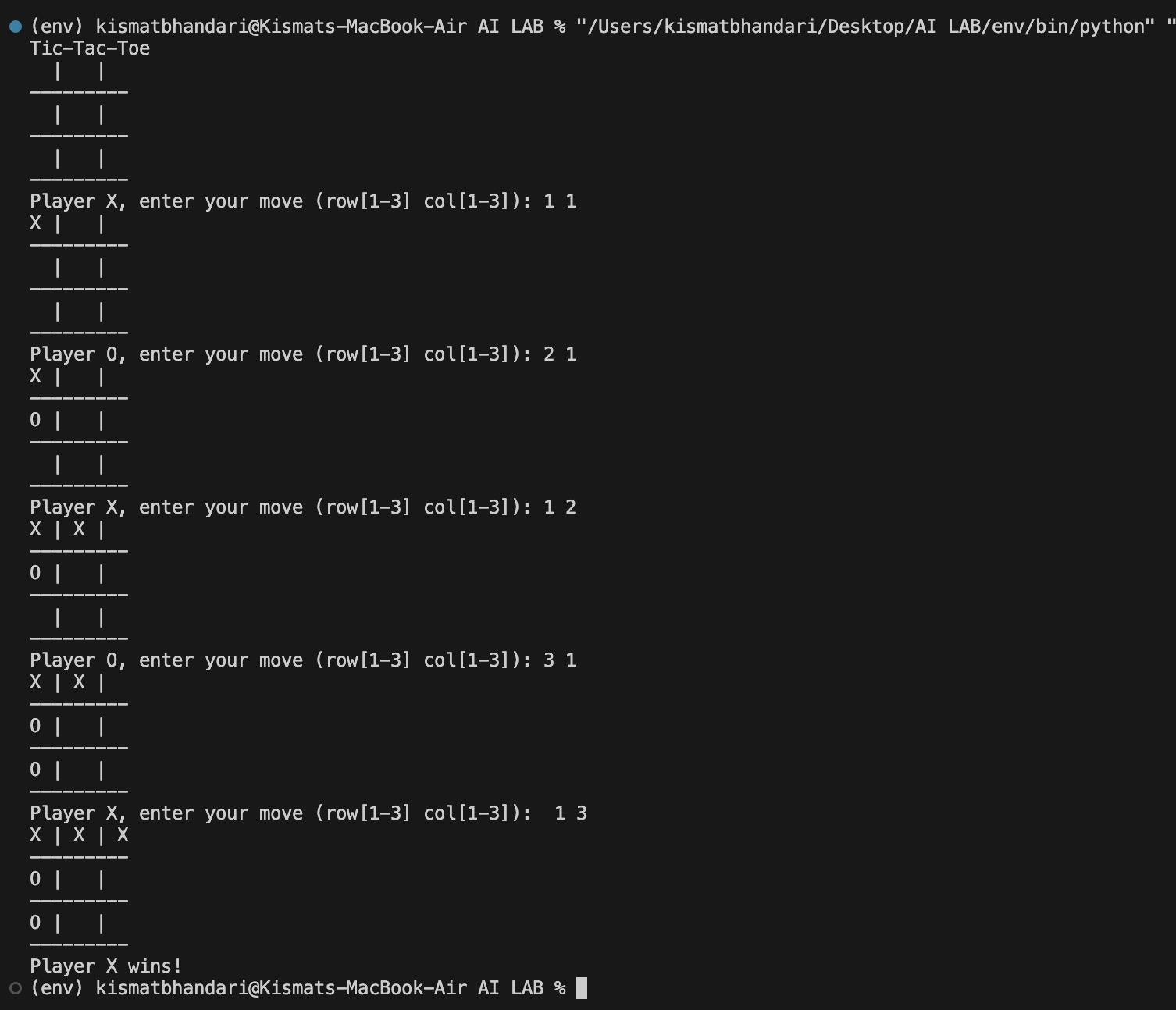
player = "O" if player == "X" else "X"

print("Draw!")

if \_\_name\_\_ == "\_\_main\_\_":

tic\_tac\_toe()

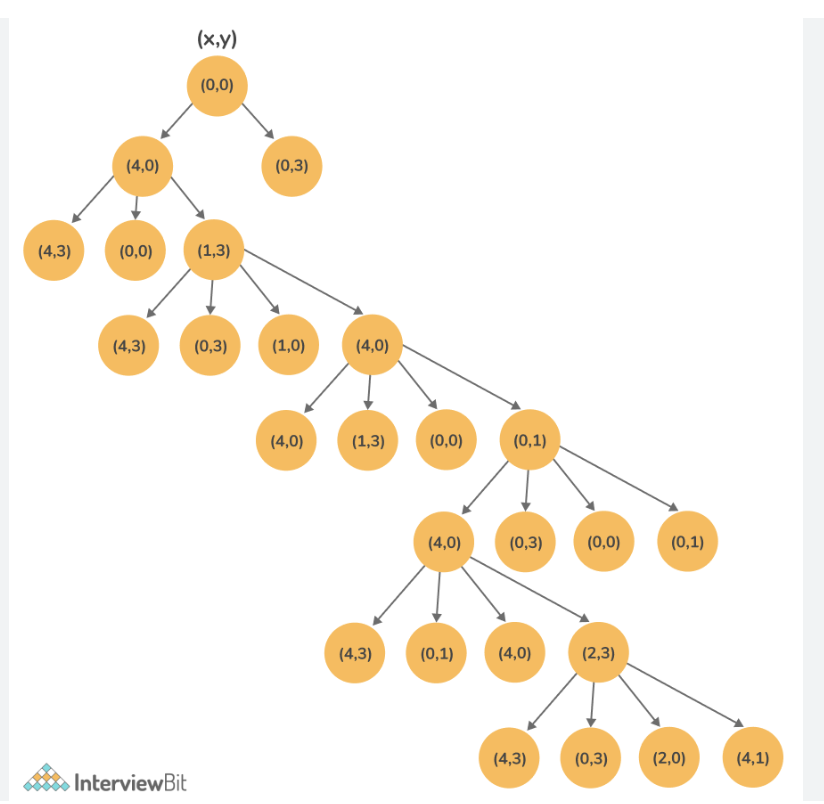
**Output**



## Lab 4: Write a program to implement WaterJug problem using Python.

**Introduction:**

You are given 2 jugs with the capacity 'm' and 'n' respectively. Initially, they are given empty. There is an unlimited supply of water. You can either fill the whole jug or a quantity that is less than the given capacity of jugs. Now, you are also given a third positive integer 'd'. Using the 2 given jugs, you need to come up with a solution to have 'd' amount of water in them and return the number of steps you took to reach that capacity.



