Q1 :Implement Backpropagation in Python

==>

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(x):
    return x * (1 - x)


inputs = np.array([[0, 0],
                   [0, 1],
                   [1, 0],
                   [1, 1]])


expected_output = np.array([[0], [1], [1], [0]])


input_layer_neurons = 2
hidden_layer_neurons = 2
output_neurons = 1


hidden_weights = np.random.uniform(size=(input_layer_neurons,
hidden_layer_neurons))
```

```
hidden_bias = np.random.uniform(size=(1, hidden_layer_neurons))
output_weights = np.random.uniform(size=(hidden_layer_neurons,
output_neurons))
output_bias = np.random.uniform(size=(1, output_neurons))

lr = 0.1
epochs = 10000

for epoch in range(epochs):
    # Forward Propagation
    hidden_layer_activation = np.dot(inputs, hidden_weights)
    hidden_layer_activation += hidden_bias
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output, output_weights)
    output_layer_activation += output_bias
    predicted_output = sigmoid(output_layer_activation)

    # Backpropagation
    error = expected_output - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)
```

```
    error_hidden_layer = d_predicted_output.dot(output_weights.T)
    d_hidden_layer = error_hidden_layer *
sigmoid_derivative(hidden_layer_output)

    # Updating Weights and Biases
    output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
    output_bias += np.sum(d_predicted_output, axis=0, keepdims=True) * lr
    hidden_weights += inputs.T.dot(d_hidden_layer) * lr
    hidden_bias += np.sum(d_hidden_layer, axis=0, keepdims=True) * lr

print("Training complete")
print("Output from neural network after 10,000 epochs: \n", predicted_output)
```

Q.2:.Buildbsimple Neural network in Python from Keras.

➔

Bash

```
pip install tensorflow
```

Python

```
import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist


# Load and preprocess the data

(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()

train_images = train_images.reshape((60000, 28, 28,
1)).astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32')
/ 255


# Build the model

model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28,
28, 1)))
```

```
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten())

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(10, activation='softmax'))


# Compile the model

model.compile(optimizer='adam',

        loss='sparse_categorical_crossentropy',

        metrics=['accuracy'])


# Train the model

model.fit(train_images, train_labels, epochs=5, batch_size=64)


# Evaluate the model

test_loss, test_acc = model.evaluate(test_images, test_labels)

print(f'Test accuracy: {test_acc}')
```

Q.3: **Write a program to implement Fuzzy Operations** [

Union

Intersection

Complement

➔

```
class FuzzySet:
    def __init__(self, elements):
        self.elements = elements  # elements is a dictionary with element: membership value

    def union(self, other_set):
        union_elements = {}
        for key in self.elements.keys():
            union_elements[key] = max(self.elements[key], other_set.elements.get(key, 0))
        for key in other_set.elements.keys():
            if key not in union_elements:
                union_elements[key] = max(other_set.elements[key], self.elements.get(key, 0))
        return FuzzySet(union_elements)

    def intersection(self, other_set):
```

```
        intersection_elements = {}
        for key in self.elements.keys():
            if key in other_set.elements:
                intersection_elements[key] = min(self.elements[key], 
other_set.elements[key])
        return FuzzySet(intersection_elements)

    def complement(self):
        complement_elements = {key: 1 - value for key, value in 
self.elements.items()}
        return FuzzySet(complement_elements)

    def __str__(self):
        return str(self.elements)


# Example usage:
setA = FuzzySet({'a': 0.5, 'b': 0.2, 'c': 0.8})
setB = FuzzySet({'a': 0.7, 'b': 0.4, 'd': 0.6})
```

```
print("Set A:", setA)
print("Set B:", setB)

unionAB = setA.union(setB)
print("Union of A and B:", unionAB)

intersectionAB = setA.intersection(setB)
print("Intersection of A and B:", intersectionAB)

complementA = setA.complement()
print("Complement of A:", complementA)
```

Q.4:Write a program to implement Max-Min Composition and Max-Product Composition.

