|  |
| --- |
| **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**  **“JnanaSangama”, Belgaum -590014, Karnataka.**    **LAB RECORD**  **Bio Inspired Systems (23CS5BSBIS)**  ***Submitted by***  **Agneya D A (1BM22CS024)**  ***in partial fulfillment for the award of the degree of***  **BACHELOR OF ENGINEERING**  ***in***  **COMPUTER SCIENCE AND ENGINEERING**    **B.M.S. COLLEGE OF ENGINEERING**  **(Autonomous Institution under VTU)**  **BENGALURU-560019**  **Sep-2024 to Jan-2025** |

**B.M.S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**

****

**CERTIFICATE**

This is to certify that the Lab work entitled “ Bio Inspired Systems (23CS5BSBIS)” carried out by **Agneya D A (1BM22CS024),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

|  |  |
| --- | --- |
| Dr. Shahshikala  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda  Professor & HOD  Department of CSE, BMSCE |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 9/10/24 | Genetic Algorithm | 1 – 4 |
| 2 | 16/10/24 | Particle Swarm Optimization | 5 – 9 |
| 3 | 23/10/24 | Ant Colony Optimization | 10 - 13 |
| 4 | 30/10/24 | Cuckoo Search | 14 – 18 |
| 5 | 13/11/24 | Grey Wolf Optimization | 19 – 22 |
| 6 | 20/11/24 | Parallel Cellular Algorithm | 23 – 25 |
| 7 | 27/11/24 | Gene Expression Optimization | 26 - 28 |

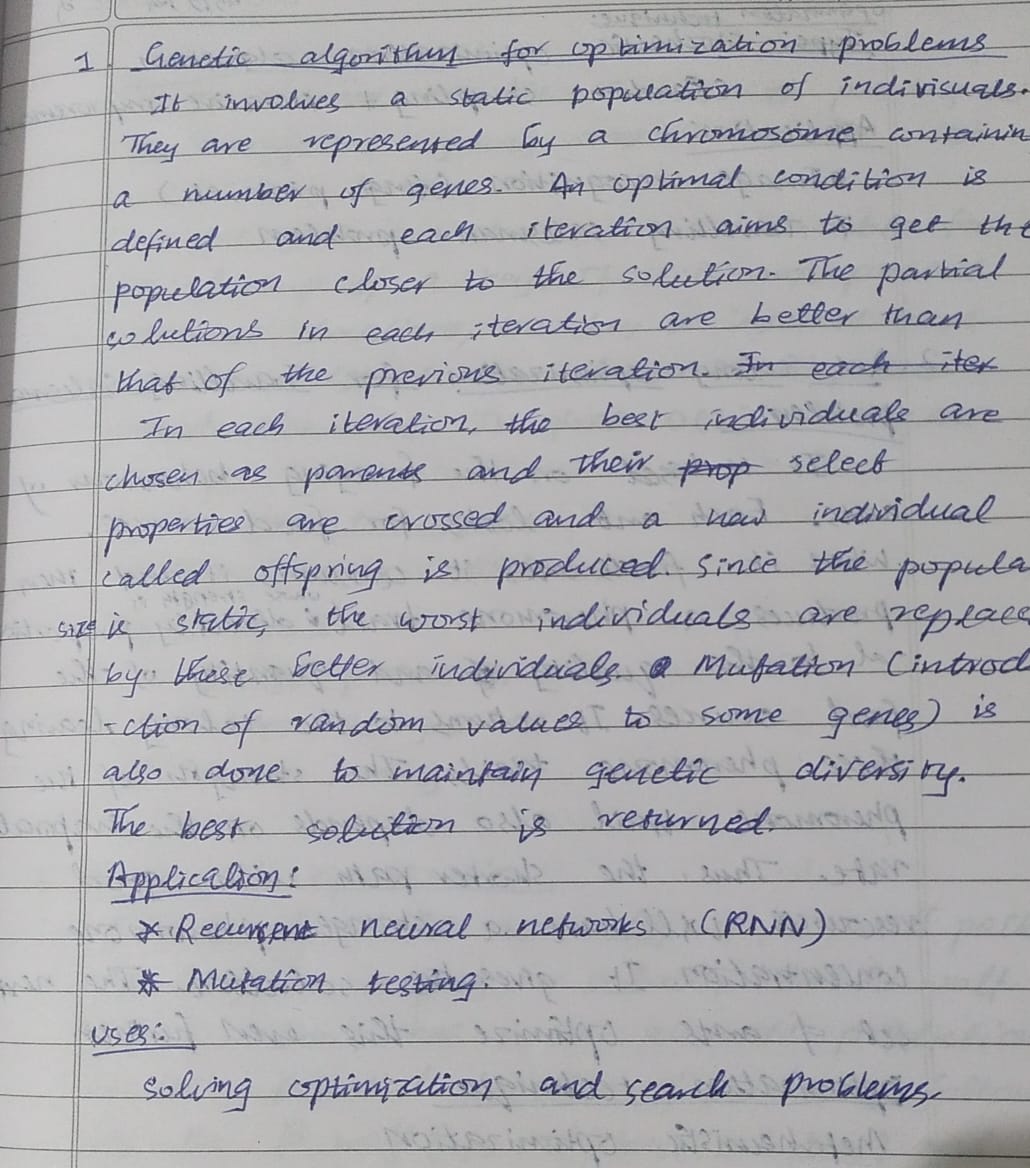
Github Link:

<https://github.com/Agneya-1BM22CS024/BIS_Lab>

**Program 1**

**Genetic Algorithm for Optimization**

**Algorithm:**



**Code:**

import random

import numpy as np

# Fitness function

def fitness(chromosome):

x = int(''.join(map(str, chromosome)), 2)

return x \*\* 2

# Generate a random chromosome

def generate\_chromosome(length):

return [random.randint(0, 1) for \_ in range(length)]

# Generate an initial population

def generate\_population(size, chromosome\_length):

return [generate\_chromosome(chromosome\_length) for \_ in range(size)]

# Select two parents based on their fitness

def select\_pair(population, fitnesses):

total\_fitness = sum(fitnesses)

selection\_probs = [f / total\_fitness for f in fitnesses]

parent1 = population[random.choices(range(len(population)), weights=selection\_probs)[0]]

parent2 = population[random.choices(range(len(population)), weights=selection\_probs)[0]]

return parent1, parent2

# Crossover between two parents

def crossover(parent1, parent2):

point = random.randint(1, len(parent1) - 1)

offspring1 = parent1[:point] + parent2[point:]

offspring2 = parent2[:point] + parent1[point:]

return offspring1, offspring2

# Mutate a chromosome

def mutate(chromosome, mutation\_rate):

return [gene if random.random() > mutation\_rate else 1 - gene for gene in chromosome]

# Parameters

population\_size = 10

chromosome\_length = 5

mutation\_rate = 0.01

generations = 20

# Initialize population

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

for generation in range(generations):

# Calculate fitness for each chromosome

fitnesses = [fitness(chromosome) for chromosome in population]

print(f"Generation {generation + 1} best fitness: {max(fitnesses)}")

# Create a new generation

new\_population = []

for \_ in range(population\_size // 2):

parent1, parent2 = select\_pair(population, fitnesses)

offspring1, offspring2 = crossover(parent1, parent2)

offspring1 = mutate(offspring1, mutation\_rate)

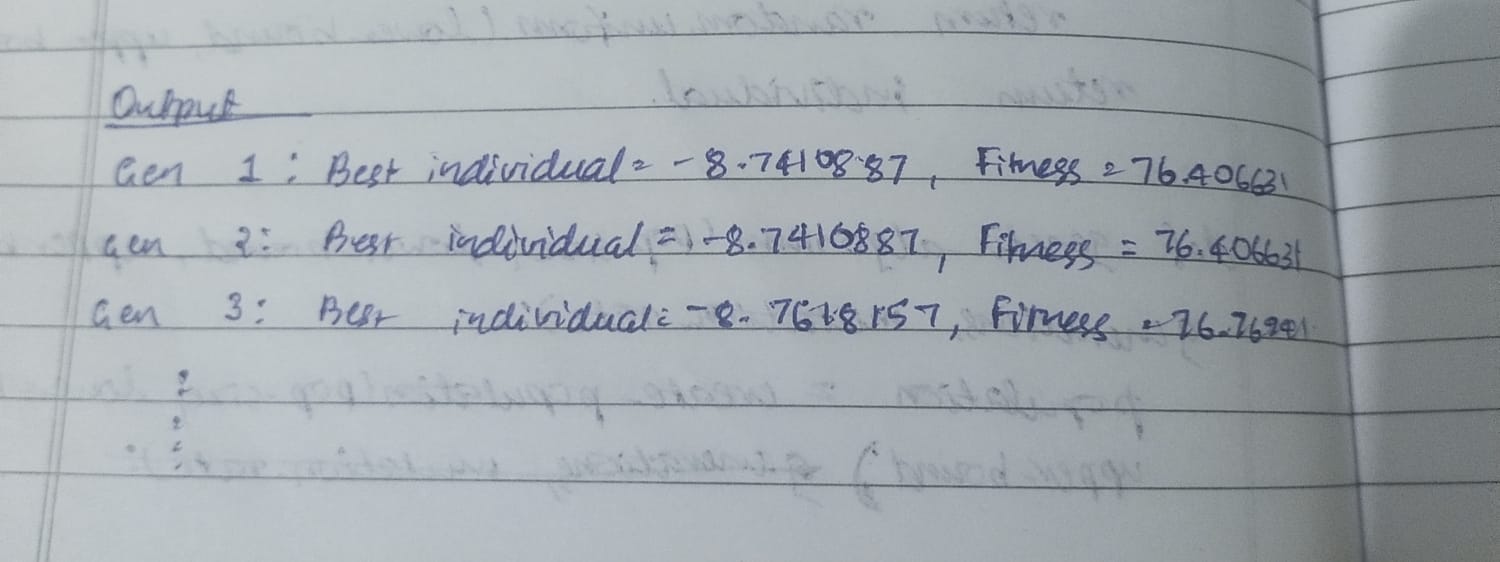
offspring2 = mutate(offspring2, mutation\_rate)