**Source Code:**

from collections import defaultdict

jug1, jug2, aim = 6, 4, 2

visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):

if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

print(amt1, amt2)

return True

if visited[(amt1, amt2)] == False:

print(amt1, amt2)

visited[(amt1, amt2)] = True

return (waterJugSolver(0, amt2) or

waterJugSolver(amt1, 0) or

waterJugSolver(jug1, amt2) or

waterJugSolver(amt1, jug2) or

waterJugSolver(amt1 + min(amt2, (jug1-amt1)),

amt2 - min(amt2, (jug1-amt1))) or

waterJugSolver(amt1 - min(amt1, (jug2-amt2)),

amt2 + min(amt1, (jug2-amt2))))

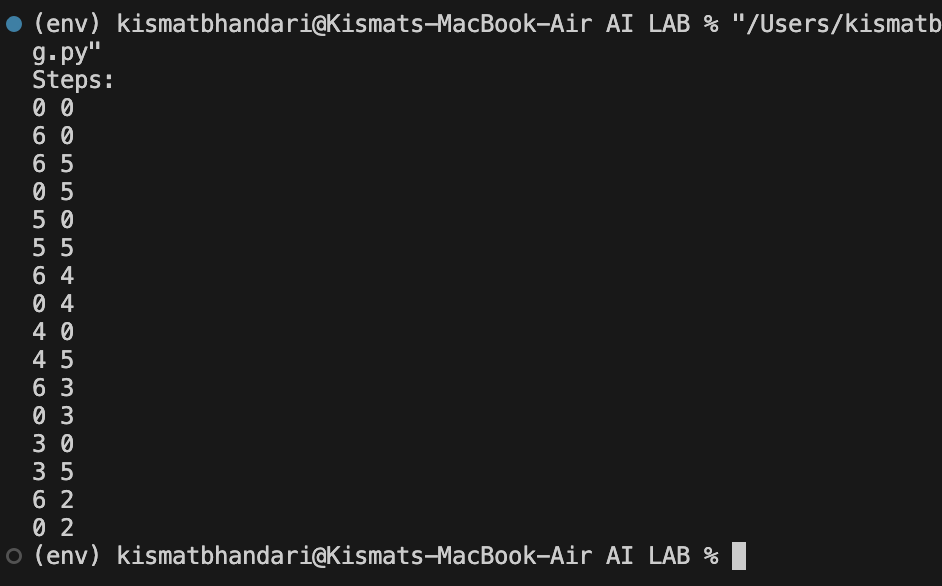
else:

return False

print("Steps: ")

waterJugSolver(0, 0)

**Output**



## Lab 5: Write a Program to Implement Travelling Salesman Problem using Python.

**Introduction:**

The Traveling Salesman Problem (TSP) is a classic problem in computer science and optimization, where the goal is to find the shortest possible route that visits every city exactly once and returns to the starting city. Solving TSP using brute force involves generating all possible permutations of the cities and calculating the total distance for each permutation to find the shortest route.

**Source Code:**

import itertools

def tsp(graph, start):

optimal\_path = None

num\_cities = len(graph)

all\_cities = set(range(num\_cities))

min\_distance = float('inf')

for path in itertools.permutations(range(1, num\_cities)):

distance = 0

current\_city = start

for next\_city in path:

distance += graph[current\_city][next\_city]

current\_city = next\_city

distance += graph[current\_city][start]

if distance < min\_distance:

min\_distance = distance

optimal\_path = (start,) + path + (start,)

return min\_distance, optimal\_path

if \_\_name\_\_ == "\_\_main\_\_":

graph = [

[0, 20, 42, 35, 25],

[20, 0, 30, 34, 42],

[42, 30, 0, 12, 28],

[35, 34, 12, 0, 15],

[25, 42, 28, 15, 0]

]

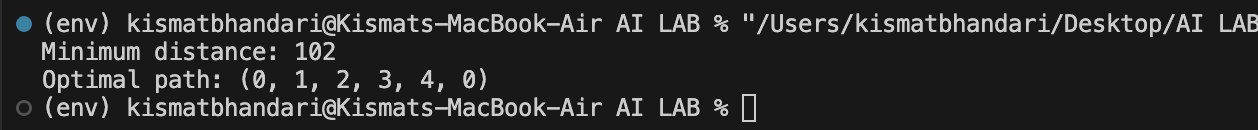
start\_city = 0

min\_distance, optimal\_path = tsp(graph, start\_city)

print(f"Minimum distance: {min\_distance}")

print(f"Optimal path: {optimal\_path}")

**Output:**



## 

## Lab 6: Write a python program to implement Tower of Hanoi.

**Introduction:**

The Tower of Hanoi is a classic problem in the field of computer science and mathematics, named after a legendary temple in India. It consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks stacked in ascending order of size on one rod, with the smallest disk at the top.

**Algorithm:**

* If there is only one disk to move, simply move it from the source rod to the target rod.
* Otherwise, recursively move the top n-1 disks from the source rod to the auxiliary rod using the target rod as the auxiliary.
* Move the n-th disk from the source rod to the target rod.
* Recursively move the n-1 disks from the auxiliary rod to the target rod using the source rod as the auxiliary.