```
import numpy as np


class FuzzyRelation:
    def __init__(self, matrix):
        self.matrix = np.array(matrix)


    def max_min_composition(self, other):
        result = np.zeros((self.matrix.shape[0], other.matrix.shape[1]))
        for i in range(self.matrix.shape[0]):
            for j in range(other.matrix.shape[1]):
                result[i, j] = np.max(np.minimum(self.matrix[i, :],
other.matrix[:, j]))
        return FuzzyRelation(result)


    def max_product_composition(self, other):
        result = np.zeros((self.matrix.shape[0], other.matrix.shape[1]))
        for i in range(self.matrix.shape[0]):
            for j in range(other.matrix.shape[1]):
```

```
            result[i, j] = np.max(self.matrix[i, :] * other.matrix[:, j])
        return FuzzyRelation(result)

    def __str__(self):
        return str(self.matrix)

def input_matrix(rows, cols):
    matrix = []
    for i in range(rows):
        row = []
        for j in range(cols):
            value = float(input(f"Enter value for element [{i+1},{j+1}]: "))
            row.append(value)
        matrix.append(row)
    return matrix

# Example usage:
print("Enter the size of the first relation matrix:")
rows1 = int(input("Number of rows: "))
cols1 = int(input("Number of columns: "))
```

```
print("Enter the elements for the first relation matrix:")
matrix1 = input_matrix(rows1, cols1)
relation1 = FuzzyRelation(matrix1)

print("Enter the size of the second relation matrix:")
rows2 = int(input("Number of rows: "))
cols2 = int(input("Number of columns: "))
print("Enter the elements for the second relation matrix:")
matrix2 = input_matrix(rows2, cols2)
relation2 = FuzzyRelation(matrix2)

print("\nRelation 1:")
print(relation1)

print("Relation 2:")
print(relation2)

max_min_result = relation1.max_min_composition(relation2)
print("\nMax-Min Composition:")
print(max_min_result)
```

```
max_product_result = relation1.max_product_composition(relation2)

print("\nMax-Product Composition:")

print(max_product_result)
```

Q.5:Write python program to study and analyse genetic life cycle

```
import random


class Individual:
    def __init__(self, genes):
        self.genes = genes
        self.fitness = self.calculate_fitness()

    def calculate_fitness(self):
        # Fitness function: count the number of 1s in the genes
        return self.genes.count('1')

    @staticmethod
    def create_random(length):
        genes = ''.join(random.choice('01') for _ in range(length))
        return Individual(genes)

    def __str__(self):
        return f"Genes: {self.genes}, Fitness: {self.fitness}"
```

```
def selection(population):
    population.sort(key=lambda x: x.fitness, reverse=True)
    return population[:2]  # Select top 2 individuals


def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1.genes) - 1)
    child1_genes = parent1.genes[:crossover_point] +
parent2.genes[crossover_point:]
    child2_genes = parent2.genes[:crossover_point] +
parent1.genes[crossover_point:]
    return Individual(child1_genes), Individual(child2_genes)


def mutate(individual, mutation_rate):
    new_genes = list(individual.genes)
    for i in range(len(new_genes)):
        if random.random() < mutation_rate:
            new_genes[i] = '1' if new_genes[i] == '0' else '0'
    return Individual(''.join(new_genes))
```

```
def genetic_algorithm(population_size, gene_length, generations,
mutation_rate):
    population = [Individual.create_random(gene_length) for _ in
range(population_size)]
    for generation in range(generations):
        print(f"Generation {generation + 1}")
        for individual in population:
            print(individual)
        selected = selection(population)
        children = []
        while len(children) < population_size:
            parent1, parent2 = random.sample(selected, 2)
            child1, child2 = crossover(parent1, parent2)
            children.append(mutate(child1, mutation_rate))
            if len(children) < population_size:
                children.append(mutate(child2, mutation_rate))
        population = children
        print()


# Taking user input
```

```
population_size = int(input("Enter population size: "))
gene_length = int(input("Enter gene length: "))
generations = int(input("Enter number of generations: "))
mutation_rate = float(input("Enter mutation rate (e.g., 0.1 for 10%): "))

genetic_algorithm(population_size, gene_length, generations,
mutation_rate)
```

```
def complement(universal_set, subset):
    return universal_set - subset

def de_morgan_union(A, B, universal_set):
    left_side = complement(universal_set, A.union(B))
    right_side = complement(universal_set,
A).intersection(complement(universal_set, B))
```

```
    return left_side, right_side


def de_morgan_intersection(A, B, universal_set):
    left_side = complement(universal_set, A.intersection(B))
    right_side = complement(universal_set,
A).union(complement(universal_set, B))
    return left_side, right_side


def input_set(prompt):
    return set(map(int, input(prompt).split()))


def main():
    print("Enter the elements of the universal set (space-separated
integers):")
    universal_set = input_set("> ")


    print("Enter the elements of set A (space-separated integers):")
    A = input_set("> ")


    print("Enter the elements of set B (space-separated integers):")
```

```
    B = input_set("> ")