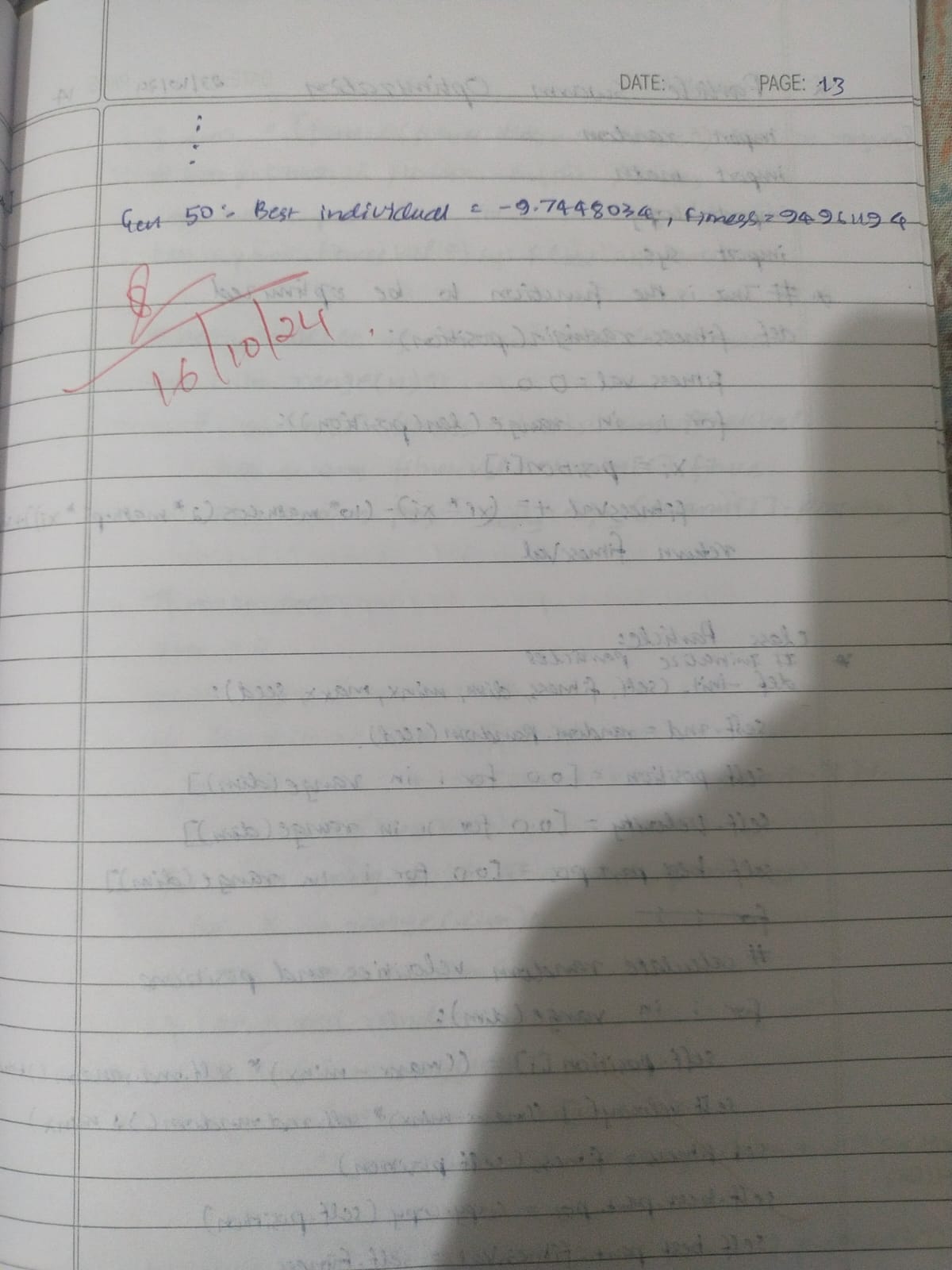
new\_population.extend([offspring1, offspring2])

# Replace the old population

population = new\_population

**Output:**

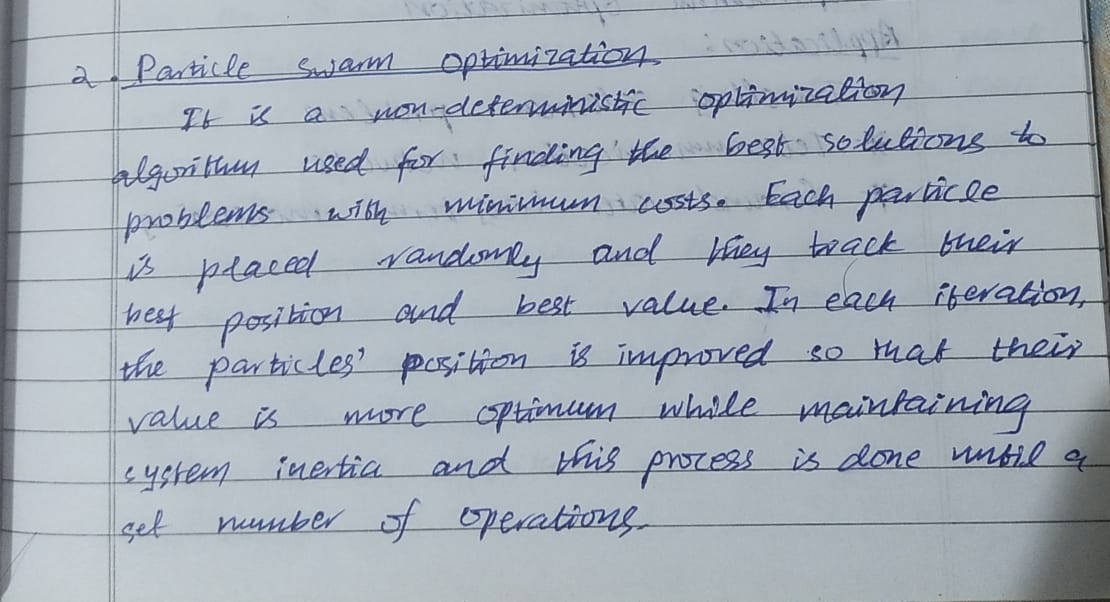


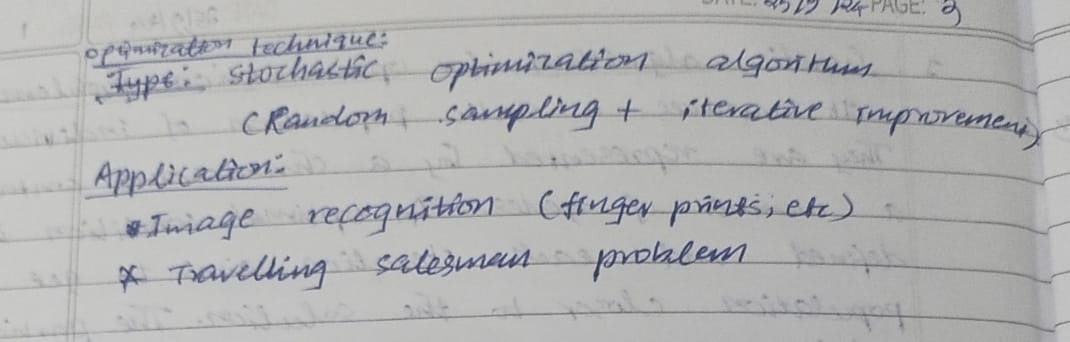


**Program 2**

**Particle Swarm Optimization**

**Algorithm:**

****

****

**Code:**

from \_\_future\_\_ import division

import random

import math

#--- COST FUNCTION ------------------------------------------------------------+

# function we are attempting to optimize (minimize)

def func1(x):

total = 0

for i in range(len(x)):

total += x[i]\*\*2

return total

#--- PARTICLE CLASS -----------------------------------------------------------+

class Particle:

def \_\_init\_\_(self, x0, bounds):

self.position\_i = [] # particle position

self.velocity\_i = [] # particle velocity

self.pos\_best\_i = [] # best position individual

self.err\_best\_i = float('inf') # best error individual

self.err\_i = float('inf') # error individual

for i in range(len(x0)):

self.velocity\_i.append(random.uniform(-1, 1))

self.position\_i.append(random.uniform(bounds[i][0], bounds[i][1]))

