VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Navneet Kumar (1BM23CS207)

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

Aug-2025 to Dec-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

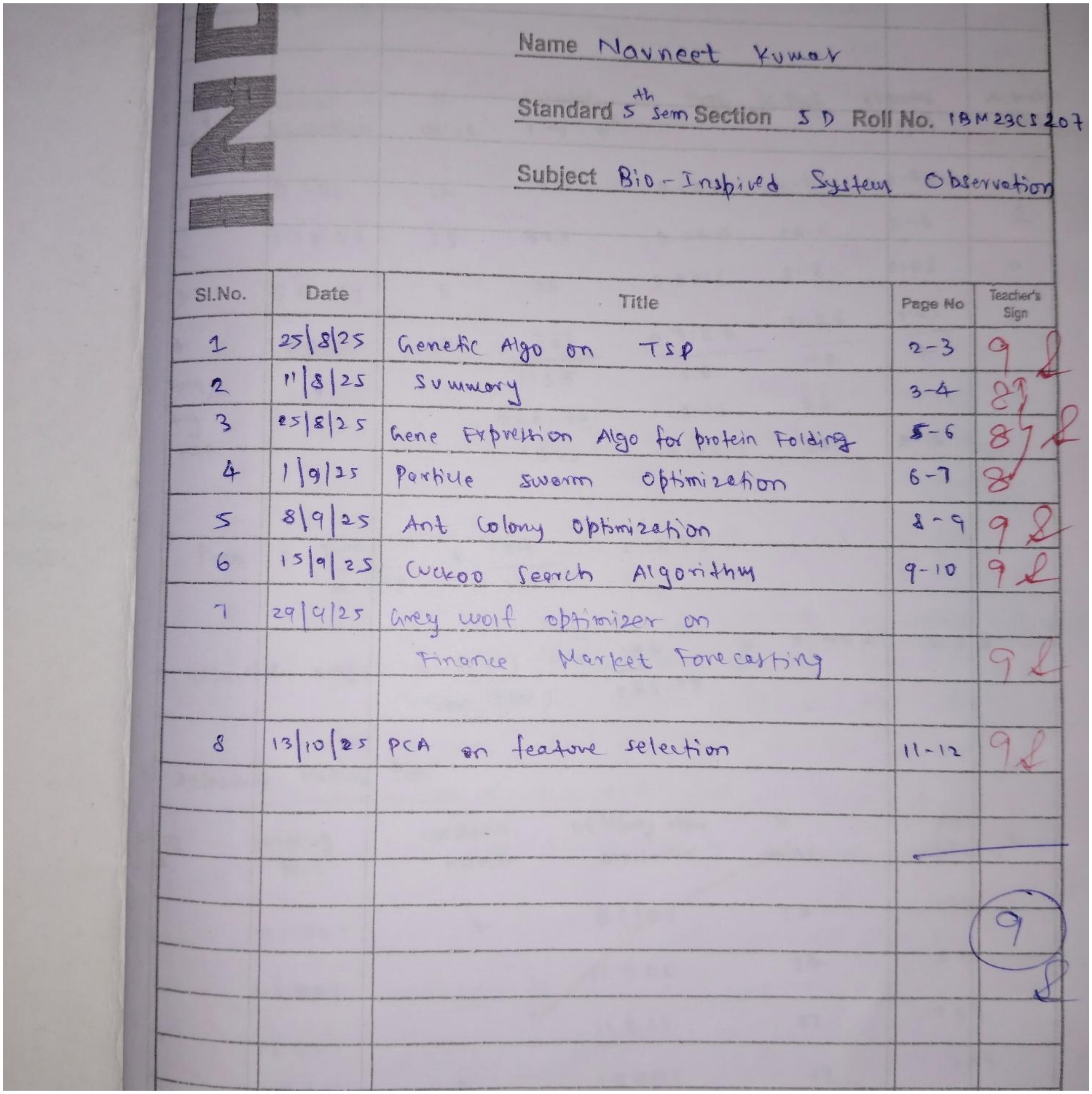
<u>CERTIFICATE</u>

This is to certify that the Lab work entitled “ Bio Inspired Systems (23CS5BSBIS)” carried out by **Navneet Kumar (1BM23CS207),** who is a 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.

Sneha S Bagalkot Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1	18/8/25	Genetic Algorithm	1
2	25/8/25	Gene Expression Algorithm	5
3	1/9/25	Particle Swarm Optimisation	8
4	8/9/25	Ant Colony Optimisation	11
5	15/9/25	Cuckoo Search Optimisation	15
6	29/9/25	Grey Wolf Optimisation	19
7	13/10/25	Parallel Cellular Algorithm	22



Name Navneet Kumar

Standard 5th sem Section 5 D Roll No. 1BM23CS207

Subject Bio-Inspired System Observation

Sl.No.	Date	Title	Page No	Teacher's Sign
1	25/8/25	Genetic Algo on TSP	2-3	9
2	11/8/25	Summary	3-4	8
3	25/8/25	Gene Expression Algo for protein Folding	5-6	8
4	1/9/25	Particle swarm optimization	6-7	8
5	8/9/25	Ant Colony optimization	8-9	9
6	15/9/25	Cuckoo Search Algorithm	9-10	9
7	29/9/25	Grey wolf optimizer on Finance Market Forecasting		9
8	13/10/25	PCA on feature selection	11-12	9