    # De Morgan's Law for Union
    union_left, union_right = de_morgan_union(A, B, universal_set)
    print(f"\nDe Morgan's Law for Union: (A ∪ B)′ = A′ ∩ B′")
    print(f"Left Side: {union_left}")
    print(f"Right Side: {union_right}")
    print(f"Law holds: {union_left == union_right}")

    # De Morgan's Law for Intersection
    intersection_left, intersection_right = de_morgan_intersection(A, B,
universal_set)
    print(f"\nDe Morgan's Law for Intersection: (A ∩ B)′ = A′ ∪ B′")
    print(f"Left Side: {intersection_left}")
    print(f"Right Side: {intersection_right}")
    print(f"Law holds: {intersection_left == intersection_right}")

if __name__ == "__main__":
    main()
```

Q 6.:Write aprogram to implement Fuzzy Operations

Algebraic sum

Algebraic product

Cartesian product

➔

```
class FuzzySet:
    def __init__(self, elements):
        self.elements = elements  # elements is a dictionary with element:
membership value


    def algebraic_sum(self, other_set):
        result = {key: self.elements.get(key, 0) +
other_set.elements.get(key, 0) - self.elements.get(key, 0) *
other_set.elements.get(key, 0) for key in set(self.elements) |
set(other_set.elements)}
        return FuzzySet(result)


    def algebraic_product(self, other_set):
        result = {key: self.elements.get(key, 0) *
other_set.elements.get(key, 0) for key in set(self.elements) |
set(other_set.elements)}
        return FuzzySet(result)
```

```
    def cartesian_product(self, other_set):
        result = {(key1, key2): min(self.elements[key1],
other_set.elements[key2]) for key1 in self.elements for key2 in
other_set.elements}
        return result

    def __str__(self):
        return str(self.elements)

def input_fuzzy_set(prompt):
    elements = input(prompt).split()
    fuzzy_set = {}
    for element in elements:
        key, value = element.split(':')
        fuzzy_set[key] = float(value)
    return FuzzySet(fuzzy_set)

def main():
    print("Enter elements of Fuzzy Set A (format:
element:membership_value, space-separated):")
```

```
    setA = input_fuzzy_set("> ")

    print("Enter elements of Fuzzy Set B (format:
element:membership_value, space-separated):")
    setB = input_fuzzy_set("> ")

    algebraic_sum = setA.algebraic_sum(setB)
    print("\nAlgebraic Sum of A and B:")
    print(algebraic_sum)

    algebraic_product = setA.algebraic_product(setB)
    print("\nAlgebraic Product of A and B:")
    print(algebraic_product)

    cartesian_product = setA.cartesian_product(setB)
    print("\nCartesian Product of A and B:")
    for pair, value in cartesian_product.items():
        print(f"{pair}: {value}")
if __name__ == "__main__":
    main()
```

Q.7: Write a program to implement lambda cut

➔

```
class FuzzySet:
    def __init__(self, elements):
        self.elements = elements  # elements is a dictionary with element:
membership value

    def lambda_cut(self, lambda_value):
        cut_set = {key: value for key, value in self.elements.items() if value
>= lambda_value}
        return cut_set

    def __str__(self):
        return str(self.elements)

def input_fuzzy_set(prompt):
    elements = input(prompt).split()
    fuzzy_set = {}
    for element in elements:
        key, value = element.split(':')
```

```
        fuzzy_set[key] = float(value)
    return FuzzySet(fuzzy_set)


def main():
    print("Enter elements of the Fuzzy Set (format: 
element:membership_value, space-separated):")
    fuzzy_set = input_fuzzy_set("> ")

    lambda_value = float(input("Enter the lambda value (threshold): "))

    lambda_cut_set = fuzzy_set.lambda_cut(lambda_value)
    print(f"\nLambda-cut at {lambda_value}:")
    print(lambda_cut_set)


if __name__ == "__main__":
    main()
```

Q.8:Implement Multilayer perceptron algorithm in Python.

➔

```
import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt


# Load the MNIST dataset

(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()


# Preprocess the data

train_images = train_images.reshape((60000, 28 * 28)).astype('float32')
/ 255

test_images = test_images.reshape((10000, 28 * 28)).astype('float32') /
255


# Build the MLP model

model = models.Sequential()

model.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
```

```python
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam',
          loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

# Train the model
history = model.fit(train_images, train_labels, epochs=10,
batch_size=128, validation_split=0.2)

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy: {test_acc}')

# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
```

```
plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.show()


# Plot training & validation loss values

plt.plot(history.history['loss'])

plt.plot(history.history['val_loss'])

plt.title('Model loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.show()
```

Q.9:.Write python program to create target string, starting from random string using Genetic Algorithm

➔

```
import random


# Define the target string and the allowed characters

target = "Hello, World!"

allowed_chars =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx
yz ,!"


# Define the population size, mutation rate, and number of
generations

population_size = 100

mutation_rate = 0.01

num_generations = 1000


def generate_random_string(length):
```

```
    return ''.join(random.choice(allowed_chars) for _ in
range(length))


def calculate_fitness(individual):
    return sum(1 for a, b in zip(individual, target) if a == b)


def mutate(individual):
    individual = list(individual)
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = random.choice(allowed_chars)
    return ''.join(individual)


def crossover(parent1, parent2):
    crossover_point = random.randint(0, len(parent1))
    child = parent1[:crossover_point] +
parent2[crossover_point:]
    return child
```

```
def main():
    # Initialize population
    population = [generate_random_string(len(target)) for _ in
range(population_size)]

    for generation in range(num_generations):
        # Calculate fitness for each individual
        population = sorted(population, key=calculate_fitness,
reverse=True)

        # If the best individual matches the target, stop the
algorithm
        if calculate_fitness(population[0]) == len(target):
            print(f"Target string evolved in generation {generation}:
{population[0]}")
            break

        # Select the best individuals to form the next generation
        next_generation = population[:population_size // 2]
```

```
        # Create the next generation through crossover and mutation
        for i in range(len(next_generation)):
            parent1 = random.choice(next_generation)
            parent2 = random.choice(next_generation)
            child = mutate(crossover(parent1, parent2))
            next_generation.append(child)

        population = next_generation

        if generation % 100 == 0:
            print(f"Generation {generation}: {population[0]}")

    if calculate_fitness(population[0]) != len(target):
        print(f"Target string not evolved in {num_generations} generations. Best string: {population[0]}")

if __name__ == "__main__":
    main()
```

Q.10: Implement deep learning using Python.

➔

```
import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import cifar10

import matplotlib.pyplot as plt


# Load and preprocess the CIFAR-10 dataset

(train_images, train_labels), (test_images, test_labels) =
cifar10.load_data()

train_images, test_images = train_images / 255.0, test_images
/ 255.0


# Build the CNN model

model = models.Sequential([

   layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32,
32, 3)),

   layers.MaxPooling2D((2, 2)),

   layers.Conv2D(64, (3, 3), activation='relu'),
```

```
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy: {test_acc}')
```

```
# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

Q.11: .Build simple Neural network in Python from scratch.

➔

```
import numpy as np


# Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(x):
    return x * (1 - x)


# Initialize the neural network
def initialize_network(input_size, hidden_size, output_size):
    network = {
        'input_to_hidden_weights': np.random.randn(input_size,
hidden_size),
        'hidden_to_output_weights': np.random.randn(hidden_size,
output_size),
        'hidden_bias': np.random.randn(hidden_size),
        'output_bias': np.random.randn(output_size)
```

```
    }
    return network

# Forward propagation
def forward_propagation(network, inputs):
    hidden_layer_input = np.dot(inputs,
network['input_to_hidden_weights']) + network['hidden_bias']
    hidden_layer_output = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_output,
network['hidden_to_output_weights']) + network['output_bias']
    output_layer_output = sigmoid(output_layer_input)

    return hidden_layer_output, output_layer_output

# Backpropagation
def backpropagation(network, inputs, hidden_layer_output,
output_layer_output, expected_output):
    output_error = expected_output - output_layer_output
    output_delta = output_error *
sigmoid_derivative(output_layer_output)
```

```
    hidden_error =
output_delta.dot(network['hidden_to_output_weights'].T)
    hidden_delta = hidden_error *
sigmoid_derivative(hidden_layer_output)


    network['input_to_hidden_weights'] += inputs.T.dot(hidden_delta)
    network['hidden_to_output_weights'] +=
hidden_layer_output.T.dot(output_delta)
    network['hidden_bias'] += np.sum(hidden_delta, axis=0)
    network['output_bias'] += np.sum(output_delta, axis=0)


# Training the neural network
def train_network(network, inputs, expected_output, epochs,
learning_rate):
    for epoch in range(epochs):
        hidden_layer_output, output_layer_output =
forward_propagation(network, inputs)
        backpropagation(network, inputs, hidden_layer_output,
output_layer_output, expected_output)


        # Optionally, print the loss every 1000 epochs
```

```
        if epoch % 1000 == 0:
            loss = np.mean((expected_output - output_layer_output) ** 2)
            print(f'Epoch {epoch}, Loss: {loss}')


# Input function to take user data
def input_data():
    num_samples = int(input("Enter the number of samples: "))
    inputs = []
    expected_output = []


    for i in range(num_samples):
        sample = list(map(float, input(f"Enter inputs for sample {i+1}
(space-separated): ").split()))
        output = list(map(float, input(f"Enter expected output for sample
{i+1} (space-separated): ").split()))
        inputs.append(sample)
        expected_output.append(output)


    return np.array(inputs), np.array(expected_output)
```

```
# Set parameters
input_size = 2
hidden_size = 2
output_size = 1
epochs = 10000
learning_rate = 0.1

# Input data
inputs, expected_output = input_data()

# Initialize and train the network
network = initialize_network(input_size, hidden_size, output_size)
train_network(network, inputs, expected_output, epochs,
learning_rate)

# Test the neural network
hidden_layer_output, output_layer_output =
forward_propagation(network, inputs)
print("\nPredicted Output:")
print(output_layer_output)
```