# evaluate current fitness

def evaluate(self, costFunc):

self.err\_i = costFunc(self.position\_i)

# check to see if the current position is an individual best

if self.err\_i < self.err\_best\_i:

self.pos\_best\_i = self.position\_i[:]

self.err\_best\_i = self.err\_i

# update new particle velocity

def update\_velocity(self, pos\_best\_g, w=0.5, c1=1.5, c2=2.0):

for i in range(len(self.position\_i)):

r1 = random.random()

r2 = random.random()

vel\_cognitive = c1 \* r1 \* (self.pos\_best\_i[i] - self.position\_i[i])

vel\_social = c2 \* r2 \* (pos\_best\_g[i] - self.position\_i[i])

self.velocity\_i[i] = w \* self.velocity\_i[i] + vel\_cognitive + vel\_social

# update the particle position based off new velocity updates

def update\_position(self, bounds):

for i in range(len(self.position\_i)):

self.position\_i[i] += self.velocity\_i[i]

# adjust maximum position if necessary

if self.position\_i[i] > bounds[i][1]:

self.position\_i[i] = bounds[i][1]

# adjust minimum position if necessary

if self.position\_i[i] < bounds[i][0]:

self.position\_i[i] = bounds[i][0]

#--- PSO CLASS ----------------------------------------------------------------+

class PSO:

def \_\_init\_\_(self, costFunc, x0, bounds, num\_particles, maxiter):

self.num\_dimensions = len(x0)

self.err\_best\_g = float('inf') # best error for group

self.pos\_best\_g = [] # best position for group

self.swarm = [Particle(x0, bounds) for \_ in range(num\_particles)]

self.costFunc = costFunc

self.bounds = bounds

self.num\_particles = num\_particles

self.maxiter = maxiter

def optimize(self):

for i in range(self.maxiter):

for particle in self.swarm:

particle.evaluate(self.costFunc)

# determine if current particle is the best (globally)

if particle.err\_i < self.err\_best\_g:

self.pos\_best\_g = particle.position\_i[:]

self.err\_best\_g = particle.err\_i

for particle in self.swarm:

particle.update\_velocity(self.pos\_best\_g)

particle.update\_position(self.bounds)

# Print iteration details

print(f"Iteration {i+1}/{self.maxiter}, Best Fitness: {self.err\_best\_g}")

# Print final results

print('FINAL RESULTS:')

print(f"Best Position: {self.pos\_best\_g}")

print(f"Best Fitness: {self.err\_best\_g}")

#--- RUN ----------------------------------------------------------------------+

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

initial = [5, 5] # initial starting location [x1, x2, ...]

bounds = [(-10, 10), (-10, 10)] # input bounds [(x1\_min, x1\_max), (x2\_min, x2\_max), ...]

num\_particles = 15

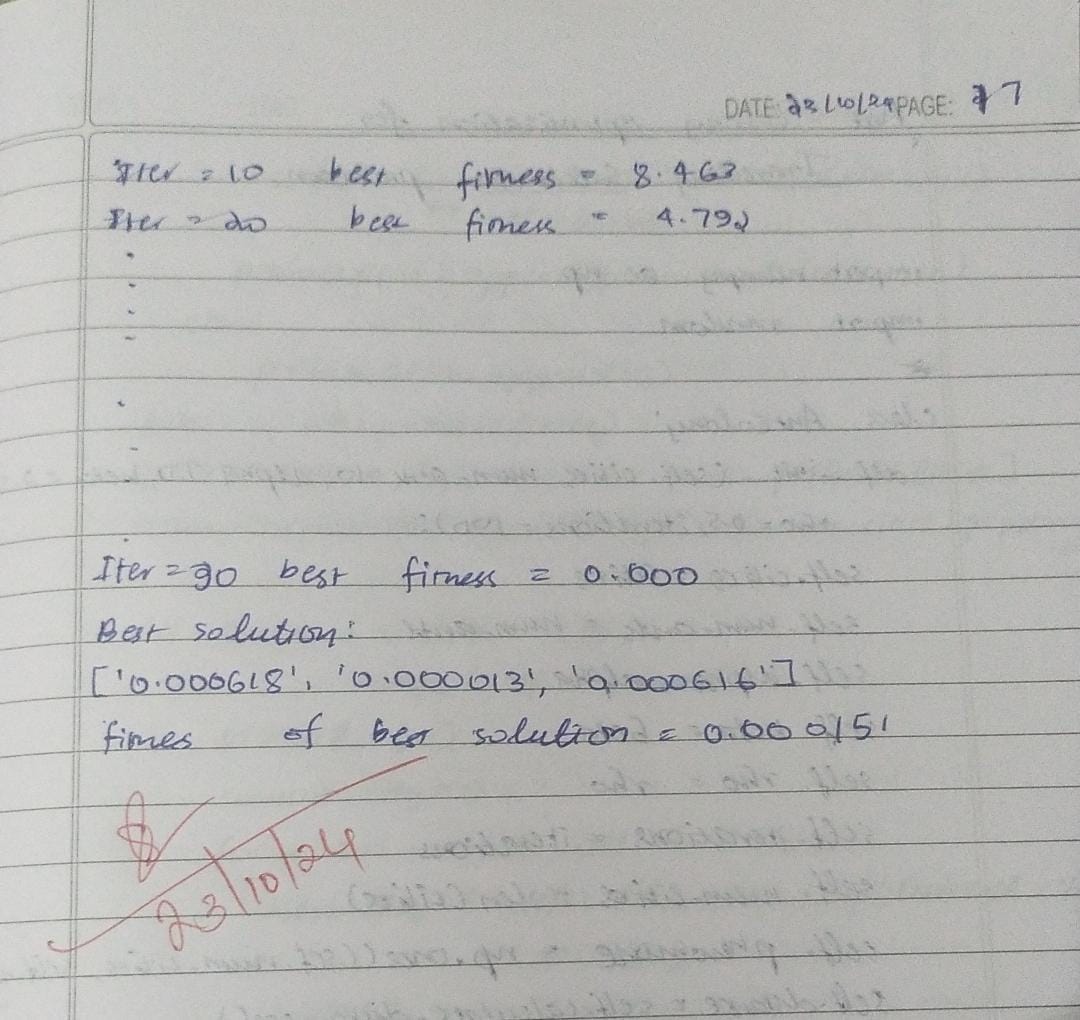
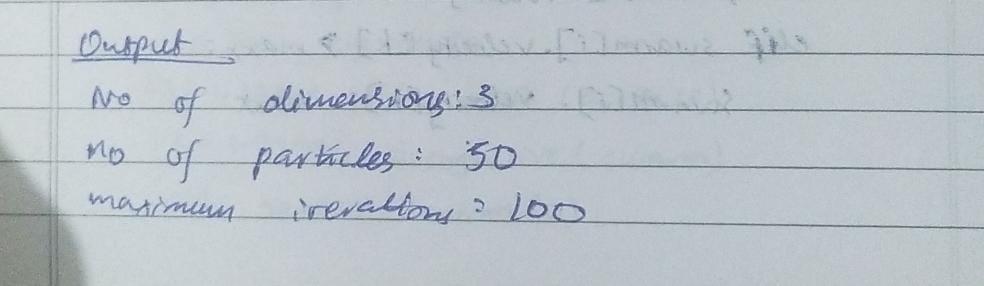
maxiter = 30

print("Starting Particle Swarm Optimization...")

optimizer = PSO(func1, initial, bounds, num\_particles, maxiter)

optimizer.optimize()

