INTRODUCTION TO ARTIFICIAL INTELLIGENCE LAB 10
MOMINA EMAN 221106
BSIT-VII-B

DEMO TASK:

```
# Python3 program to create target string starting from
# random string using Genetic Algorithm

import random

# Number of individuals in each generation
POPULATION_SIZE = 200

# Valid genes
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ 1234567890,
.-;:_!"#%&/()=?@${[]}'''

# Target string to be generated
TARGET = "Qasim"


class Individual(object):
    # Class representing individual in population

    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    @classmethod
    def mutated_genes(cls):
        # Create random genes for mutation
        global GENES
        gene = random.choice(GENES)
        return gene

    @classmethod
    def create_gnome(cls):
        # Create chromosome or string of genes
        global TARGET
        gnome_len = len(TARGET)
        return [cls.mutated_genes() for _ in range(gnome_len)]

    def mate(self, par2):
        # Perform mating and produce new offspring

        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):
            prob = random.random()

            if prob < 0.45:
                child_chromosome.append(gp1)
            elif prob < 0.90:
```

```
                child_chromosome.append(gp2)
            else:
                child_chromosome.append(self.mutated_genes())

        return Individual(child_chromosome)

    def cal_fitness(self):
        # Calculate fitness score = number of different characters
        global TARGET
        fitness = 0
        for gs, gt in zip(self.chromosome, TARGET):
            if gs != gt:
                fitness += 1
        return fitness


def main():

    global POPULATION_SIZE

    generation = 1
    found = False

    population = []

    # Create initial population
    for _ in range(POPULATION_SIZE):
        gnome = Individual.create_gnome()
        population.append(Individual(gnome))

    while not found:

        # Sort population by fitness
        population = sorted(population, key=lambda x: x.fitness)

        # If target found
        if population[0].fitness == 0:
            found = True
            break

        new_generation = []

        # Elitism : top 10% carried to next generation
        s = int((10 * POPULATION_SIZE) / 100)
        new_generation.extend(population[:s])

        # Remaining 90% : Crossover & Mutation
        s2 = int((90 * POPULATION_SIZE) / 100)
        for _ in range(s2):
            parent1 = random.choice(population[:50])
            parent2 = random.choice(population[:50])
            child = parent1.mate(parent2)
            new_generation.append(child)
```

```python
        population = new_generation

        print("Generation: {} \tString: {}\tFitness: {}".format(
            generation,
            "".join(population[0].chromosome),
            population[0].fitness))

        generation += 1

    print("\nTarget Reached!")
    print("Generation: {} \tString: {}\tFitness: {}".format(
        generation,
        "".join(population[0].chromosome),
        population[0].fitness))


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

OUTPUT:

```
C:\Users\221106\PycharmProjects\PythonProject\.venv\Scripts\python.exe "C:\Users\221106\PycharmProjects\Py
Generation: 1   String: %1#bmB,E7h[ Fitness: 10
Generation: 2   String: VVo&=b &mG8 Fitness: 9
Generation: 3   String: Mo/sZa}mA7R Fitness: 8
Generation: 4   String: fM[i7a}Sma- Fitness: 7
Generation: 5   String: Mom"Za$EmL- Fitness: 6
Generation: 6   String: MMminaqSma- Fitness: 4
Generation: 7   String: MMminaqSma- Fitness: 4
Generation: 8   String: MomiNa Emao Fitness: 2
Generation: 9   String: Momina Ema[ Fitness: 1
Generation: 10  String: Momina Ema[ Fitness: 1

Target Reached!
Generation: 11  String: Momina Eman Fitness: 0

Process finished with exit code 0
```

LAB TASK:

```python
import random
import math

# Number of routes in each generation
POPULATION_SIZE = 100

# Warehouse + 10 locations (You can rename them if needed)
locations = {
    "W": (0, 0),
    "A": (2, 3),
    "B": (5, 4),
    "C": (1, 8),
    "D": (3, 7),
    "E": (8, 9),
    "F": (6, 1),
```

```python
    "G": (4, 6),
    "H": (9, 3),
    "I": (7, 7),
    "J": (2, 2)
}

location_ids = list(locations.keys())
location_ids.remove("W") # Warehouse fixed start & end


def distance(p1, p2):
    """ Euclidean distance """
    return math.dist(locations[p1], locations[p2])


def create_route():
    """ Chromosome representation: route starting/ending at Warehouse
    """ route = location_ids[:]
    random.shuffle(route)
    return ["W"] + route + ["W"]


def route_distance(route):
    """ Total distance of route """
    dist = 0
    for i in range(len(route) - 1):
        dist += distance(route[i], route[i + 1])
    return dist


def fitness(route):
    """ Higher fitness for shorter routes """
    return 1 / route_distance(route)
def selection(population):
    """ Roulette Wheel Selection """
    total_fitness = sum(fitness(r) for r in population)
    pick = random.uniform(0, total_fitness)
    current = 0
    for route in population:
        current += fitness(route)
        if current > pick:
            return route


def crossover(parent1, parent2):
    """ Order Crossover (OX) """
    start, end = sorted(random.sample(range(1, len(parent1)-1), 2))

    child = [None] * len(parent1)
    child[0] = "W"
    child[-1] = "W"

    child[start:end] = parent1[start:end]
```

```python
    fill = [p for p in parent2 if p not in child]
    j = 1
    for x in fill:
        while child[j] is not None:
            j += 1
        child[j] = x
    return child


def mutate(route):
    """ Swap Mutation """
    i, j = random.sample(range(1, len(route)-1), 2)
    route[i], route[j] = route[j], route[i]
    return route


def genetic_algorithm():
    population = [create_route() for _ in range(POPULATION_SIZE)]

    best_route = min(population, key=route_distance)
    best_distance = route_distance(best_route)

    generations_no_improve = 0

    for generation in range(100):
        new_population = []

        for _ in range(POPULATION_SIZE):
            parent1 = selection(population)
            parent2 = selection(population)
            child = crossover(parent1, parent2)

            if random.random() < 0.1: # mutation rate 10%
                child = mutate(child)

            new_population.append(child)

        population = new_population

        current_best = min(population, key=route_distance)
        current_distance = route_distance(current_best)

        print(f"Generation {generation+1} → Distance: {current_distance:.2f} 
Route: {current_best}")

        if current_distance < best_distance:
            best_distance = current_distance
            best_route = current_best
            generations_no_improve = 0
        else:
            generations_no_improve += 1

        if generations_no_improve >= 20:
```

```python
                break

    print("\n Best Route Found:")
    print(f"Route: {best_route}")
    print(f"Total Distance: {best_distance:.2f}")


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

OUTPUT: