**Q1 == For Target String**

**Code:**

import random

# Number of individuals in each generation

POPULATION\_SIZE = 100

# Valid genes

GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP

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

# Target string to be generated (can change to experiment)

TARGET = "Hello, Genetic Algorithm"

class Individual:

'''

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):

# Random probability

prob = random.random()

# If prob is less than 0.45, insert gene from parent 1

if prob < 0.45:

child\_chromosome.append(gp1)

# If prob is between 0.45 and 0.90, insert gene from parent 2

elif prob < 0.90:

child\_chromosome.append(gp2)

# Otherwise insert random gene (mutate) for maintaining diversity

else:

child\_chromosome.append(self.mutated\_genes())

# Create new Individual (offspring) using generated chromosome for offspring

return Individual(child\_chromosome)

def cal\_fitness(self):

'''

Calculate fitness score, it is the number of characters in string

which differ from target string.

'''

global TARGET

fitness = 0

for gs, gt in zip(self.chromosome, TARGET):

if gs != gt:

fitness += 1

return fitness

# Driver code for Target String

def main\_target\_string():

global POPULATION\_SIZE

# Current generation

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 the population in increasing order of fitness score

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

# If the individual with the lowest fitness score (0) is found, we reached the target string

if population[0].fitness == 0:

found = True

break

# Otherwise, generate new offspring for the next generation

new\_generation = []

# Perform Elitism: 10% of fittest population goes to the next generation

s = int((10 \* POPULATION\_SIZE) / 100)

new\_generation.extend(population[:s])

# From 90% of fittest population, individuals will mate to produce offspring

s = int((90 \* POPULATION\_SIZE) / 100)

for \_ in range(s):

parent1 = random.choice(population[:50])

parent2 = random.choice(population[:50])

child = parent1.mate(parent2)

new\_generation.append(child)

population = new\_generation

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

generation += 1

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

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

main\_target\_string()

**Q2 == For N Queen Problem**

**Code:**

import random

# Define the N-Queens problem

N = 8 # Size of the board and the number of queens

# Class to represent an individual solution for the N-Queens problem

class Individual:

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

self.chromosome = chromosome

self.fitness = self.cal\_fitness()

@classmethod

def create\_gnome(cls):

"""

Create a random solution (chromosome) where each queen is placed in a random row of each column

"""

return random.sample(range(N), N)

def cal\_fitness(self):

"""

Calculate fitness by counting the number of conflicts between queens.

The fewer conflicts, the better the fitness.

"""

conflicts = 0

for i in range(N):

for j in range(i + 1, N):

if self.chromosome[i] == self.chromosome[j] or \

abs(self.chromosome[i] - self.chromosome[j]) == abs(i - j):

conflicts += 1

return conflicts

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(random.choice(range(N)))

return Individual(child\_chromosome)

# Driver code for N-Queens

def main\_n\_queens():

global N

POPULATION\_SIZE = 100

generations = 0

found = False

population = []

# Create initial population

for \_ in range(POPULATION\_SIZE):

chromosome = Individual.create\_gnome()

population.append(Individual(chromosome))

while not found:

# Sort population based on fitness

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

if population[0].fitness == 0:

found = True

break

new\_generation = []

s = int((10 \* POPULATION\_SIZE) / 100)

new\_generation.extend(population[:s]) # Elitism: 10% fittest go to next generation

s = int((90 \* POPULATION\_SIZE) / 100)

for \_ in range(s):

parent1 = random.choice(population[:50])

parent2 = random.choice(population[:50])

child = parent1.mate(parent2)

new\_generation.append(child)

population = new\_generation

generations += 1

print(f"Generation: {generations}\tFitness: {population[0].fitness}")

print(f"Solution found in Generation: {generations}\n{population[0].chromosome}")

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

main\_n\_queens()