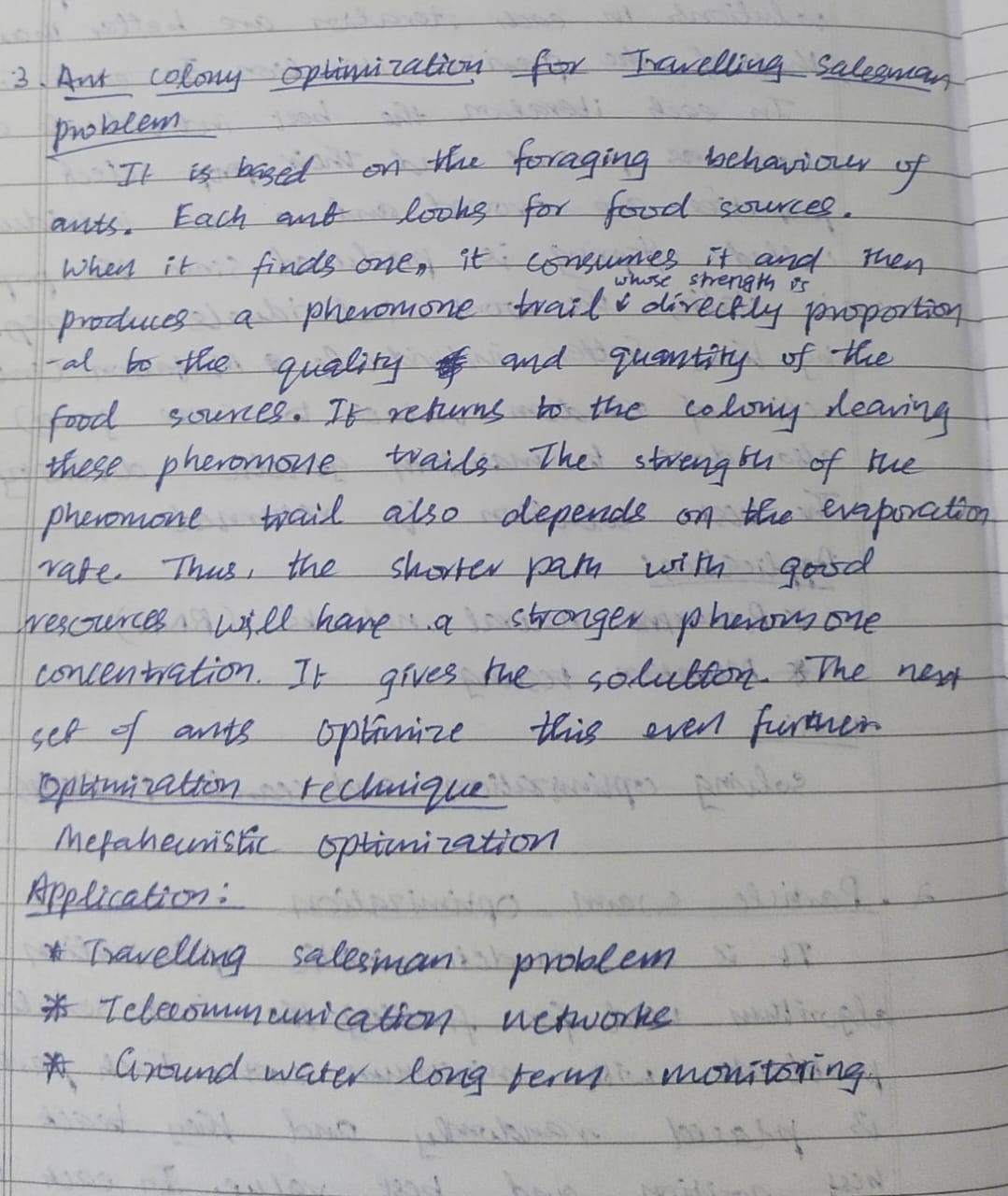
**Output:**

****

**Program 3**

**Ant Colony Optimization**

**Algorithm:**



**Code:**

# Ant Colony Optimization

import numpy as np

from numpy.random import choice as np\_choice

class AntColony(object):

def \_\_init\_\_(self, distances, n\_ants, n\_best, n\_iterations, decay, alpha=1, beta=1):

"""

Args:

distances (2D numpy.array): Square matrix of distances. Diagonal is assumed to be np.inf.

n\_ants (int): Number of ants running per iteration

n\_best (int): Number of best ants who deposit pheromone

n\_iterations (int): Number of iterations

decay (float): Rate it which pheromone decays. The pheromone value is multiplied by decay, so 0.95 will lead to decay, 0.5 to much faster decay.

alpha (int or float): Exponent on pheromone, higher alpha gives pheromone more weight. Default=1

beta (int or float): Exponent on distance, higher beta give distance more weight. Default=1

"""

self.distances = distances

self.pheromone = np.ones(self.distances.shape) / len(distances)

self.all\_inds = range(len(distances))

self.n\_ants = n\_ants

self.n\_best = n\_best

self.n\_iterations = n\_iterations

self.decay = decay

self.alpha = alpha

self.beta = beta

def run(self):

shortest\_path = None

all\_time\_shortest\_path = ("placeholder", np.inf)

for i in range(self.n\_iterations):

all\_paths = self.gen\_all\_paths()

self.spread\_pheromone(all\_paths, self.n\_best, shortest\_path=shortest\_path)

shortest\_path = min(all\_paths, key=lambda x: x[1])

print(f"Iteration {i + 1}, shortest path: {shortest\_path}")

if shortest\_path[1] < all\_time\_shortest\_path[1]:

all\_time\_shortest\_path = shortest\_path

self.pheromone = self.pheromone \* self.decay

return all\_time\_shortest\_path

def spread\_pheromone(self, all\_paths, n\_best, shortest\_path):

sorted\_paths = sorted(all\_paths, key=lambda x: x[1])

for path, dist in sorted\_paths[:n\_best]:

for move in path:

self.pheromone[move] += 1.0 / self.distances[move]

def gen\_path\_dist(self, path):

total\_dist = 0

for ele in path:

total\_dist += self.distances[ele]

return total\_dist

def gen\_all\_paths(self):

all\_paths = []

for i in range(self.n\_ants):

path = self.gen\_path(0)

all\_paths.append((path, self.gen\_path\_dist(path)))

return all\_paths

def gen\_path(self, start):

path = []

visited = set()

visited.add(start)

prev = start

for i in range(len(self.distances) - 1):

move = self.pick\_move(self.pheromone[prev], self.distances[prev], visited)

path.append((prev, move))

prev = move

visited.add(move)

path.append((prev, start)) # going back to where we started

return path

def pick\_move(self, pheromone, dist, visited):

pheromone = np.copy(pheromone)

pheromone[list(visited)] = 0

row = pheromone \*\* self.alpha \* ((1.0 / dist) \*\* self.beta)

norm\_row = row / row.sum()

move = np\_choice(self.all\_inds, 1, p=norm\_row)[0]

return move

# Example usage

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

distances = np.array([[np.inf, 2, 2, 5, 7],

[2, np.inf, 4, 8, 2],

[2, 4, np.inf, 1, 3],

[5, 8, 1, np.inf, 2],

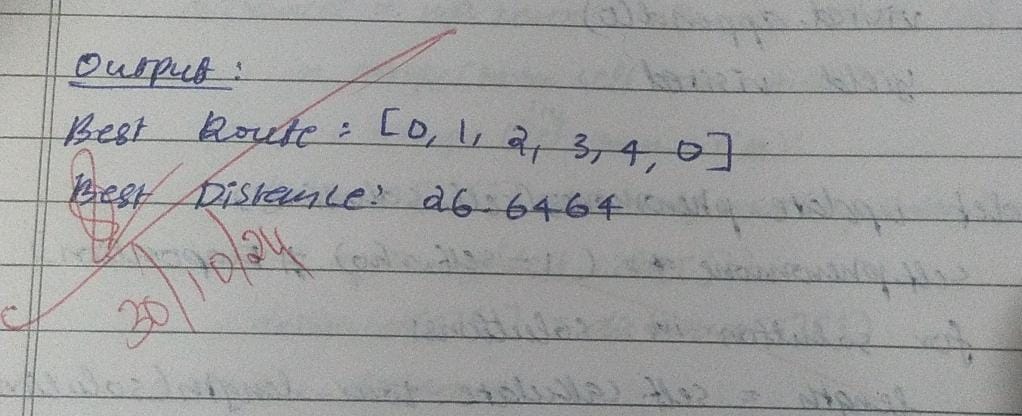
[7, 2, 3, 2, np.inf]])

ant\_colony = AntColony(distances, n\_ants=5, n\_best=1, n\_iterations=10, decay=0.95, alpha=1, beta=1)

shortest\_path = ant\_colony.run()

print("Final shortest path:", shortest\_path)

