Assignment 2(SCOA)

Code--

```
import random

POPULATION_SIZE = 100

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

TARGET = "Hello World"

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

      @classmethod
      def mutated_genes(self):
            global GENES
            gene = random.choice(GENES)
            return gene

      @classmethod
      def create_gnome(self):
            global TARGET
            gnome_len = len(TARGET)
            return [self.mutated_genes() for _ in range(gnome_len)]

      def mate(self, par2):
            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):
            global TARGET
            fitness = 0
            for gs, gt in zip(self.chromosome, TARGET):
                  if gs != gt:
                        fitness+= 1
            return fitness

generation = 1
found = False
population = []

for _ in range(POPULATION_SIZE):
                  gnome = Individual.create_gnome()
```

```
                population.append(Individual(gnome))

while not found:
      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])

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

      generation += 1


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

Output—

Generation: 1 String: }Yj2O#7Orwq Fitness: 10

Generation: 2 String: zr,l a 3?zd Fitness: 9

Generation: 3 String: zr,l a 3?zd Fitness: 9

Generation: 4 String: uf5l?5Wg(Xd Fitness: 8

Generation: 5 String: Zrll[R7G:ld Fitness: 7

Generation: 6 String: ,fHlM 9orFd Fitness: 6

Generation: 7 String: ,fHlM 9orFd Fitness: 6

Generation: 8 String: @sllM uorwd Fitness: 5

Generation: 9 String: Esll WorFd Fitness: 4

Generation: 10 String: Esll WorFd Fitness: 4

Generation: 11 String: Esll WorFd Fitness: 4

Generation: 12 String: u!ll World Fitness: 3

Generation: 13 String: u!ll World Fitness: 3

Generation: 14 String: Hsllo Woryd Fitness: 2

Generation: 15 String: Hsllo Woryd Fitness: 2

Generation: 16 String: Hsllo Woryd Fitness: 2

Generation: 17 String: H@llo World Fitness: 1

Generation: 18 String: H@llo World Fitness: 1

Generation: 19 String: H@llo World Fitness: 1

Generation: 20 String: H@llo World Fitness: 1

Generation: 21 String: H@llo World Fitness: 1

Generation: 22 String: H@llo World Fitness: 1

Generation: 23 String: H@llo World Fitness: 1

Generation: 24 String: H@llo World Fitness: 1

Generation: 25 String: H@llo World Fitness: 1

Generation: 26 String: H@llo World Fitness: 1

Generation: 27 String: H@llo World Fitness: 1

Generation: 28 String: H@llo World Fitness: 1

Generation: 29 String: H@llo World Fitness: 1

Generation: 30 String: H@llo World Fitness: 1

Generation: 31 String: H@llo World Fitness: 1

Generation: 32 String: H@llo World Fitness: 1

Generation: 33 String: H@llo World Fitness: 1

Generation: 34 String: H@llo World Fitness: 1

Generation: 35 String: H@llo World Fitness: 1

Generation: 36 String: H@llo World Fitness: 1

Generation: 37 String: H@llo World Fitness: 1

Generation: 38 String: H@llo World Fitness: 1

Generation: 39 String: H@llo World Fitness: 1

Generation: 40 String: H@llo World Fitness: 1

Generation: 41 String: H@llo World Fitness: 1

Generation: 42 String: H@llo World Fitness: 1

Generation: 43 String: H@llo World Fitness: 1

Generation: 44 String: H@llo World Fitness: 1

Generation: 45 String: H@llo World Fitness: 1

Generation: 46 String: H@llo World Fitness: 1

Generation: 47 String: H@llo World Fitness: 1

Generation: 48 String: H@llo World Fitness: 1

Generation: 49 String: H@llo World Fitness: 1

Generation: 50 String: H@llo World Fitness: 1

Generation: 51 String: H@llo World Fitness: 1

Generation: 52 String: H@llo World Fitness: 1

Generation: 53 String: H@llo World Fitness: 1

Generation: 54 String: H@llo World Fitness: 1

Generation: 55 String: H@llo World Fitness: 1

Generation: 56 String: H@llo World Fitness: 1

Generation: 57 String: H@llo World Fitness: 1

Generation: 58 String: H@llo World Fitness: 1

Generation: 59 String: H@llo World Fitness: 1

Generation: 60 String: H@llo World Fitness: 1

Generation: 61 String: H@llo World Fitness: 1

Generation: 62 String: H@llo World Fitness: 1

Generation: 63 String: H@llo World Fitness: 1

Generation: 64 String: H@llo World Fitness: 1

Generation: 65 String: H@llo World Fitness: 1

Generation: 66 String: H@llo World Fitness: 1

Generation: 67 String: H@llo World Fitness: 1

Generation: 68 String: H@llo World Fitness: 1

Generation: 69 String: H@llo World Fitness: 1

Generation: 70 String: Hello World Fitness: 0

Name: Nachiket Erlekar
Roll: 41434 Batch: Q4

Assignment No: 02

Aim: Implement genetic algorithm for benchmark function (eg. Square, Rosenbrock function etc) Initialize the population from the Standard Normal Distribution. Evaluate the fitness of all its individuals. Then you will do multiple generation of a genetic algorithm. A generation consists of applying selection, crossover, mutation, and replacement.