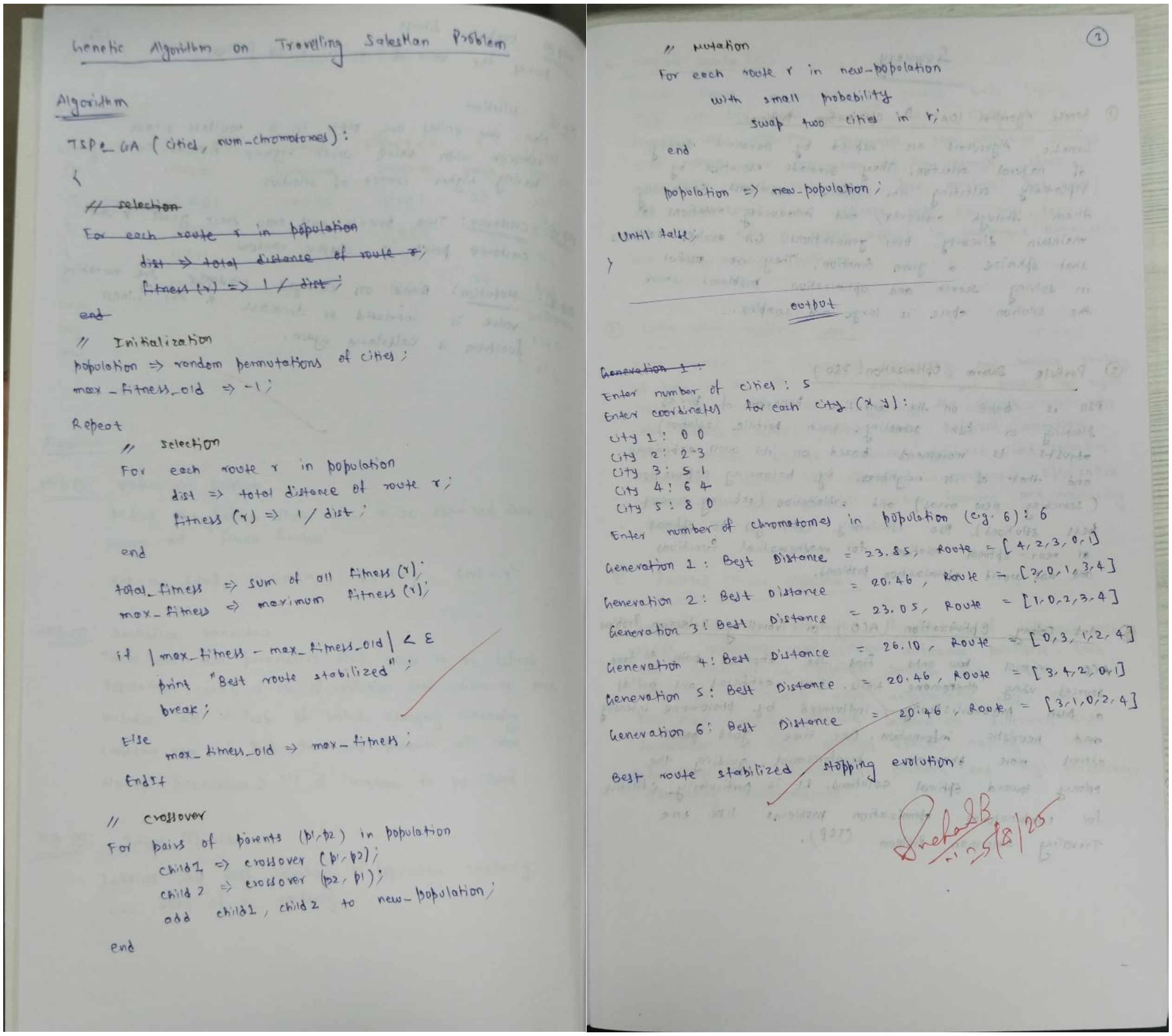
9

Github Link:
https://github.com/navneet207/Bio_Inspired_Systems

Program 1

Design and implement a **genetic algorithm** in Python that evolves a population of random character strings to match a given target phrase ("HELLO WORLD").

Algorithm:



Genetic Algorithm on Travelling SalesMan Problem

Algorithm

```
TSP_GA ( cities, num_chromotomes ):
{
    ~~// selection~~
    ~~For each route r in population~~
        ~~dist => total distance of route r;~~
        ~~fitness (r) => 1 / dist;~~
    ~~end~~
 // Initialization
 population => random permutations of cities;
 max_fitness_old => -1;

 Repeat
     // selection
     For each route r in population
         dist => total distance of route r;
         fitness (r) => 1 / dist;
     end

     total_fitness => sum of all fitness (r);
     max_fitness => maximum fitness (r);

     if | max_fitness - max_fitness_old | < ε
         print "Best route stabilized";
         break;
     Else
         max_fitness_old => max_fitness;
     EndIf

     // crossover
     For pairs of parents (p1, p2) in population
         child1 => crossover (p1, p2);
         child2 => crossover (p2, p1);
         add child1, child2 to new_population;
     end

     // mutation
     For each route r in new_population
         with small probability
             swap two cities in r;
     end

     population => new_population;

 Until false;
}
```

output

```
~~Generation 1:~~
Enter number of cities : 5
Enter coordinates for each city (x y):
City 1: 0 0
City 2: 2 3
City 3: 5 1
City 4: 6 4
City 5: 8 0
Enter number of chromotomes in population (e.g. 6): 6
Generation 1: Best Distance = 23.85, Route = [4, 2, 3, 0, 1]
Generation 2: Best Distance = 20.46, Route = [2, 0, 1, 3, 4]
Generation 3: Best Distance = 23.05, Route = [1, 0, 2, 3, 4]
Generation 4: Best Distance = 26.10, Route = [0, 3, 1, 2, 4]
Generation 5: Best Distance = 20.46, Route = [3, 4, 2, 0, 1]
Generation 6: Best Distance = 20.46, Route = [3, 1, 0, 2, 4]

Best route stabilized, stopping evolution.
```