Q.12: Write python program to Implement travelling sales man problem using genetic algorithm

➔

```
import random
import numpy as np

# Define the distance matrix
def create_distance_matrix(num_cities):
    matrix = np.random.rand(num_cities, num_cities) * 100
    matrix = (matrix + matrix.T) / 2  # Make the matrix symmetric
    np.fill_diagonal(matrix, 0)  # Distance from a city to itself is 0
    return matrix

# Create an initial population
def create_initial_population(pop_size, num_cities):
    population = []
    for _ in range(pop_size):
        tour = list(np.random.permutation(num_cities))
        population.append(tour)
    return population
```

```
# Calculate the total distance of a tour

def calculate_fitness(tour, distance_matrix):

    distance = 0

    for i in range(len(tour)):

        distance += distance_matrix[tour[i - 1], tour[i]]

    return distance


# Selection (tournament selection)

def selection(population, distance_matrix):

    selected = []

    for _ in range(len(population) // 2):

        tournament = random.sample(population, k=4)

        tournament = sorted(tournament, key=lambda x:
calculate_fitness(x, distance_matrix))

        selected.extend(tournament[:2])

    return selected


# Crossover (ordered crossover)

def crossover(parent1, parent2):
```

```
    size = len(parent1)
    start, end = sorted(random.sample(range(size), 2))
    child = [None] * size
    child[start:end] = parent1[start:end]
    for city in parent2:
        if city not in child:
            for i in range(size):
                if child[i] is None:
                    child[i] = city
                    break
    return child

# Mutation (swap mutation)
def mutate(tour, mutation_rate):
    for i in range(len(tour)):
        if random.random() < mutation_rate:
            j = random.randint(0, len(tour) - 1)
            tour[i], tour[j] = tour[j], tour[i]

# Genetic Algorithm
```

```
def genetic_algorithm(num_cities, pop_size, generations,
mutation_rate):
    distance_matrix = create_distance_matrix(num_cities)
    population = create_initial_population(pop_size, num_cities)

    for generation in range(generations):
        population = sorted(population, key=lambda x: calculate_fitness(x,
distance_matrix))
        next_generation = selection(population, distance_matrix)

        while len(next_generation) < pop_size:
            parent1, parent2 = random.sample(next_generation, 2)
            child = crossover(parent1, parent2)
            mutate(child, mutation_rate)
            next_generation.append(child)

        population = next_generation

        if generation % 100 == 0:
            print(f"Generation {generation}, Best fitness:
{calculate_fitness(population[0], distance_matrix)}")
```

```
    best_tour = min(population, key=lambda x: calculate_fitness(x,
distance_matrix))

    print(f"Best tour: {best_tour}")

    print(f"Best fitness: {calculate_fitness(best_tour, distance_matrix)}")


# User input for parameters

num_cities = int(input("Enter the number of cities: "))

pop_size = int(input("Enter the population size: "))

generations = int(input("Enter the number of generations: "))

mutation_rate = float(input("Enter the mutation rate (e.g., 0.01 for 1%):
"))


# Run the genetic algorithm

genetic_algorithm(num_cities, pop_size, generations, mutation_rate)
```