Use:

- Tournament selection without replacement with tournament sizes
- One point crossover with probability Pc
- bit-flip mutation with probability Pm
- use full replacement strategy

1. To learn Genetic Algorithm.
2. To learn about optimization algorithm.

Software Requirements:

Ubuntu 18.04

Hardware Requirements:

Pentium IV system with latest configuration

Theory:

Genetic Algorithms are a class of stochastic, population based optimization algorithms inspired by

the biological evolution process using the concepts of “Natural Selection” and “Genetic Inheritance” (Darwin 1859) and originally developed by Holland.

GAs are now used in engineering and business optimization applications, where the search space is large and/or too complex (non-smooth) for analytic treatment. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found.

This notion can be applied for a search problem. We consider a set of solutions for a problem and select the set of best ones out of them.

Five phases are considered in a genetic algorithm.

1. Initial population

2. Fitness function
3. Selection
4. Crossover
5. Mutation

Initial Population

The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve. An individual is characterized by a set of parameters (variables) known as **Genes**. Genes are joined into a string to form a **Chromosome**(solution). In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.

Fitness Function

The **fitness function** determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a **fitness score** to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

Selection

The idea of **selection** phase is to select the fittest individuals and let them pass their genes to the next generation.

Two pairs of individuals (**parents**) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.

Crossover

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a **crossover point** is chosen at random from within the genes.

Off spring are created by exchanging the genes of parents among themselves until the crossover point is reached.

Mutation

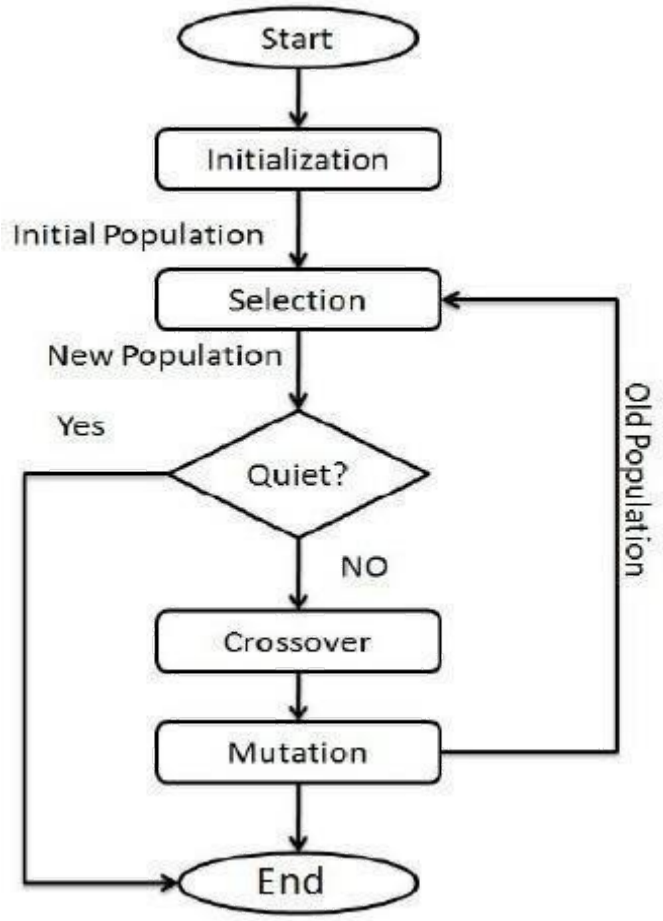
In certain new offspring formed, some of their genes can be subjected to a**mutation**

with a low random probability. This implies that some of the bits in the bit string can beflipped.

Mutation occurs to maintain diversity within the population and prevent prematureconvergence.

Termination

The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.



Conclusion:

Thus we learnt implementation of genetic algorithm for benchmark function.