25/8/25

Code:

```python
import random
import string

# Genetic Algorithm Parameters
TARGET = "HELLO WORLD"
POP_SIZE = 100
MUTATION_RATE = 0.01
GENERATIONS = 1000
CHARS = string.ascii_uppercase + " "

def fitness(ind):
    """Measure how close the individual is to the target string."""
    return sum(c1 == c2 for c1, c2 in zip(ind, TARGET))

def generate_individual():
    """Generate a random string of the same length as the target."""
    return ''.join(random.choice(CHARS) for _ in TARGET)

def mutate(ind):
    """Randomly mutate characters based on the mutation rate."""
    return ''.join(
        c if random.random() > MUTATION_RATE else random.choice(CHARS)
        for c in ind
    )

def crossover(p1, p2):
    """Combine parts of two parents to create a child."""
    point = random.randint(0, len(TARGET) - 1)
    return p1[:point] + p2[point:]

def selection(pop):
    """Tournament selection: choose the fittest from a random subset."""
    return max(random.sample(pop, 5), key=fitness)

def genetic_algorithm():
    """Run the genetic algorithm until the target is found or max generations are
reached."""
    random.seed(42)  # Optional: makes results reproducible
    population = [generate_individual() for _ in range(POP_SIZE)]

    for gen in range(GENERATIONS):
        # Sort by fitness (best first)
        population = sorted(population, key=fitness, reverse=True)
        best = population[0]
        print(f"Gen {gen}: {best} (Fitness = {fitness(best)})")
```

```python
        # Stop if target is reached
        if best == TARGET:
            break

        # Elitism: keep top 2
        next_gen = population[:2]

        # Create new generation
        while len(next_gen) < POP_SIZE:
            p1 = selection(population)
            p2 = selection(population)
            child = crossover(p1, p2)
            child = mutate(child)
            next_gen.append(child)

        population = next_gen

    print("\nBest Match:", best)
    return best

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

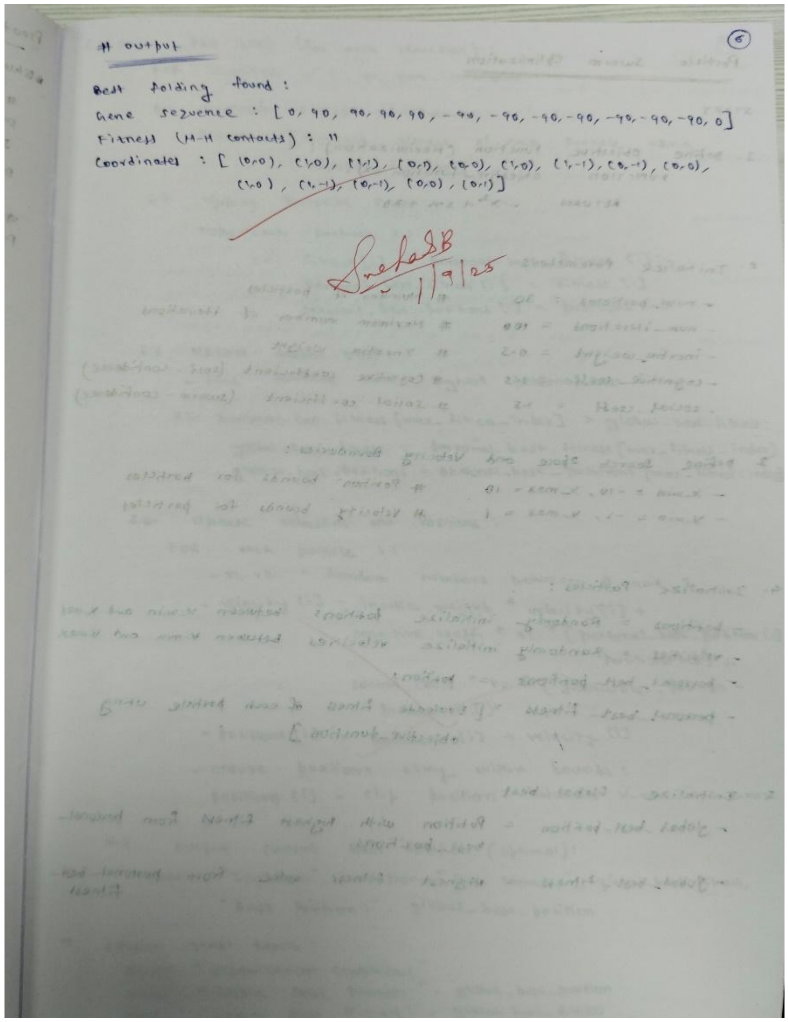
Program 2

Use a **gene expression** algorithm to find the optimal sequence for CPU process scheduling that minimizes the average waiting time of processes given their burst times.

Algorithm:

Pseudo Code: Gene Expression Algorithm for Protein Folding (HP Model) 5

```
BEGIN

# --- Parameters ---
Define protein sequence (H = hydrophobic, P = polar)
Set population size, mutation rate, crossover rate, generations
Define possible moves = { straight, left, right }

# --- Helper Functions ---
FUNCTION FoldProtein (sequence, gene):
    Initialize start position (x=0, y=0) and direction = east
    Store coordinates of amino acids
    FOR each move in gene:
        Update direction based on turn
        Move one step in direction
        Append one new position to coordinates
    RETURN coordinates

FUNCTION Fitness (sequence, coordinates):
    Score = 0
    FOR each amino acid i:
        IF sequence [i] = H :
            FOR each amino acid j (not consecutive to i):
                IF sequence [j] = H AND distance (coords [i],
                                                 coords [j]) = 1 :
                    Increase score by 1
    RETURN score

FUNCTION RandomGene (length):
    RETURN random sequence of moves (length-1)

FUNCTION Select (population)
    Randomly pick 2 solutions
    RETURN one with better fitness

FUNCTION Crossover (parent1, parent2):
    IF random > crossover_rate :
        RETURN copy of parent1
    choose random crossover point
    RETURN child = part of parent1 + part of parent2

FUNCTION Mutate (gene):
    FOR each move in gene :
        IF random < mutation-rate :
            Replace with new random move
    RETURN mutated gene

# --- Initialization ---
population = []
FOR i in 1 to population-size :
    gene = RandomGene (length of protein)
    coords = FoldProtein (protein, gene)
    fit = Fitness (protein, coords)
    Add (gene, fit) to population

# --- Evolution Loop ---
FOR generation in 1 to generations:
    new-population = []
    FOR i in 1 to population-size :
        parent1 = Select (population)
        parent 2 = Select (population)
        child_gene = Crossover (parent1, parent2)
        child_gene = Mutate (child-gene)
        coords = FoldProtein (protein, child-gene)
        fit = Fitness (protein, coords)
        Add (child_gene, fit) to new_population
    population = new_population
    Print best fitness in this generation

# --- Final output ---
best_solution = solution with max fitness
Print best gene, best fitness, and coordinates

END
```



output

Best folding found :
Gene sequence : [0, 90, 90, 90, 90, -90, -90, -90, -90, -90, -90, -90, 0]
Fitness (H-H contacts) : 11
Coordinates : [(0,0), (1,0), (1,1), (0,1), (0,0), (1,0), (1,-1), (0,-1), (0,0), (1,0), (1,-1), (0,-1), (0,0), (0,1)]

1/9/25

Code:

```
import random

# ---------------------------------------------
# Calculate average waiting time for a schedule
# ---------------------------------------------
def avg_waiting_time(schedule, burst_times):
    waiting_time = 0
```

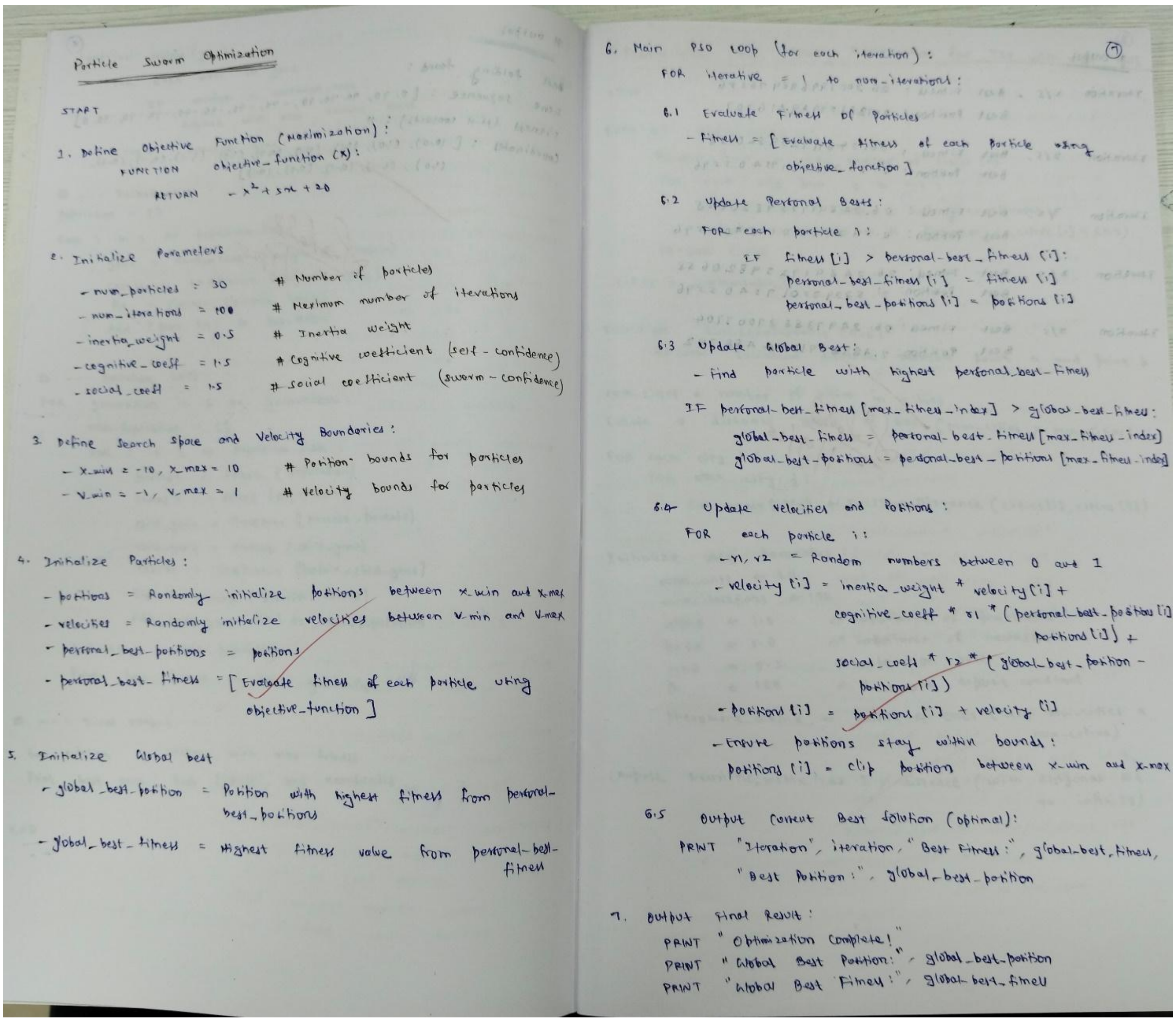
```
    total_waiting = 0
    for p in schedule[:-1]:
        total_waiting += waiting_time
        waiting_time += burst_times[p]
    return total_waiting / len(schedule)

# ----------------------------------------------
# Fitness function (maximize inverse of waiting time)
# ----------------------------------------------
def fitness(schedule, burst_times):
    return 1 / (1 + avg_waiting_time(schedule, burst_times))

# ----------------------------------------------
# Generate a random schedule (random permutation)
# ----------------------------------------------
def random_schedule(n):
    schedule = list(range(n))
    random.shuffle(schedule)
    return schedule

# ----------------------------------------------
# Tournament selection
# ----------------------------------------------
def selection(population, burst_times):
    contenders = random.sample(population, 3)
    return max(contenders, key=lambda s: fitness(s, burst_times))

# ----------------------------------------------
# Ordered crossover for scheduling
# ----------------------------------------------
def crossover(p1, p2):
    a, b = sorted(random.sample(range(len(p1)), 2))
    child = [None] * len(p1)
    child[a:b] = p1[a:b]
    fill = [p for p in p2 if p not in child]
    idx = 0
    for i in range(len(p1)):
        if child[i] is None:
            child[i] = fill[idx]
            idx += 1
    return child

# ----------------------------------------------
# Mutation: swap two processes
# ----------------------------------------------
def mutate(schedule, rate=0.2):
```

```
    for _ in range(len(schedule)):
        if random.random() < rate:
            i, j = random.sample(range(len(schedule)), 2)
            schedule[i], schedule[j] = schedule[j], schedule[i]
    return schedule

# ---------------------------------------------
# Genetic Algorithm for CPU Scheduling
# ---------------------------------------------
def gene_expression_scheduler(burst_times, pop_size=30, generations=200):
    n = len(burst_times)
    population = [random_schedule(n) for _ in range(pop_size)]
    best = None

    for g in range(generations + 1):  # include final generation
        population.sort(key=lambda s: fitness(s, burst_times), reverse=True)
        if best is None or fitness(population[0], burst_times) > fitness(best,
burst_times):
            best = population[0]

        new_pop = []
        while len(new_pop) < pop_size:
            p1 = selection(population, burst_times)
            p2 = selection(population, burst_times)
            child = crossover(p1, p2)
            child = mutate(child)
            new_pop.append(child)

        population = new_pop

        # Print progress occasionally
        if g == 0 or g == 10 or g % 50 == 0:
            print(f"Gen {g}: Best Avg Waiting Time = {avg_waiting_time(best,
burst_times):.2f}")

    return best

# ---------------------------------------------
# Example usage
# ---------------------------------------------
if __name__ == "__main__":
    burst_times = [5, 2, 8, 3, 6]  # CPU burst times for processes P1...P5
    best_schedule = gene_expression_scheduler(burst_times)

    print("\nBest Schedule Found:", best_schedule)
    print("Avg Waiting Time:", avg_waiting_time(best_schedule, burst_times))
```

Program 3

Find the maximum of the function f(x)=−x^2+20x+5 using **Particle Swarm Optimization** by iteratively updating particle positions and velocities.

Algorithm:



```
Particle Swarm Optimization

START

1. Define Objective Function (Maximization):
      FUNCTION objective_function (x):
          RETURN -x^2 + 5x + 20

2. Initialize Parameters
   - num_particles = 30          # Number of particles
   - num_iterations = 100        # Maximum number of iterations
   - inertia_weight = 0.5        # Inertia weight
   - cognitive_coeff = 1.5       # Cognitive coefficient (self - confidence)
   - social_coeff = 1.5          # Social coefficient (swarm - confidence)

3. Define Search Space and Velocity Boundaries:
   - x_min = -10, x_max = 10     # Position bounds for particles
   - v_min = -1, v_max = 1       # Velocity bounds for particles

4. Initialize Particles:
   - positions = Randomly initialize positions between x_min and x_max
   - velocities = Randomly initialize velocities between v_min and v_max
   - personal_best_positions = positions
   - personal_best_fitness = [Evaluate fitness of each particle using
                              objective_function]

5. Initialize Global best
   - global_best_position = Position with highest fitness from personal_
                            best_positions
   - global_best_fitness = Highest fitness value from personal_best_
                           fitness

6. Main PSO Loop (for each iteration):
   FOR iterative = 1 to num_iterations:
     6.1 Evaluate Fitness of Particles
         - fitness = [Evaluate fitness of each particle using
                      objective_function]
     6.2 Update Personal Bests:
         FOR each particle i:
              IF fitness[i] > personal_best_fitness[i]:
                   personal_best_fitness[i] = fitness[i]
                   personal_best_positions[i] = positions[i]
     6.3 Update Global Best:
         - find particle with highest personal_best_fitness
         IF personal_best_fitness[max_fitness_index] > global_best_fitness:
              global_best_fitness = personal_best_fitness[max_fitness_index]
              global_best_positions = personal_best_positions[max_fitness_index]
     6.4 Update Velocities and Positions:
         FOR each particle i:
              - r1, r2 = Random numbers between 0 and 1
              - velocity[i] = inertia_weight * velocity[i] +
                              cognitive_coeff * r1 * (personal_best_position[i] -
                                                      positions[i]) +
                              social_coeff * r2 * (global_best_position -
                                                   positions[i])
              - positions[i] = positions[i] + velocity[i]
              - Ensure positions stay within bounds:
                positions[i] = clip position between x_min and x_max
     6.5 Output Current Best Solution (optimal):
         PRINT "Iteration", iteration, "Best Fitness:", global_best_fitness,
               "Best Position:", global_best_position

7. Output Final Result:
   PRINT "Optimization Complete!"
   PRINT "Global Best Position:", global_best_position
   PRINT "Global Best Fitness:", global_best_fitness
```

output

Iteration 1/5, Best Fitness: 26.200199636976556
Best Position: 2.7231599494 16207

Iteration 2/5, Best Fitness: 26.248917259820658
Best Position: 2.53240501754 05596

Iteration 3/5, Best Fitness: 26.248917259820658
Best Position: 2.5324050175405596

Iteration 4/5, Best Fitness: 26.248917259820658
Best Position: 2.5329050175405596

Iteration 5/5, Best Fitness: 26.249738827007704
Best Position: 2.483839152448 8392

1/9/25

Code:

```
import numpy as np

# --------------------------------------------
# Objective Function
# --------------------------------------------
def f(x):
    return -x**2 + 20*x + 5

# --------------------------------------------
# Initialization
# --------------------------------------------
positions = np.array([0.6, 2.3, 2.8, 8.3, 10, 9.6, 6, 2.6, 1.1])
velocities = np.zeros_like(positions)
pbest_positions = positions.copy()
pbest_scores = f(positions)
gbest_position = pbest_positions[np.argmax(pbest_scores)]

# PSO parameters
c1 = c2 = 1
w = 1

# Random values for reproducibility (from example)
r_values = [
    (0.213, 0.876),
    (0.113, 0.706),
    (0.178, 0.507)
]

# --------------------------------------------
# Initial Information
# --------------------------------------------
print("Initial positions:\n", positions)
print("Initial function values:\n", f(positions))
print("Initial global best position:", gbest_position, "with value:",
f(gbest_position))
print("-" * 50)

# --------------------------------------------
# PSO Main Loop (3 iterations)
# --------------------------------------------
for t in range(3):
    r1, r2 = r_values[t]
```

```python
    # Update velocities
    for i in range(len(positions)):
        velocities[i] = (
            w * velocities[i]
            + c1 * r1 * (pbest_positions[i] - positions[i])
            + c2 * r2 * (gbest_position - positions[i])
        )

    # Update positions
    positions += velocities
    scores = f(positions)

    # Update personal bests
    for i in range(len(positions)):
        if scores[i] > pbest_scores[i]:
            pbest_positions[i] = positions[i]
            pbest_scores[i] = scores[i]

    # Update global best
    gbest_position = pbest_positions[np.argmax(pbest_scores)]

    # Display iteration results
    print(f"Iteration {t + 1}")
    print("Positions:", positions.round(4))
    print("Velocities:", velocities.round(4))
    print("Function values:", scores.round(4))
    print("Global best position:", gbest_position, "with value:", f(gbest_position))
    print("-" * 50)
```

Program 4

Find the shortest route visiting all cities once and returning to the start using **Ant Colony Optimization.**

Algorithm:

```
START

FUNCTION    GetUserInput():
    Prompt user to enter number of cities (n)
    For each city from 0 to n-1 :
        Prompt user to enter x and y coordinates
        Store coordinates in a dictionary : cities[i] = (x,y)
    RETURN cities

cities ← GetUserInput()

FUNCTION    EuclideanDistance(a,b):
    RETURN Euclidean distance between point a and point b

num_cities ← number of cities in cities
Create a distance matrix of size (num_cities x num_cities)

FOR each city i :
    FOR each city j :
        dist_matrix[i][j] ← EuclideanDistance(cities[i], cities[j])

Initialize ACO parameters :
    num_ants ← 20
    num_iterations ← 100
    alpha ← 1.0        // Importance of pheromone
    beta ← 5.0         // Importance of heuristic
    rho ← 0.5          // Evaporation rate
    Q ← 100            // Pheromone deposit constant

    pheromone_matrix ← matrix of ones (size num_cities x
                                       num_cities)

Compute heuristic_matrix as 1 / distance (with diagonal set
                                           to infinity)
```

```
Initialize :
    best_tour ← None
    best_distance ← infinity

FOR iteration = 1 to num_iterations :
    all_tours ← empty list
    all_distances ← empty list