**Source Code:**

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}")

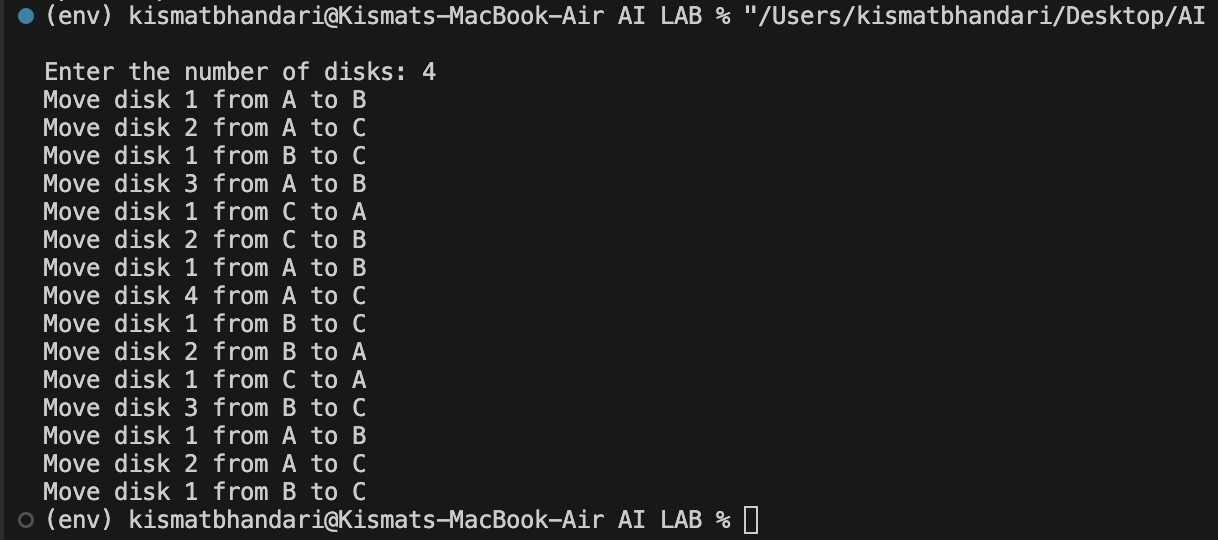
tower\_of\_hanoi(n-1, auxiliary, source, target)

if \_\_name\_\_ == "\_\_main\_\_":

n = int(input("Enter the number of disks: "))

tower\_of\_hanoi(n, 'A', 'B', 'C')

**Output:**

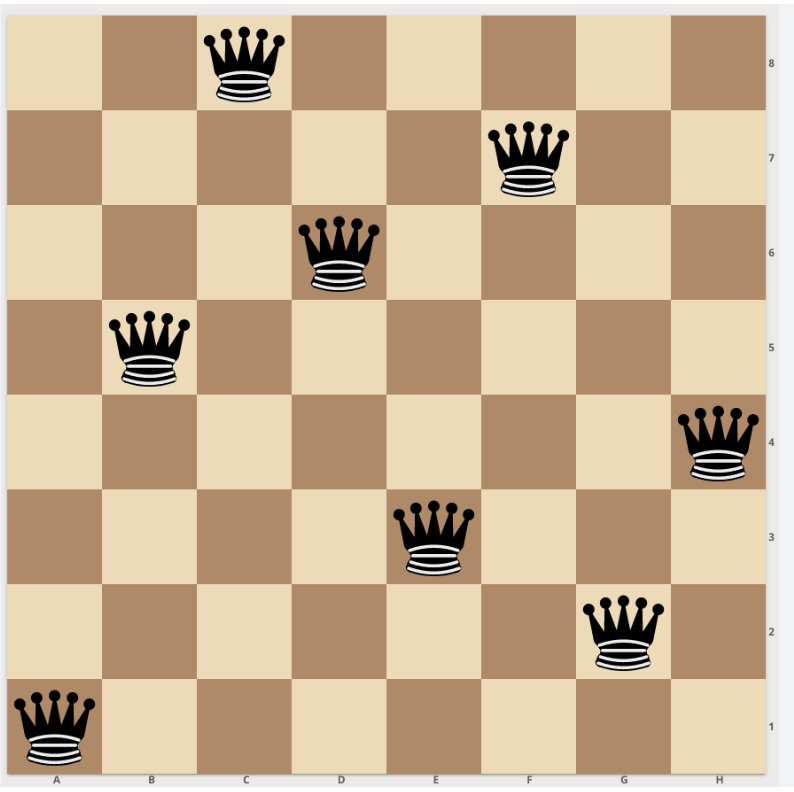


## Lab 7: Write a program to implement N Queen program in Python.

**Introduction:**

The N-Queens problem is a classic problem in combinatorial optimization and computer science. It involves placing N chess queens on an N×N chessboard so that no two queens threaten each other. In other words, no two queens can share the same row, column, or diagonal.

**Algorithm:**

* **Initialize**: Create a board of size NxN where each cell represents a square on the chessboard. Initially, all cells are empty.
* **Place Queens**: Start with the leftmost column. For each column:
  + Place a queen in the current column in each row, one by one.
  + Check if the queen placement is safe by ensuring that no other queen threatens the current queen horizontally, vertically, or diagonally.
  + If a safe placement is found, mark the cell as occupied by the queen.
  + Move to the next column and recursively repeat the process.
* **Backtracking**: If placing a queen in a certain row of the current column leads to a conflict (i.e., it's not safe), backtrack and try placing the queen in the next row of the current column.
* **Termination Condition**: If all N queens are successfully placed on the board, a solution is found. If not, backtrack until all possible combinations are exhausted.
* **Output Solution**: Once a solution is found, print the board configuration representing the placement of N queens
* 

**Source Code:**

global N

N = 5

def printSolution(board):

for i in range(N):

for j in range(N):

print (board[i][j],end=' ')

print()

def isSafe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, N, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solveNQUtil(board, col):

if col >= N:

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQUtil(board, col + 1) == True:

return True

board[i][col] = 0

return False

def solveNQueen():

board = [ [0, 0, 0, 0,0],

[0, 0, 0, 0, 0],

[0, 0, 0, 0, 0],

[0, 0, 0, 0, 0],

[0, 0, 0, 0, 0]

]

if solveNQUtil(board, 0) == False:

print ("Solution does not exist")

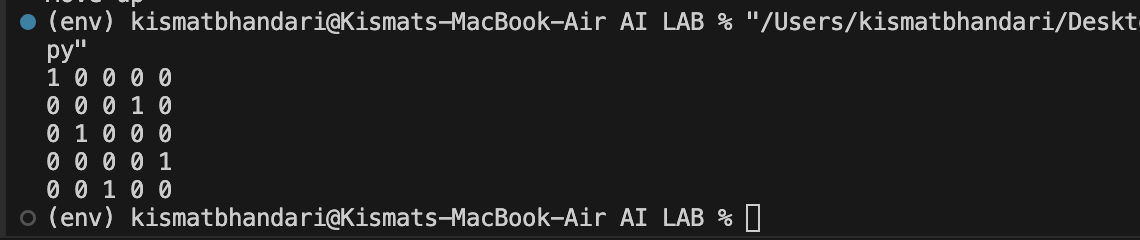
return False

printSolution(board)

return True

solveNQueen()

**Output:**



## Lab 8: Write a program to implement Monkey Banana Problem using Python.

**Introduction:**

A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey's reach. However, in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey?

**Source Code:**

def monkey\_banana():

monkey\_pos = (0, 0)

banana\_pos = (4, 4)

plan = []

# Move the monkey to the banana

while monkey\_pos != banana\_pos:

if monkey\_pos[0] < banana\_pos[0]:

monkey\_pos = (monkey\_pos[0] + 1, monkey\_pos[1])

plan.append("Move right")

elif monkey\_pos[0] > banana\_pos[0]:

monkey\_pos = (monkey\_pos[0] - 1, monkey\_pos[1])

plan.append("Move left")

if monkey\_pos[1] < banana\_pos[1]:

monkey\_pos = (monkey\_pos[0], monkey\_pos[1] + 1)

plan.append("Move up")

elif monkey\_pos[1] > banana\_pos[1]:

monkey\_pos = (monkey\_pos[0], monkey\_pos[1] - 1)

plan.append("Move down")

print("Monkey's plan to reach the banana:")

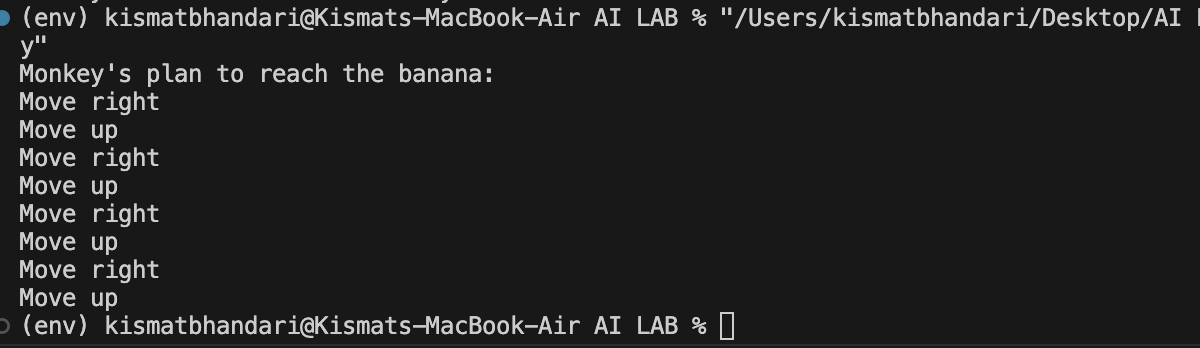
for action in plan:

print(action)

if \_\_name\_\_ == "\_\_main\_\_":

monkey\_banana()

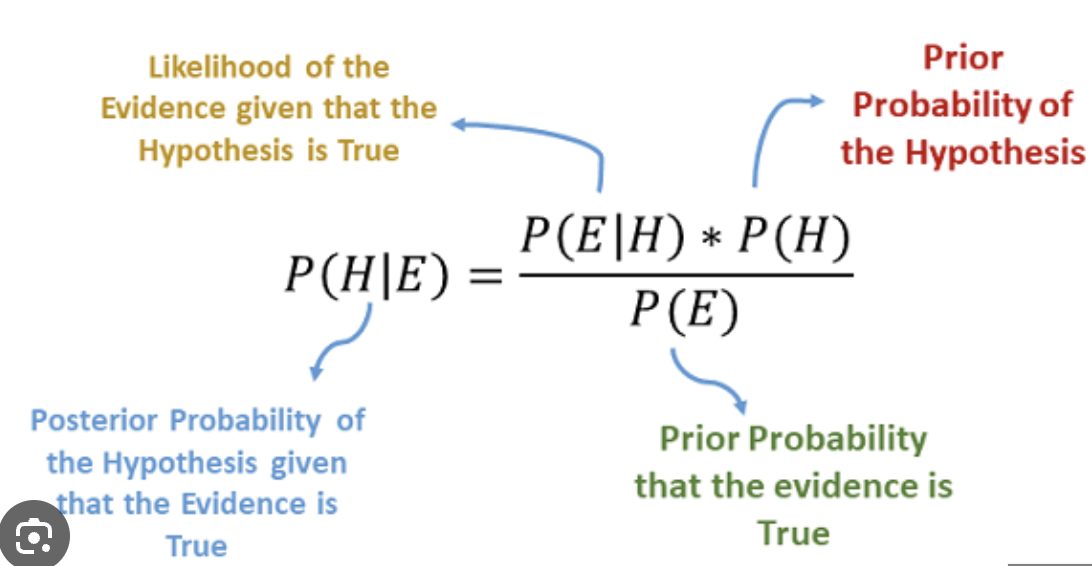
**Output:**



## Lab 9 Write a Program to Implement Naïve Bayes Algorithm using Python.

**Introduction:**

* Naïve Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems.
* It is mainly used in *text classification* that includes a high-dimensional training dataset.
* Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.



