# Python3 program to create target string, starting from

# random string using Genetic Algorithm

import random

# Number of individuals in each generation

POPULATION\_SIZE = 100

# Valid genes

GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP

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

# Target string to be generated

TARGET = "I love GeeksforGeeks"

class Individual(object):

'''

Class representing individual in population

'''

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

self.chromosome = chromosome

self.fitness = self.cal\_fitness()

@classmethod

def mutated\_genes(self):

'''

create random genes for mutation

'''

global GENES

gene = random.choice(GENES)

return gene

@classmethod

def create\_gnome(self):

'''

create chromosome or string of genes

'''

global TARGET

gnome\_len = len(TARGET)

return [self.mutated\_genes() for \_ in range(gnome\_len)]

def mate(self, par2):

'''

Perform mating and produce new offspring

'''

# chromosome for 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

def main():

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 having lowest fitness score ie.

# 0 then we know that we have reached to the target

# and break the loop

if population[0].fitness <= 0:

found = True

break

# Otherwise generate new offsprings for new generation

new\_generation = []

# Perform Elitism, that mean 10% of fittest population

# goes to the next generation

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

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

# From 50% 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("Generation: {}\tString: {}\tFitness: {}",

format(generation,

"".join(population[0].chromosome),

population[0].fitness))

generation += 1

print("Generation: {}\tString: {}\tFitness: {}",

format(generation,

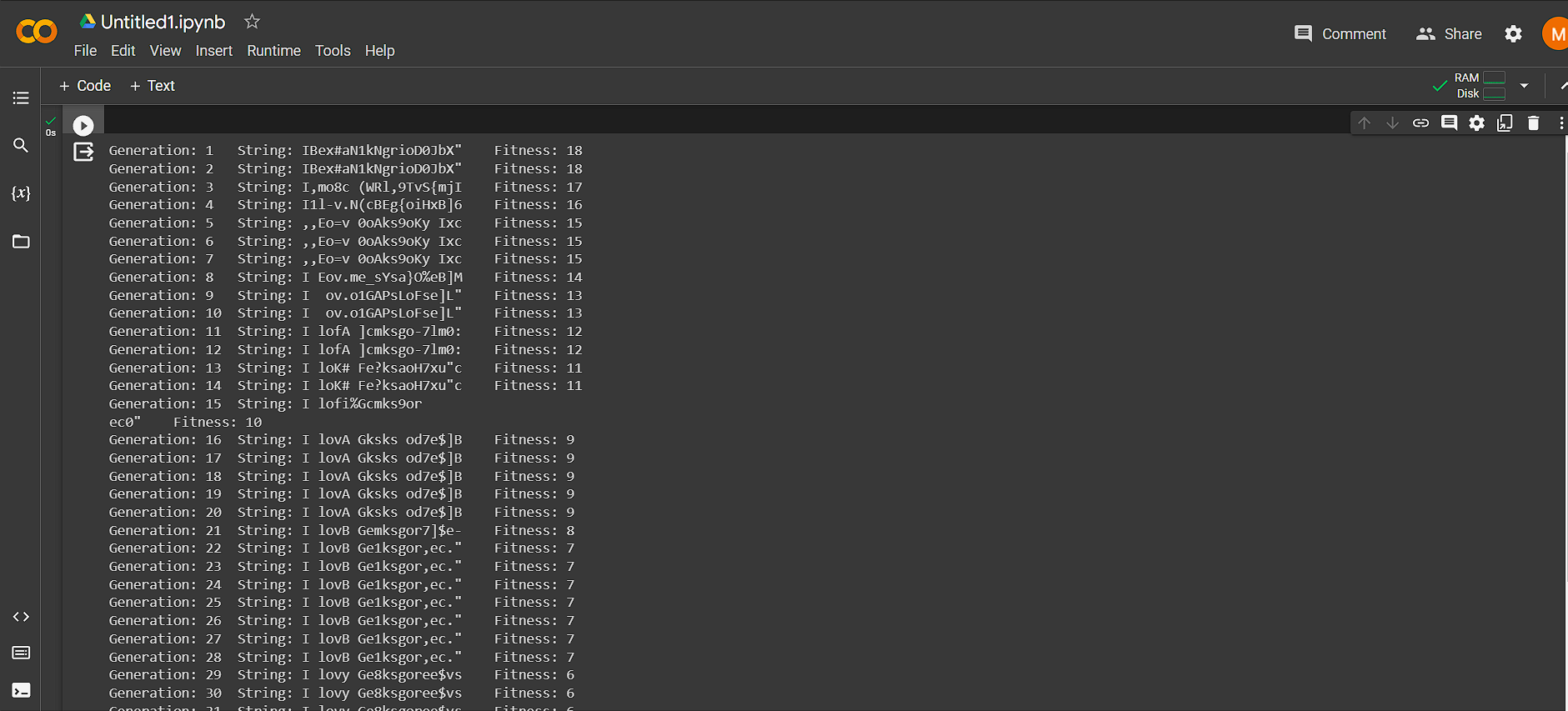
"".join(population[0].chromosome),

population[0].fitness))

if \_\_name\_\_ == '\_\_main\_\_':

main()

OUTPUT



Code 2

import numpy as np

# Define the fitness function to be optimized

def fitness\_function(x):

return -x\*\*2 # We want to find the maximum of this function

# Genetic algorithm parameters

population\_size = 20

chromosome\_length = 5

mutation\_rate = 0.1

crossover\_rate = 0.8

num\_generations = 100

# Generate an initial population with random chromosomes

def initialize\_population(population\_size, chromosome\_length):

return np.random.randint(2, size=(population\_size, chromosome\_length))

# Decode the binary chromosome to a decimal value

def decode\_chromosome(chromosome):

return int(''.join([str(bit) for bit in chromosome]), 2)

# Calculate fitness for each individual in the population

def calculate\_fitness(population):

return np.array([fitness\_function(decode\_chromosome(chromosome)) for chromosome in population])

# Perform single-point crossover between two parent chromosomes

def crossover(parent1, parent2):

crossover\_point = np.random.randint(1, len(parent1) - 1)

child1 = np.concatenate((parent1[:crossover\_point], parent2[crossover\_point:]))

child2 = np.concatenate((parent2[:crossover\_point], parent1[crossover\_point:]))

return child1, child2

# Perform mutation on a chromosome

def mutate(chromosome):

for i in range(len(chromosome)):

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

chromosome[i] = 1 - chromosome[i]

return chromosome

# Genetic algorithm

def genetic\_algorithm(population\_size, chromosome\_length, mutation\_rate, crossover\_rate, num\_generations):

population = initialize\_population(population\_size, chromosome\_length)

for generation in range(num\_generations):

fitness\_values = calculate\_fitness(population)

# Select parents for crossover

parents = population[np.random.choice(population\_size, size=population\_size, replace=True, p=fitness\_values / np.sum(fitness\_values))]

# Perform crossover and mutation

for i in range(0, population\_size, 2):

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

population[i], population[i + 1] = crossover(parents[i], parents[i + 1])

population[i] = mutate(population[i])

population[i + 1] = mutate(population[i + 1])

best\_individual = population[np.argmax(fitness\_values)]

best\_value = decode\_chromosome(best\_individual)

return best\_value

# Run the genetic algorithm

best\_solution = genetic\_algorithm(population\_size, chromosome\_length, mutation\_rate, crossover\_rate, num\_generations)

print("Best solution found:", best\_solution)

print("Fitness value of best solution:", fitness\_function(best\_solution))

Output