    FOR each ant in num_ants :
        unvisited_cities ← list of all city indices
        start_city ← randomly select from unvisited_cities
        tour ← [ start_city]
        Remove start_city from unvisited_cities .

        current_city ← start_city

        WHILE unvisited_cities is not empty :
            FOR each unvisited city :
                Calculate probability using :
                    ( pheromone [current] [city]) ^ alpha * ( heuristic
                      [current] [city]) ^ beta
            Normalize probabilities
            Select next_city based on probabilities
            Add next_city to tour
            Remove next_city from unvisited
            current_city ← next_city

        Append start_city to tour to complete cycle

        total_distance ← sum of distances between consecutive
                         cities in tour
        Add tour to all_tours
        Add total_distance to all_distances

        IF total_distances < best_distance :
            best_distance ← total_distance
            best_tour ← tour
```

```
                                                            (9)
// Evaporate pheromone
Multiply each pheromone value by (1 - rho)
// update pheromones
FOR each tour and its distance :
    FOR each edge in tour :
        pheromone [ from_city] [ to_city] += Q / distance
        pheromone [ to_city] [ from_city] += Q / distance

  PRINT "Iteration i - Best distance so far : best_distance"

PRINT "Best Tour and Best Distance"
PLOT the best tour path :
    Extract x and y coordinates for cities in best_tour
    Plot cities and lines between them

END
```

output

```
Enter the number of cities : 5
Enter city coordinates in format : x y
city 0 : 0 0
city 1 : 1 3
city 2 : 4 3
city 3 : 6 1
city 4 : 3 0

Best Tour : [ 0, 4, 3, 2, 1, 0]
Best Distance : 15.15298244508295
```

Sneha SB
8/9/25

Code:

```
import numpy as np
import random
import math

# --------------------------------------------
# Step 1: Define the problem (set of cities with coordinates)
# --------------------------------------------
cities = [
    (0, 0),
    (1, 5),
    (5, 2),
    (6, 6),
```

```
    (8, 3),
    (7, 9),
    (2, 7),
    (3, 3)
]
N = len(cities)

# ---------------------------------------------
# Calculate Euclidean distance matrix
# ---------------------------------------------
def euclidean_distance(a, b):
    return math.sqrt((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2)

distance_matrix = [
    [euclidean_distance(cities[i], cities[j]) for j in range(N)]
    for i in range(N)
]

# ---------------------------------------------
# Step 2: Initialize Parameters
# ---------------------------------------------
num_ants = N
max_iterations = 100
alpha = 1.0   # influence of pheromone
beta = 5.0    # influence of heuristic (1/distance)
rho = 0.5     # evaporation rate
Q = 100.0     # pheromone deposit factor
tau_0 = 1.0   # initial pheromone level

pheromone = [[tau_0 for _ in range(N)] for _ in range(N)]
heuristic = [
    [1 / distance_matrix[i][j] if i != j else 0 for j in range(N)]
    for i in range(N)
]

best_tour = None
best_length = float('inf')

# ---------------------------------------------
# Step 3-5: Main Loop
# ---------------------------------------------
for iteration in range(max_iterations):
    all_tours = []
    all_lengths = []

    for ant in range(num_ants):
```

```
        unvisited = list(range(N))
        current_city = random.choice(unvisited)
        tour = [current_city]
        unvisited.remove(current_city)

        # Construct a tour
        while unvisited:
            probabilities = []
            for j in unvisited:
                tau = pheromone[current_city][j] ** alpha
                eta = heuristic[current_city][j] ** beta
                probabilities.append(tau * eta)
            total = sum(probabilities)
            probabilities = [p / total for p in probabilities]

            next_city = random.choices(unvisited, weights=probabilities, k=1)[0]
            tour.append(next_city)
            unvisited.remove(next_city)
            current_city = next_city

        # Return to starting city
        tour.append(tour[0])
        tour_length = sum(distance_matrix[tour[i]][tour[i + 1]] for i in range(N))
        all_tours.append(tour)
        all_lengths.append(tour_length)

        # Update best tour
        if tour_length < best_length:
            best_length = tour_length
            best_tour = tour

    # ---------------------------------------------
    # Step 4: Update Pheromones
    # ---------------------------------------------
    # Evaporation
    for i in range(N):
        for j in range(N):
            pheromone[i][j] *= (1 - rho)

    # Deposit pheromone
    for tour, length in zip(all_tours, all_lengths):
        for i in range(N):
            a = tour[i]
            b = tour[i + 1]
            pheromone[a][b] += Q / length
            pheromone[b][a] += Q / length  # symmetric TSP
```

```
# ----------------------------------------------
# Step 6: Output the Best Solution
# ----------------------------------------------
print("Best tour:", best_tour)
print("Best tour length:", round(best_length, 2))
```

Program 5

Solve the 0/1 Knapsack Problem by using **Cuckoo Search** to maximize total item value without exceeding the weight limit.

Algorithm:

Pseudocode for Cuckoo Search Algorithm

```
# Define the objective function to minimize
FUNCTION objective_function(x):
    RETURN sum of squares of elements in x

# Define the Levy flight distribution
FUNCTION levy_flight(Lambda = 1.5, size = 1):
    COMPUTE sigma using Gamma distribution and Lambda
    GENERATE Levy flight step using random numbers
    RETURN step

# Initialize the algorithm parameters
SET N = 50                 # Number of nests (solutions)
SET dimension = 2          # Number of dimensions for each solution (problem size)
SET lower_bound = -10      # lower bound for each dimension of solution
SET upper_bound = 10       # upper bound for each dimension of solution
SET T = 100                # Number of iterations
SET p = 0.25               # Probability of discovery (replacing worst nests)

# Initialize the population of nests randomly
CREATE nests as a matrix of size N x dimensions with
random values between lower_bound and upper_bound.