**Source Code:**

import numpy as np

class NaiveBayes:

def fit(self, X, y):

n\_samples, n\_features = X.shape

self.\_classes = np.unique(y)

n\_classes = len(self.\_classes)

# calculate mean, var, and prior for each class

self.\_mean = np.zeros((n\_classes, n\_features), dtype=np.float64)

self.\_var = np.zeros((n\_classes, n\_features), dtype=np.float64)

self.\_priors = np.zeros(n\_classes, dtype=np.float64)

for idx, c in enumerate(self.\_classes):

X\_c = X[y == c]

self.\_mean[idx, :] = X\_c.mean(axis=0)

self.\_var[idx, :] = X\_c.var(axis=0)

self.\_priors[idx] = X\_c.shape[0] / float(n\_samples)

def predict(self, X):

y\_pred = [self.\_predict(x) for x in X]

return np.array(y\_pred)

def \_predict(self, x):

posteriors = []

# calculate posterior probability for each class

for idx, c in enumerate(self.\_classes):

prior = np.log(self.\_priors[idx])

posterior = np.sum(np.log(self.\_pdf(idx, x)))

posterior = posterior + prior

posteriors.append(posterior)

# return class with the highest posterior

return self.\_classes[np.argmax(posteriors)]

def \_pdf(self, class\_idx, x):

mean = self.\_mean[class\_idx]

var = self.\_var[class\_idx]

numerator = np.exp(-((x - mean) \*\* 2) / (2 \* var))

denominator = np.sqrt(2 \* np.pi \* var)

return numerator / denominator

# Testing

if \_\_name\_\_ == "\_\_main\_\_":

# Imports

from sklearn.model\_selection import train\_test\_split

from sklearn import datasets

def accuracy(y\_true, y\_pred):

accuracy = np.sum(y\_true == y\_pred) / len(y\_true)

return accuracy

X, y = datasets.make\_classification(

n\_samples=1000, n\_features=10, n\_classes=2, random\_state=123

)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.2, random\_state=123

)

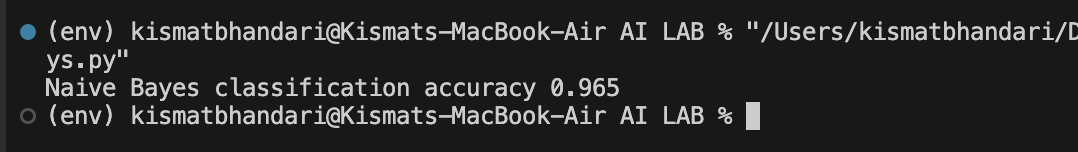
nb = NaiveBayes()

nb.fit(X\_train, y\_train)

predictions = nb.predict(X\_test)

print("Naive Bayes classification accuracy", accuracy(y\_test, predictions))

**Output:**



## 

## Lab 10: Write a program to implement Backward Propagation Algorithm using Python

**Introduction:**

Backpropagation (short for "backward propagation of errors") is a fundamental algorithm used in training artificial neural networks, particularly in the context of supervised learning. It's a type of gradient-based optimization algorithm that adjusts the weights of the connections between neurons in a neural network in order to minimize the difference between the predicted outputs and the actual targets.

**Source Code:**

Backward propagation algorithm to estimated marks of student based on hours sleep and hour studied.

import numpy as np

# X = (hours sleeping, hours studying), y = test score of the student

X = np.array(([4, 5], [6, 7], [8, 9]), dtype=float)

y = np.array(([65], [84], [89]), dtype=float)

# scale units

X = X/np.amax(X, axis=0) #maximum of X array

y = y/100 # maximum test score is 100

class NeuralNetwork(object):

def \_\_init\_\_(self):

#parameters

self.inputSize = 2

self.outputSize = 1

self.hiddenSize = 3

#weights

self.W1 = np.random.randn(self.inputSize, self.hiddenSize) # (3x2) weight matrix from input to hidden layer

self.W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1) weight matrix from hidden to output layer

def feedForward(self, X):

#forward propogation through the network

self.z = np.dot(X, self.W1) #dot product of X (input) and first set of weights (3x2)

self.z2 = self.sigmoid(self.z) #activation function

self.z3 = np.dot(self.z2, self.W2) #dot product of hidden layer (z2) and second set of weights (3x1)

output = self.sigmoid(self.z3)

return output

def sigmoid(self, s, deriv=False):

if (deriv == True):

return s \* (1 - s)

return 1/(1 + np.exp(-s))

def backward(self, X, y, output):

#backward propogate through the network

self.output\_error = y - output # error in output

self.output\_delta = self.output\_error \* self.sigmoid(output, deriv=True)

self.z2\_error = self.output\_delta.dot(self.W2.T) #z2 error: how much our hidden layer weights contribute to output error

self.z2\_delta = self.z2\_error \* self.sigmoid(self.z2, deriv=True) #applying derivative of sigmoid to z2 error

self.W1 += X.T.dot(self.z2\_delta) # adjusting first set (input -> hidden) weights

self.W2 += self.z2.T.dot(self.output\_delta) # adjusting second set (hidden -> output) weights

def train(self, X, y):

output = self.feedForward(X)

self.backward(X, y, output)

NN = NeuralNetwork()

for i in range(1000): #trains the NN 1000 times

if (i % 100 == 0):

print("Loss: " + str(np.mean(np.square(y - NN.feedForward(X)))))

NN.train(X, y)

print("Input: " + str(X))