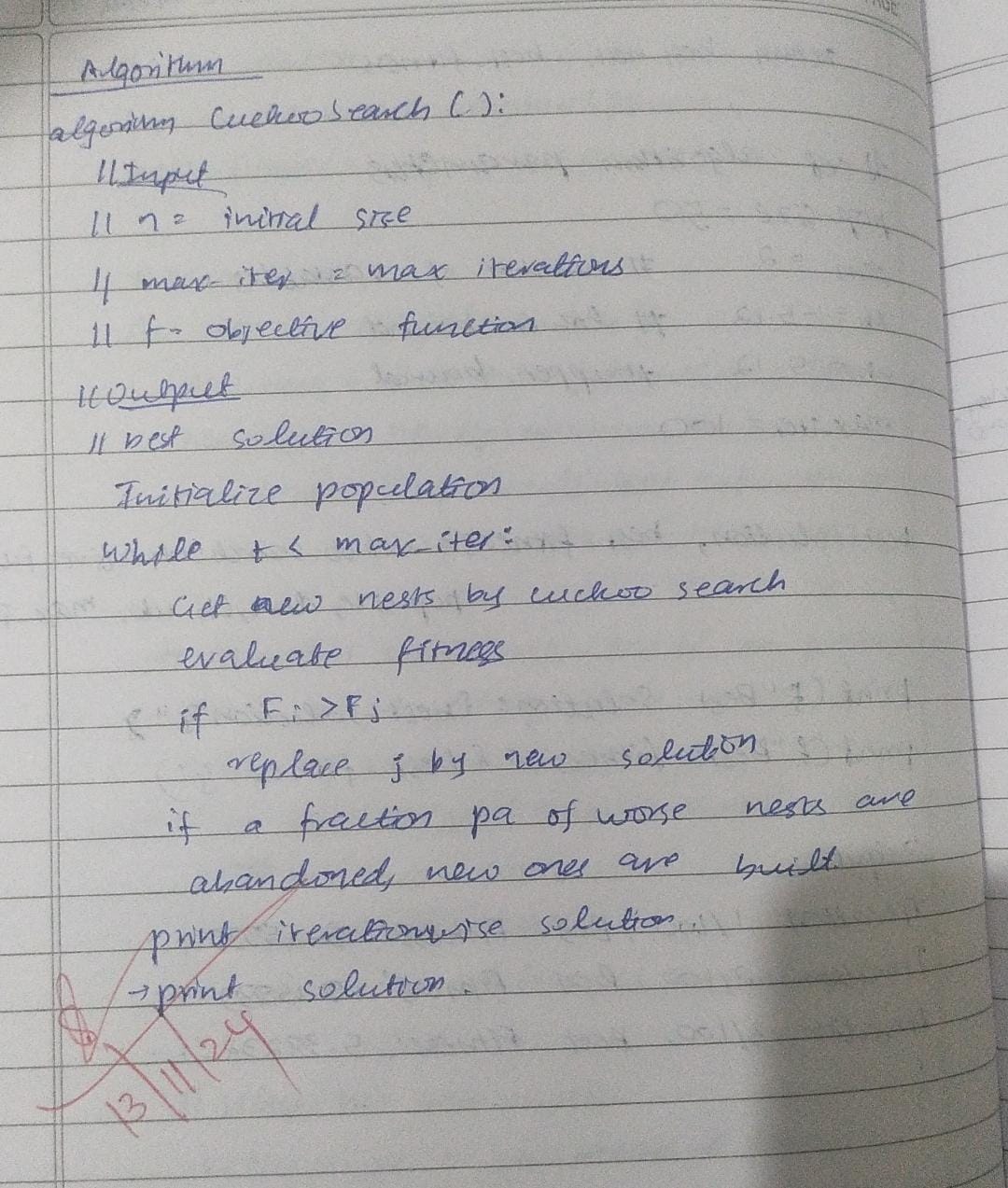
**Output:**



**Program 4**

**Cuckoo Search Algorithm**

**Algorithm:**

****

**Code:**

# Cuckoo Search Algorithm

import random

import networkx as nx

import numpy as np

import math

import datetime

import pandas as pd

class Cuckoo:

def \_\_init\_\_(self, path, G, eps = 0.9):

self.path = path

self.G = G

self.nodes = list(G.nodes)

self.eps = eps

self.fitness = self.calculate\_fitness()

"""

Function to Compute fitness value.

"""

def calculate\_fitness(self):

fitness = 0.0

for i in range(1, len(self.path)):

total\_distance = 0

curr\_node = self.path[i-1]

next\_node = self.path[i]

if self.G.has\_edge(curr\_node, next\_node):

fitness += self.G[curr\_node][next\_node]['weight']

else:

fitness += 0

fitness = np.power(abs(fitness + self.eps), 2)

return fitness

def generate\_new\_path(self):

"""

This function generates a random solution (a random path) in the graph

"""

nodes = list(self.G.nodes)

start = nodes[0]

end = nodes[-1]

samples = list(nx.all\_simple\_paths(self.G, start, end))

for i in range(len(samples)):

if len(samples[i]) != len(nodes):

extra\_nodes = [node for node in nodes if node not in samples[i]]

random.shuffle(extra\_nodes)

samples[i] = samples[i] + extra\_nodes

sample\_node = random.choice(samples)

return sample\_node

class CuckooSearch:

def \_\_init\_\_(self, G, num\_cuckoos, max\_iterations, beta):

self.G = G

self.nodes = list(G.nodes)

self.num\_cuckoos = num\_cuckoos

self.max\_iterations = max\_iterations

self.beta = beta

self.cuckoos = [Cuckoo(random.sample(self.nodes, len(self.nodes)), self.G) for \_ in range(self.num\_cuckoos)]

self.test\_results = []

self.test\_cases = 0

"""

Function to buld new nests at new location and abandon old ones using Levi flights.

"""

def levy\_flight(self):

sigma = (math.gamma(1 + self.beta) \* np.sin(np.pi \* self.beta / 2) / (math.gamma((1 + self.beta) / 2) \* self.beta \* 2 \*\* ((self.beta - 1) / 2))) \*\* (1 / self.beta)

u = np.random.normal(0, sigma, 1)

v = np.random.normal(0, 1, 1)

step = u / (abs(v) \*\* (1 / self.beta))

return step

def optimize(self):

for i in range(self.max\_iterations):

for j in range(self.num\_cuckoos):

cuckoo = self.cuckoos[j]

step = self.levy\_flight()

new\_path = cuckoo.generate\_new\_path()

new\_cuckoo = Cuckoo(new\_path, self.G)

if new\_cuckoo.fitness > cuckoo.fitness:

self.cuckoos[j] = new\_cuckoo

self.test\_cases+=1

self.cuckoos = sorted(self.cuckoos, key=lambda x: x.fitness, reverse=True)

best\_path=self.cuckoos[0].path

best\_fitness=self.cuckoos[0].fitness

self.test\_results.append([i, best\_fitness, self.test\_cases])

last\_node = list(self.G.nodes)[-1]

last\_node\_index = best\_path.index(last\_node) + 1

return best\_path[:last\_node\_index], best\_fitness

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

"""

Example usage

"""

Gn = nx.DiGraph()

#Add nodes to the graph

for i in range(11):

Gn.add\_node(i)

edges = [(0, 1,{'weight': 1}), (1, 3,{'weight': 2}), (1, 2,{'weight': 1}),(2, 4,{'weight': 2}),

(3, 2,{'weight': 2}),(3, 4,{'weight': 1}),(3, 5,{'weight': 2}),(3, 7,{'weight': 4}),

(4, 5,{'weight': 1}),(4, 6,{'weight': 2}),(5, 7,{'weight': 2}),(5, 8,{'weight': 3}),

(6, 7,{'weight': 1}),(7, 9,{'weight': 2}),(8, 10,{'weight': 2}),(9, 10,{'weight': 1})]

Gn.add\_edges\_from(edges)

csa = CuckooSearch(Gn, num\_cuckoos = 30, max\_iterations=1000, beta=0.27)

start = datetime.datetime.now()

best\_path, best\_fitness = csa.optimize()

end = datetime.datetime.now()

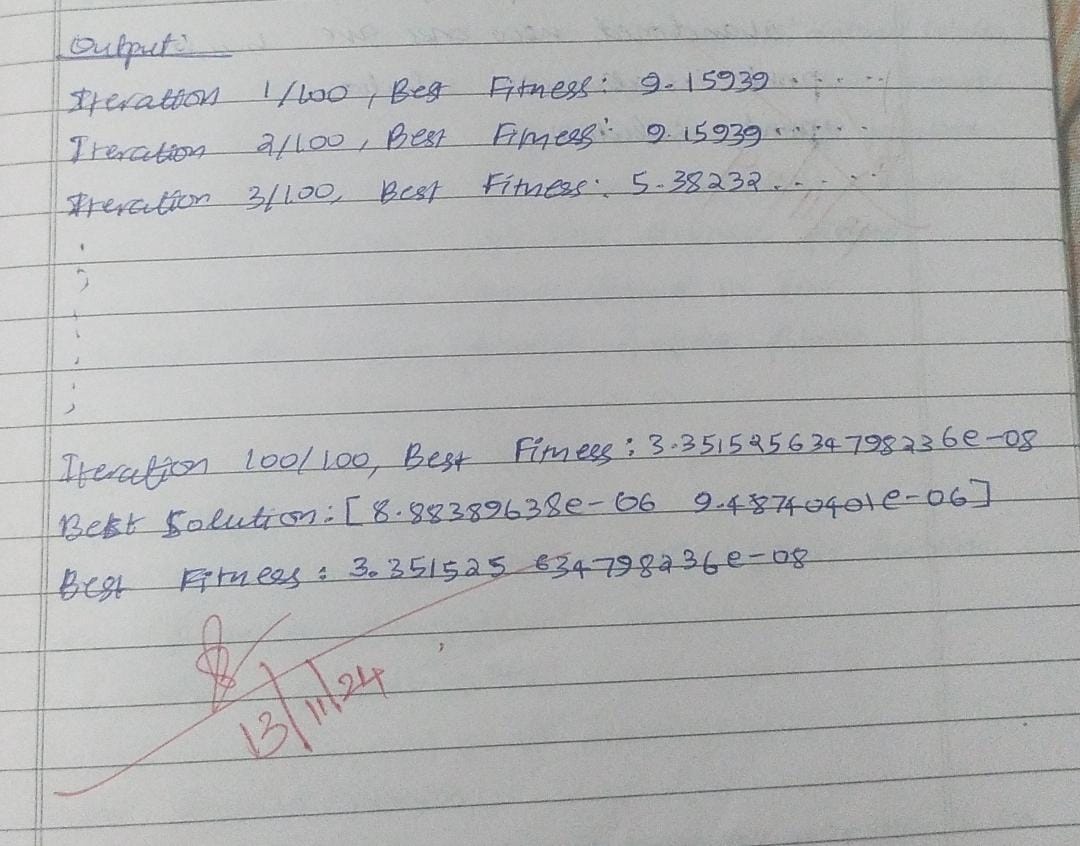
csa\_time = end - start

csa\_test\_data = pd.DataFrame(csa.test\_results,columns = ["iterations","fitness\_value","test\_cases"])

print("Optimal path: ", best\_path)

print("Optimal path cost: ", best\_fitness)