# Main loop to perform the search
FOR each iteration t from 1 to T:
    # step 1: Evaluate the fitness of each nest
    COMPUTE fitness for each nest using objective_function

    # step 2: Identify the best nest (solution)
    FIND the index of the nest with the best (lowest) fitness.

    # step 3: Generate new solutions using Levy flights
    GENERATE Levy flight steps for N nests
    UPDATE the nests by adding the Levy flight steps
    to current positions

    # step 4: Evaluate the fitness of the new nests
    COMPUTE the fitness of the new nests using
    objective_function

    # step 5: Replace the worst-performing nests with new
    random nests
    IDENTIFY the worst N * p nests (based on fitness)
    REPLACE the worst nests with new random positions
    (within the bounds)

    # step 6: Update the fitness of the current population
    of nests
    COMPUTE the fitness of the current population of nests

    # step 7: Track the best solution found so far
    FIND the index of the nest with the best fitness.

# output the best solution found after all iterations
OUTPUT the best nest and its corresponding fitness.
```

output

Best Solution: [-0.17065946 0.18250002]

with fitness: 0.06243090823291184

Application:
1. Neural Network Training
2. Scheduling
3. Travelling Salesman Problem
4. Knapsack problem

22/9/25

Code:

```
import random
import numpy as np

# ---------------------------------------------------------
# Step 1: Define the Knapsack Problem
# ---------------------------------------------------------
# Sample data: values and weights of items
values = [60, 100, 120, 80, 30]
weights = [10, 20, 30, 15, 5]
max_weight = 50
num_items = len(values)

# Fitness function: maximize total value while staying within capacity
def fitness(nest):
    total_value = 0
    total_weight = 0
    for i in range(num_items):
        if nest[i] == 1:
            total_value += values[i]
            total_weight += weights[i]
    if total_weight > max_weight:
        return 0  # Penalty for exceeding capacity
    return total_value


# ---------------------------------------------------------
# Step 2: Initialize Parameters
# ---------------------------------------------------------
num_nests = 25
max_iterations = 100
discovery_rate = 0.25  # Probability of discovering a nest (abandon rate)
alpha = 1.0  # Step size for Lévy flights (conceptual, here used for flipping)

# ---------------------------------------------------------
# Step 3: Initialize Population
# ---------------------------------------------------------
def initialize_population():
    return [[random.randint(0, 1) for _ in range(num_items)] for _ in
range(num_nests)]

population = initialize_population()

# ---------------------------------------------------------
```

```python
# Step 4: Evaluate Fitness
# -------------------------------------------------------
fitness_values = [fitness(nest) for nest in population]

# -------------------------------------------------------
# Step 5: Generate New Solutions using Lévy flights
# -------------------------------------------------------
# In this binary knapsack version, we simulate Lévy flight as bit flipping
def levy_flight(nest):
    new_nest = nest.copy()
    for i in range(num_items):
        if random.random() < 0.3:  # 30% chance to flip each bit
            new_nest[i] = 1 - new_nest[i]
    return new_nest

# -------------------------------------------------------
# Step 6: Replace Worst Nests
# -------------------------------------------------------
def abandon_worst_nests(population, fitness_values):
    num_abandon = int(discovery_rate * num_nests)
    worst_indices = np.argsort(fitness_values)[:num_abandon]

    for idx in worst_indices:
        population[idx] = [random.randint(0, 1) for _ in range(num_items)]
    return population


# -------------------------------------------------------
# Step 7: Main Loop
# -------------------------------------------------------
best_nest = None
best_fitness = -1

for iteration in range(max_iterations):
    for i in range(num_nests):
        # Generate a new solution by Lévy flight
        new_nest = levy_flight(population[i])
        new_fitness = fitness(new_nest)

        # Greedy selection
        if new_fitness > fitness_values[i]:
            population[i] = new_nest
            fitness_values[i] = new_fitness

        # Track the global best
        if new_fitness > best_fitness:
```

```
            best_fitness = new_fitness
            best_nest = new_nest.copy()

    # Abandon a fraction of the worst nests
    population = abandon_worst_nests(population, fitness_values)
    fitness_values = [fitness(nest) for nest in population]

# ------------------------------------------------------
# Step 8: Output the Best Solution
# ------------------------------------------------------
print("Best Solution (Item Inclusion):", best_nest)
print("Best Total Value:", best_fitness)
print("Total Weight:", sum(weights[i] for i in range(num_items) if best_nest[i] ==
1))
```

Program 6

Optimize the assignment of cars to parking slots to minimize total walking distance using **Grey Wolf Optimization**, simulating the social hierarchy and hunting behavior of grey wolves to iteratively improve solutions.

Algorithm:

Grey Wolf Optimizer on Financial Market Forecasting

Function fitness_function (params, stock_data):
    short_window = params[0]
    long_window = params[1]

\# Calculate moving averages
short_ma = CalculateMovingAverage (stock_data, short_window)
long_ma = CalculateMovingAverage (stock_data, long_window)

\# Generate Buy/Sell signals
signals = GenerateSignals (short_ma, long_ma)

\# calculate daily returns
daily_returns = CalculateDailyReturns (stock_data)

\# calculate strategy returns
strategy_returns = CalculateStrategyReturns (daily_returns, signal)

\# calculate total profit
total_profit = Sum (strategy_returns)

\# Return negative profit for minimization
return -total_profit

Class GreyWolfOptimizer:
    Initialize:
        Set num_wolves, max_iter, fitness_function, stock_data
        Initialize wolves' positions randomly in range [5, 50]
        Evaluate fitness of each wolf

    Function update_position (alpha_pos, wolf_pos, A, C):
        Calculate distance (D) between alpha_pos and wolf_pos
        Update wolf's position based on A and C coefficients

    Function optimize:
        For each iteration (t in max_iter):
            Sort wolves based on fitness
            Set alpha, beta, delta wolves based on sorted fitness
            Generate random A and C for each wolf
            For each wolf:
                If random() < 0.5:
                    Update position using alpha position
                Else:
                    Update position using beta position
                Clamp position to valid range [5, 50]
            Recalculate fitness for all wolves
            Print best fitness in current iteration
        Return best wolf's position (alpha) and best profit

Main:
    Fetch stock_data for a specific symbol (e.g. 'AAPL')
    Create a GWO instance and pass the fitness_function and stock_data
    Run the optimization to get the best parameters and profit
    Print the best moving average windows and profit
    Plot stock price and optimized moving averages on a graph.

\# output
Iteration 1/25, Best Profit: 0.0000

Iteration 25/25, Best Profit: 0.0000
Best Parameters: Short window: 13.371446253377613
                 Long window: 7.341203132992351

Best Profit: 0.0000

Sneha SB 29/9/25

Code:

```
import numpy as np
import random

# ------------------------------------------------
# Problem Definition: Parking Slot Allocation
# ------------------------------------------------
num_cars = 5
num_slots = 5
slot_distances = np.array([10, 20, 15, 30, 25])  # Distances of parking slots from entrance

num_wolves = 10
```

```python
max_iter = 50

# ---------------------------------------------
# Initialize Wolves (Population)
# ---------------------------------------------
def initialize_population(num_wolves, num_slots):
    population = []
    for _ in range(num_wolves):
        wolf = np.random.permutation(num_slots)
        population.append(wolf)
    return population