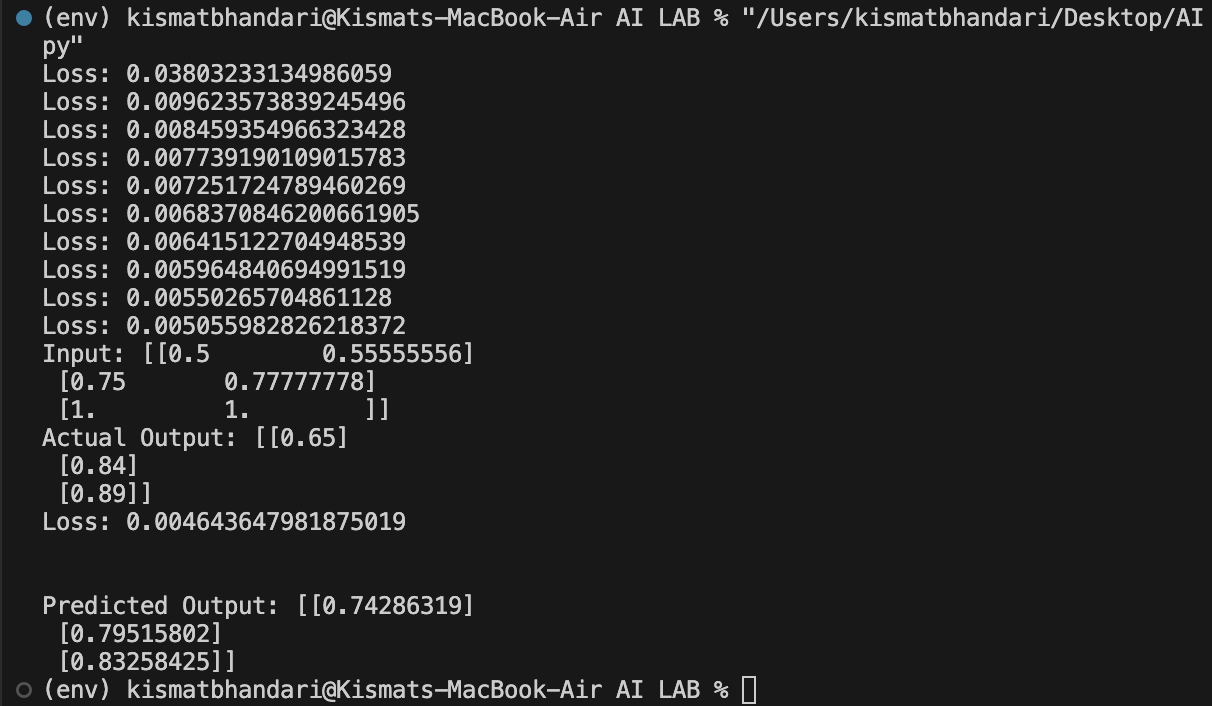
print("Actual Output: " + str(y))

print("Loss: " + str(np.mean(np.square(y - NN.feedForward(X)))))

print("\n")

print("Predicted Output: " + str(NN.feedForward(X)))

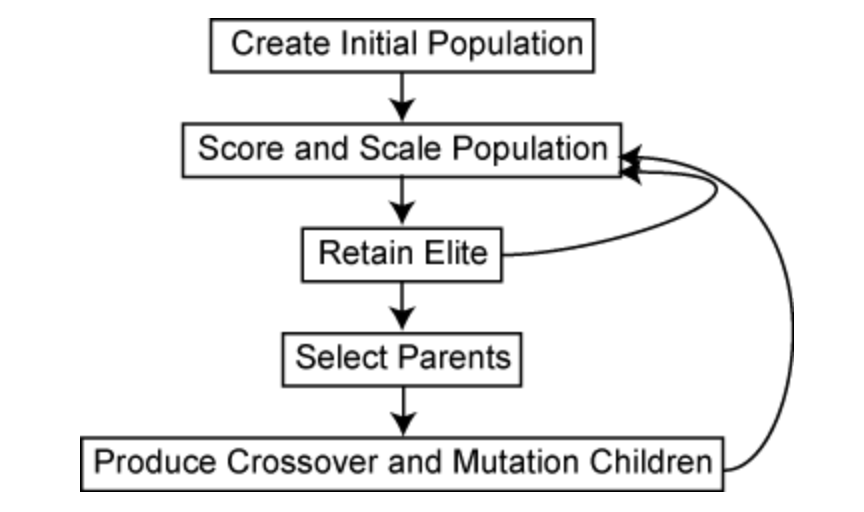
Output:



## Lab 11: Write a Program to Implement Genetics algorithm using Python

**Introduction:**

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals from the current population to be parents and uses them to produce the children for the next generation.



**Source Code:**

Genetic algorithm to find value of equation:

import numpy as np

# Define the function to maximize

def objective\_function(x):

return x \* np.sin(10 \* np.pi \* x) + 1

# Genetic algorithm parameters

population\_size = 500

num\_generations = 1000

mutation\_rate = 0.01

# Initialize the population with random values

population = np.random.uniform(0, 1, size=population\_size)

for generation in range(num\_generations):

# Evaluate fitness for each individual

fitness = objective\_function(population)

# Select parents (tournament selection)

num\_parents = 10

parent\_indices = np.random.choice(population\_size, size=num\_parents, replace=False)

parents = population[parent\_indices]

# Create offspring through crossover

offspring = np.mean(parents)

# Apply mutation

if np.random.rand() < mutation\_rate:

offspring += np.random.normal(scale=0.1)

# Replace the least fit individual with the offspring

worst\_index = np.argmin(fitness)

population[worst\_index] = offspring

# Find the best solution

best\_index = np.argmax(fitness)

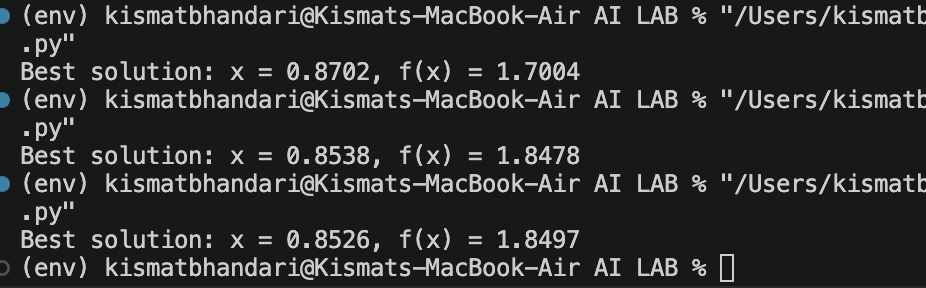
best\_x = population[best\_index]

best\_value = objective\_function(best\_x)

print(f"Best solution: x = {best\_x:.4f}, f(x) = {best\_value:.4f}")