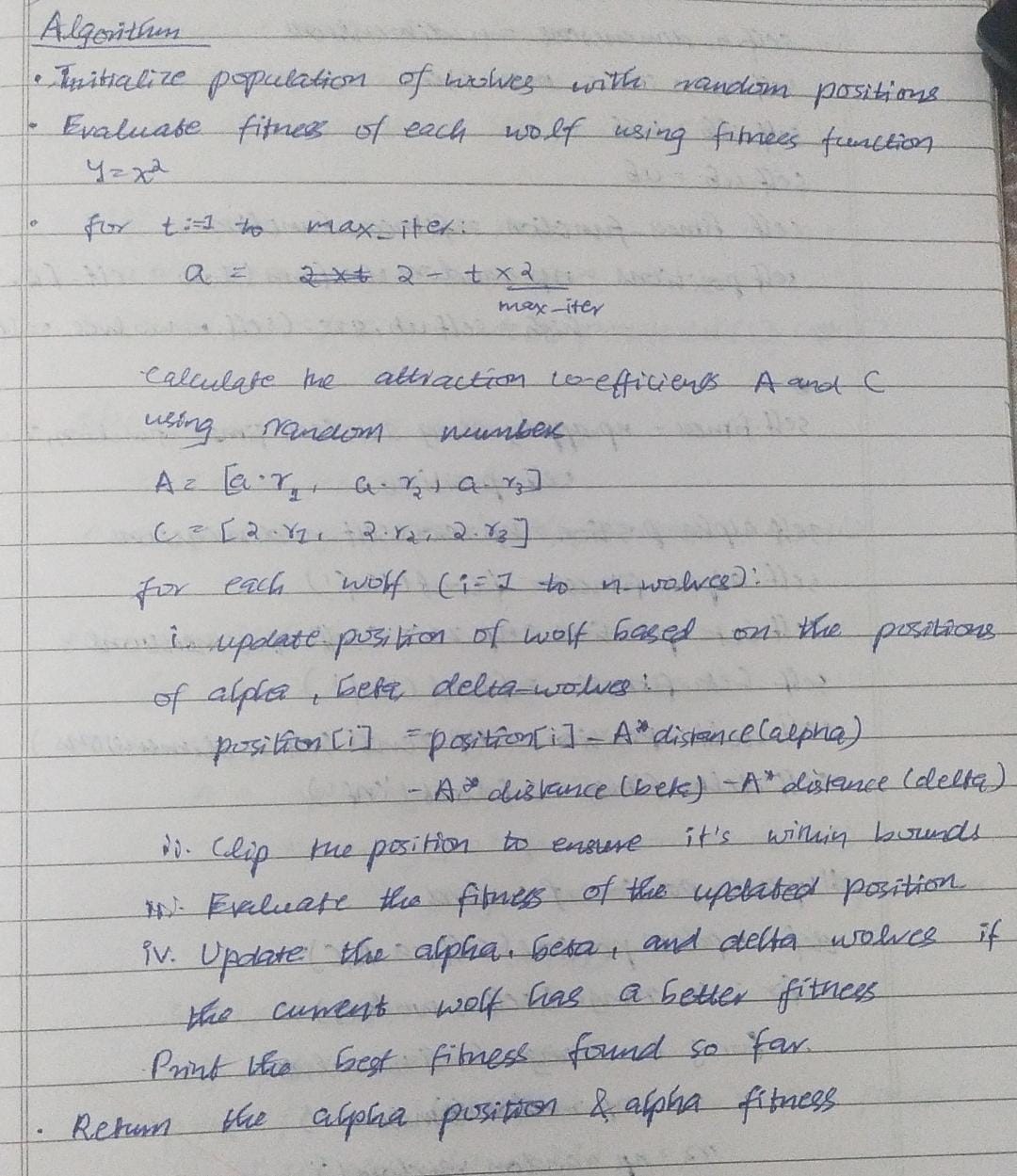
**Output:**



**Program 5**

**Gray Wolf Optimizer**

**Algorithm:**

****

**Code:**

import random

import math

import copy

# Fitness functions

def fitness\_rastrigin(position):

return sum(x\*\*2 - 10 \* math.cos(2 \* math.pi \* x) + 10 for x in position)

def fitness\_sphere(position):

return sum(x\*\*2 for x in position)

# Wolf class

class Wolf:

def \_\_init\_\_(self, fitness, dim, minx, maxx, seed):

self.rnd = random.Random(seed)

self.position = [self.rnd.uniform(minx, maxx) for \_ in range(dim)]

self.fitness = fitness(self.position)

def gwo(fitness, max\_iter, n, dim, minx, maxx):

rnd = random.Random(0)

population = [Wolf(fitness, dim, minx, maxx, i) for i in range(n)]

population.sort(key=lambda w: w.fitness)

alpha\_wolf, beta\_wolf, gamma\_wolf = copy.deepcopy(population[:3])

for Iter in range(max\_iter):

if Iter % 10 == 0 and Iter > 0:

print(f"Iter = {Iter} best fitness = {alpha\_wolf.fitness:.3f}")

a = 2 \* (1 - Iter / max\_iter)

for wolf in population:

A1, A2, A3 = [a \* (2 \* rnd.random() - 1) for \_ in range(3)]

C1, C2, C3 = [2 \* rnd.random() for \_ in range(3)]

X1 = [alpha\_wolf.position[j] - A1 \* abs(C1 \* alpha\_wolf.position[j] - wolf.position[j]) for j in range(dim)]

X2 = [beta\_wolf.position[j] - A2 \* abs(C2 \* beta\_wolf.position[j] - wolf.position[j]) for j in range(dim)]

X3 = [gamma\_wolf.position[j] - A3 \* abs(C3 \* gamma\_wolf.position[j] - wolf.position[j]) for j in range(dim)]

Xnew = [(X1[j] + X2[j] + X3[j]) / 3.0 for j in range(dim)]

fnew = fitness(Xnew)

if fnew < wolf.fitness:

wolf.position = Xnew

wolf.fitness = fnew

population.sort(key=lambda w: w.fitness)

alpha\_wolf, beta\_wolf, gamma\_wolf = copy.deepcopy(population[:3])

return alpha\_wolf.position

# Driver code

def run\_gwo(fitness, func\_name, dim=3, num\_particles=50, max\_iter=100, minx=-10.0, maxx=10.0):

print(f"\nBegin grey wolf optimization on {func\_name} function\n")

print(f"Goal is to minimize {func\_name} in {dim} variables")

print(f"Function has known min = 0.0 at ({', '.join(['0'] \* dim)})")

print(f"Setting num\_particles = {num\_particles}")

print(f"Setting max\_iter = {max\_iter}\n")

best\_position = gwo(fitness, max\_iter, num\_particles, dim, minx, maxx)

print(f"\nGWO completed\n")

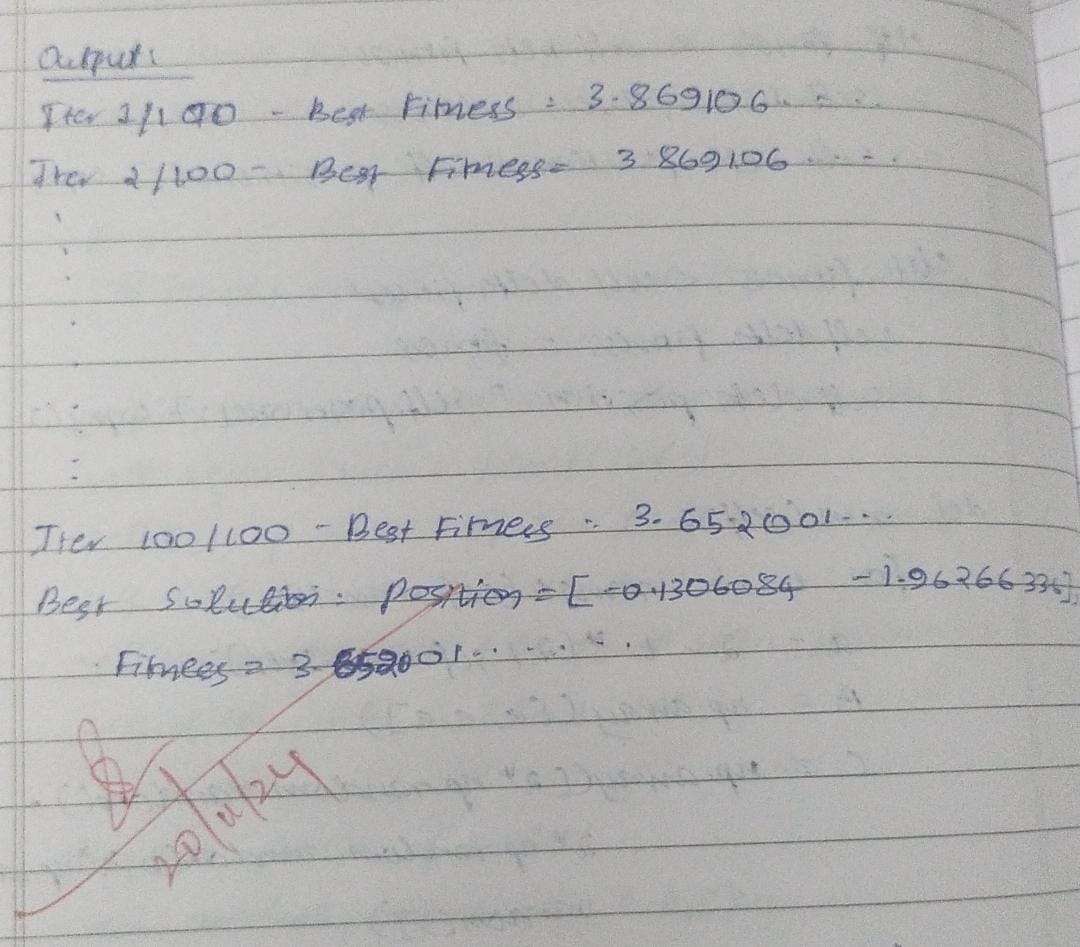
print(f"Best solution found: {[f'{x:.6f}' for x in best\_position]}")

print(f"Fitness of best solution = {fitness(best\_position):.6f}\n")

# Run GWO for Rastrigin and Sphere functions

run\_gwo(fitness\_rastrigin, "Rastrigin")

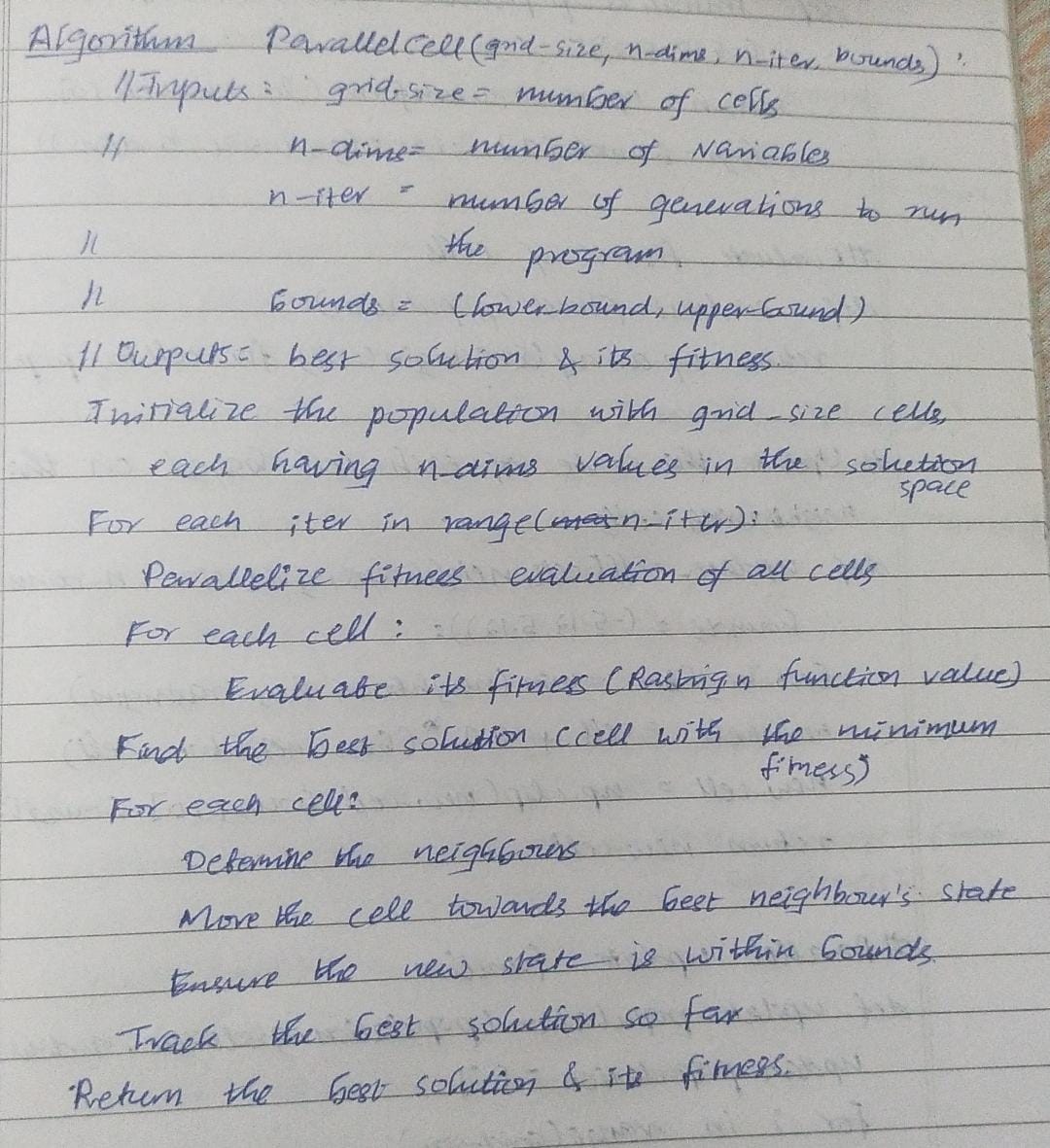
**Output**:



**Program 6**

**Parallel Cell Algorithm**

**Algorithm:**

****

**Code:**

#Parallel Cell Algorithm

import numpy as np

from concurrent.futures import ThreadPoolExecutor

# Define the grid size

grid\_size = 5 # Set a smaller grid for easier display, e.g., 10x10

grid = np.random.randint(0, 2, (grid\_size, grid\_size))

# Define a simple rule (e.g., Conway's Game of Life)

def game\_of\_life\_cell(i, j, grid):

# Count live neighbors

neighbors = grid[i-1:i+2, j-1:j+2]

total = np.sum(neighbors) - grid[i, j]

if grid[i, j] == 1:

return 1 if total in [2, 3] else 0

else:

return 1 if total == 3 else 0

def update\_cell(i, j, grid):

return game\_of\_life\_cell(i, j, grid)

def parallel\_update(grid):

# Create a copy of the grid to store updated values

new\_grid = grid.copy()

with ThreadPoolExecutor() as executor:

futures = []

for i in range(1, grid.shape[0]-1):

for j in range(1, grid.shape[1]-1):

futures.append(executor.submit(update\_cell, i, j, grid))

# Collect results from the futures and update the new grid

idx = 0

for future in futures:

i, j = divmod(idx, grid.shape[1] - 2)

new\_grid[i+1, j+1] = future.result()