**Output:**

**Same program is runned three times:**



## Lab 12: Write a prolog program of family relation.

**Source Code:**

parent(bhojraj, kismat).

parent(gyanu, kismat).

parent(bhojraj, rimisha).

parent(gyanu, rimisha).

father(bhojraj, kismat).

father(bhojraj, rimisha).

mother(gyanu, kismat).

mother(gyanu, rimisha).

son(X, Y) :- parent(Y, X), male(X).

daughter(X, Y):- parent(Y,X), female(X).

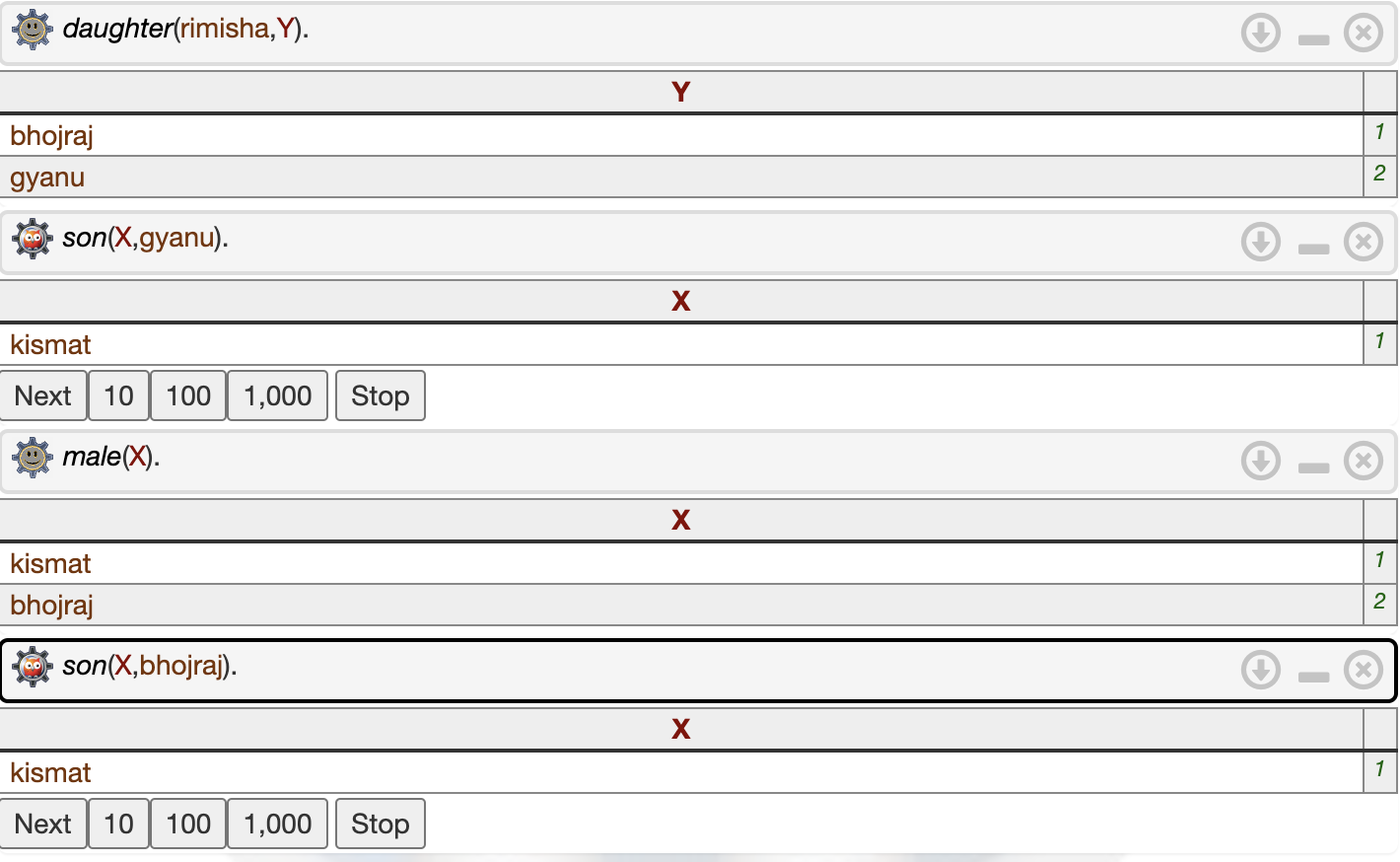
male(kismat).

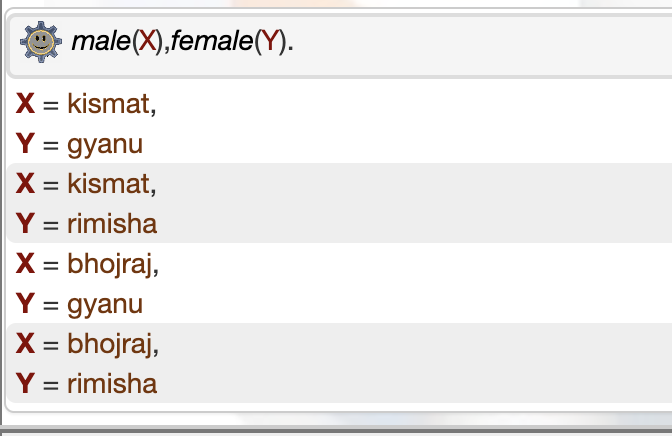
male(bhojraj).

female(gyanu).

female(rimisha).

**Output:**





## Lab 13: Write a prolog program to perform an arithmetic operation.

**Source Code:**

arithmetic:-

write('Number 1: '),

read(X), nl,

write('Number 2: '),

read(Y), nl,

A is X+Y,

S is X-Y,

P is X\*Y,

D is X/Y,

Q is X//Y,

write('Sum: '), write(A),nl,

write('Difference: '), write(S),nl,

write('Product: '), write(P),nl,

write('Division: '), write(D),nl,

write('Quotient: '), write(Q).

**Output:**

A screenshot of a computer

Description automatically generated

## Lab 14: Write a prolog program to find factorial of a number.

**Source Code:**

factorial(0, 1).

factorial(N, Result) :-

N > 0,

N1 is N - 1,

factorial(N1, SubResult),

Result is N \* SubResult.

**Output:**

A screenshot of a social media post

Description automatically generated

## Lab 15: Write a prolog program to find the area and perimeter of a rectangle.

**Source Code:**

rectangle\_properties :-

write('Enter the length of the rectangle: '),

read(Length),

write('Enter the width of the rectangle: '),

read(Width),

Area is Length \* Width,

Perimeter is 2 \* (Length + Width),

write('The area of the rectangle is '), write(Area), nl,

write('The perimeter of the rectangle is '), write(Perimeter), nl.

**Output:**

A white and grey text box

Description automatically generated with medium confidence

## Lab 16. Write a prolog program for Tower Of Hanoi.

**Source Code:**

toh(1,X,Y,\_):-

write('Move top disk from'), write(' '),

write(X), write(' to '), write(Y), nl.

toh(N,X,Y,Z):-

N>1,

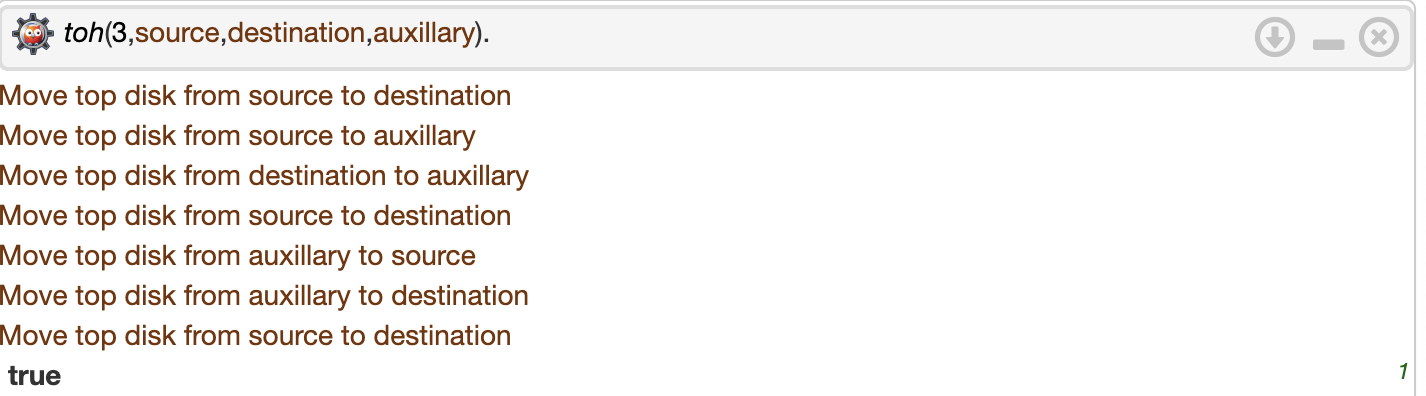
M is N-1,

toh(M,X,Z,Y),

toh(1,X,Y,\_),

toh(M,Z,Y,X).

**Output:**



## Lab 17. Write a prolog program to find GCD.

**Source Code:**

gcd(X, 0, X) :- X > 0.

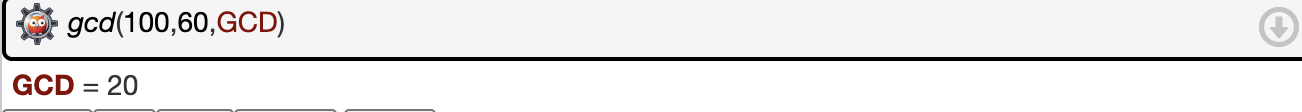
gcd(X, Y, G) :-

Y > 0,

Z is X mod Y,

gcd(Y, Z, G).

**Output:**



## Lab 18. Write a prolog program to find LCM.

**Source Code:**

gcd(X, 0, X) :- X > 0.

gcd(X, Y, G) :-

Y > 0,

Z is X mod Y,

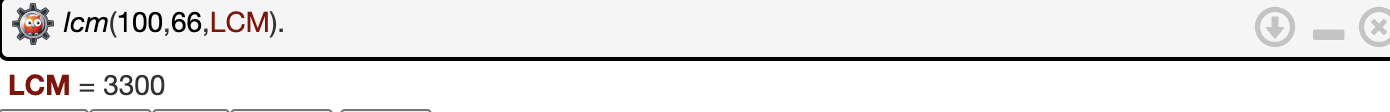
gcd(Y, Z, G).

lcm(X, Y, L) :-

gcd(X, Y, G),

L is (X \* Y) / G.

**Output:**



## Lab 19: Write a program in prolog to find min and max.

**Source Code:**

find\_max(X, Y, X) :- X >= Y.

find\_max(X, Y, Y) :- X < Y.

find\_min(X, Y, X) :- X =< Y.

find\_min(X, Y, Y) :- X > Y.

**Output:**



## Lab 20. Write a prolog program for car\_owner problem.

**Source Code:**

owns(kismat,car(bmw)).

owns(bhojraj,car(nissan)).

owns(saurav,car(ferrai)).

owns(rimisha,car(nissan)).

sedan(car(bmw)).

sedan(car(ferrai)).

truck(car(nissan)).

**Output:**

A close-up of a white rectangular object

Description automatically generatedA website with a website address

Description automatically generated with medium confidence