idx += 1

return new\_grid

# Update the grid

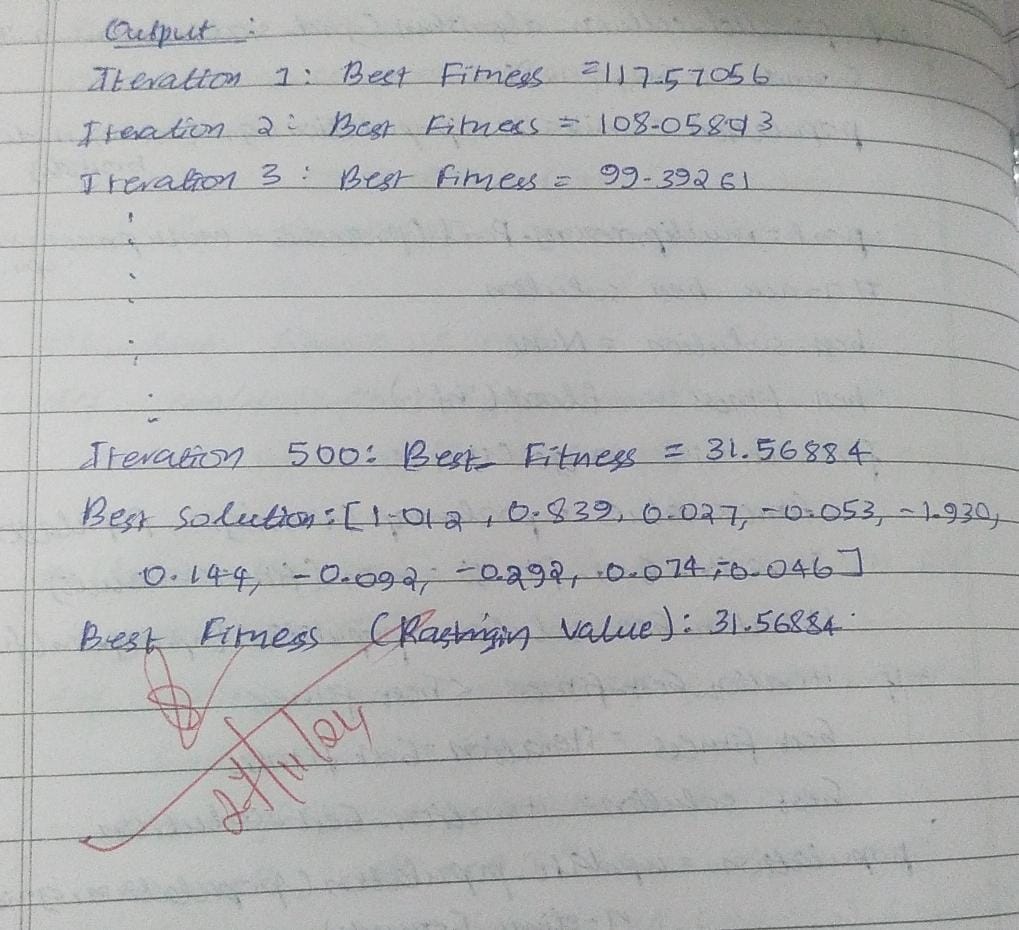
grid = parallel\_update(grid)

# Display the grid as a matrix (instead of plotting)

print("Updated Grid after applying Conway's Game of Life rule:")

print(grid)

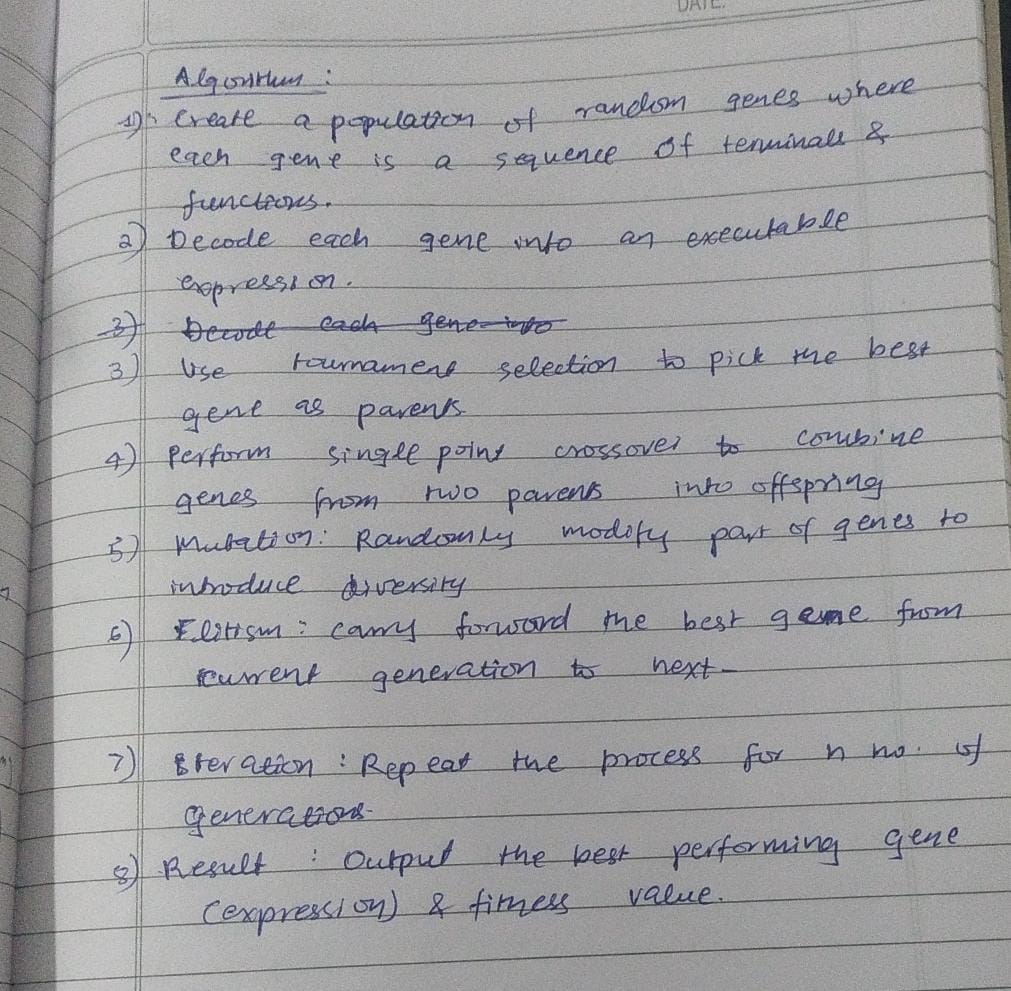
**Output:**

****

**Program 7**

**Gene Expression Algorithm**

**Algorithm:**

****

**Code:**

# Optimization via gene expression

import random

# Fitness function: minimize f(x) = x^2

def fitness(x):

return -x\*\*2 # Negative because higher fitness is better

# Generate initial population

def generate\_population(size, lower, upper):

return [random.uniform(lower, upper) for \_ in range(size)]

# Crossover

def crossover(parent1, parent2):

alpha = random.random()

return alpha \* parent1 + (1 - alpha) \* parent2

# Mutation

def mutate(gene, mutation\_rate, lower, upper):

if random.random() < mutation\_rate:

return random.uniform(lower, upper)

return gene

# Main Genetic Algorithm

def genetic\_algorithm(pop\_size, generations, mutation\_rate, lower, upper):

population = generate\_population(pop\_size, lower, upper)

for \_ in range(generations):

# Evaluate fitness

population.sort(key=fitness, reverse=True)

new\_population = []

# Selection and reproduction

for i in range(pop\_size // 2):

parent1, parent2 = random.choices(population[:pop\_size // 2], k=2)

child = mutate(crossover(parent1, parent2), mutation\_rate, lower, upper)

new\_population.extend([parent1, child])

population = new\_population

# Best solution

best\_gene = max(population, key=fitness)

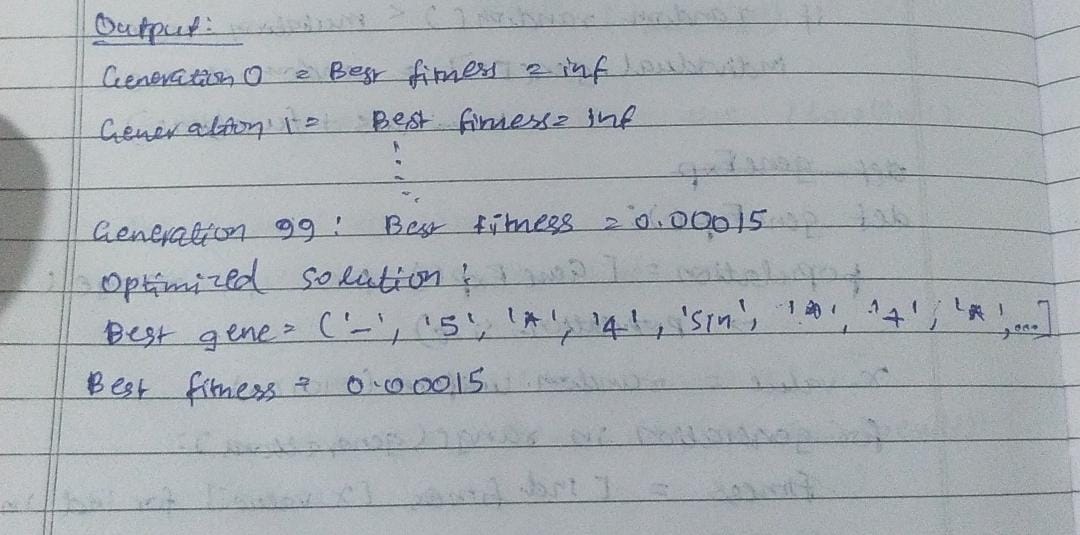
return best\_gene, -fitness(best\_gene)

# Run the algorithm

best\_solution, best\_fitness = genetic\_algorithm(50, 100, 0.1, -10, 10)

print(f"Best Solution: {best\_solution}, Fitness: {best\_fitness}")

**Output:**

****