# ---------------------------------------------
# Fitness Function: Total Walking Distance
# ---------------------------------------------
def fitness(wolf):
    total_distance = 0
    for car_index, slot_index in enumerate(wolf):
        total_distance += slot_distances[slot_index]
    return total_distance

# ---------------------------------------------
# Update Position (GWO-inspired for permutation problems)
# ---------------------------------------------
def update_position(wolf, alpha, beta, delta, a):
    new_wolf = wolf.copy()

    for i in range(len(wolf)):
        r1 = np.random.rand()
        r2 = np.random.rand()

        # Move the wolf toward alpha, beta, or delta
        if r1 < 0.33:
            new_wolf[i] = alpha[i]
        elif r1 < 0.66:
            new_wolf[i] = beta[i]
        else:
            new_wolf[i] = delta[i]

    # Ensure new_wolf is a valid permutation (no duplicates)
    new_wolf = np.unique(new_wolf)
    if len(new_wolf) < len(wolf):
        missing_values = list(set(range(len(wolf))) - set(new_wolf))
        np.random.shuffle(missing_values)
        new_wolf = np.append(new_wolf, missing_values)
```

```python
    # Random swap for exploration
    swap_indices = np.random.choice(len(wolf), 2, replace=False)
    new_wolf[swap_indices[0]], new_wolf[swap_indices[1]] = (
        new_wolf[swap_indices[1]],
        new_wolf[swap_indices[0]],
    )

    return new_wolf

# ---------------------------------------------
# Main Grey Wolf Optimizer (GWO)
# ---------------------------------------------
def gwo_parking_allocation():
    # Initialize wolves (population)
    population = initialize_population(num_wolves, num_slots)

    # Initialize alpha, beta, delta wolves
    alpha = beta = delta = None
    alpha_score = beta_score = delta_score = float("inf")

    # Main optimization loop
    for iteration in range(max_iter):
        # Evaluate fitness of each wolf
        fitness_scores = []
        for wolf in population:
            score = fitness(wolf)
            fitness_scores.append(score)

            # Update alpha, beta, delta
            if score < alpha_score:
                delta_score, delta = beta_score, beta
                beta_score, beta = alpha_score, alpha
                alpha_score, alpha = score, wolf.copy()
            elif score < beta_score:
                delta_score, delta = beta_score, beta
                beta_score, beta = score, wolf.copy()
            elif score < delta_score:
                delta_score, delta = score, wolf.copy()

        # Update positions of wolves
        new_population = []
        for wolf in population:
            a = 2 - iteration * (2 / max_iter)  # exploration parameter
            new_wolf = update_position(wolf, alpha, beta, delta, a)
            new_population.append(new_wolf)
```

```
        population = new_population

        # Display progress
        print(f"Iteration {iteration + 1}: Best total walking distance = 
{alpha_score}")

    # Final best result
    print("\nBest Assignment of Cars to Slots (car i → slot number):")
    print(alpha)
    print("Slot distances:", slot_distances[alpha])
    print("Total walking distance:", alpha_score)

# ---------------------------------------------
# Run the Optimizer
# ---------------------------------------------
if __name__ == "__main__":
    gwo_parking_allocation()
```

Program 7

Optimize the search efficiency of multiple drones exploring a grid by coordinating their positions based on neighbors' success using a **Parallel Cellular Algorithm** to maximize the total searched area.

Algorithm:

PCA for Feature Selection

Load dataset
Split dataset into training and validation sets

Define:
    GRID_ROWS, GRID_COLS   // Dimensions of 2D grid
    NUM_FEATURES   // Number of features in the dataset
    ITERATIONS   // Number of iterations to run

Initialize:
    population [GRID_ROWS][GRID_COLS] ← Random binary masks
                                        (length = NUM_FEATURES)
    best_solution ← None
    best_fitness ← ∞

Define function FITNESS (mask):
    If mask has no selected features:
        Return 1.0   # maximum error
    Train classifier (e.g, logistic regression) on selected features
    Evaluate classifier on validation set
    Return classification error (1 − accuracy)

Define function GET_NEIGHBORS (i, j):
    Return list of neighbor coordinates (Moore neighborhood with wrap-around)

For iteration from 1 to ITERATIONS:
    fitness_grid ← Evaluate FITNESS for each cell in population
    Find cell with minimum fitness in fitness_grid:
        If this fitness < best_fitness:
            best_fitness ← this fitness
            best_solution ← corresponding feature mask
    new_population ← copy of current population

    For each cell (i, j) in grid:
        neighbor ← GET_NEIGHBORS (i, j)
        best_neighbor ← population [i][j]
        best_neighbor_fitness ← fitness_grid [i][j]

        For each neighbor (ni, nj) in neighbors:
            If fitness_grid [ni][nj] < best_neighbor_fitness:
                best_neighbor ← population [ni][nj]
                best_neighbor_fitness ← fitness_grid [ni][nj]

        If best_neighbor_fitness < fitness_grid [i][j]
            new_population [i][j] ← best_neighbor

    population ← new_population

Output:

Iteration 1/15, Best Fitness (error): 0.0175
Iteration 2/15, Best Fitness (error): 0.0175
⋮
Iteration 15/15, Best Fitness (error): 0.0175

Best feature subset found (binary mask):
[0 0 1 1 0 0 0 0 0 1 1 1 0 1 0 0 1 1 0 1 0 0 1 1 0 0 1 1 0 0]
Number of selected features : 13
Classification accuracy : 0.9825

13/10/25

Code:

```
import random
import numpy as np

# ---------------------------------------------
# PARAMETERS
# ---------------------------------------------
GRID_SIZE = 5        # 5x5 search area
NUM_DRONES = 10      # Number of drones
NUM_ITERATIONS = 20 # Number of time cycles

# ---------------------------------------------
# DRONE CLASS
# ---------------------------------------------
```

```python
class Drone:
    def __init__(self, x, y):
        self.x = x  # X-coordinate of the drone
        self.y = y  # Y-coordinate of the drone
        self.fitness = 0  # Represents search success (number of finds)

    def search_area(self):
        """Simulate the drone searching its area (20% chance of finding
something)."""
        if random.random() < 0.2:
            self.fitness += 1
        return self.fitness

    def update_position(self, new_x, new_y):
        """Move the drone to a new position within grid bounds."""
        self.x = max(0, min(GRID_SIZE - 1, new_x))
        self.y = max(0, min(GRID_SIZE - 1, new_y))

# ---------------------------------------------
# INITIALIZATION
# ---------------------------------------------
def initialize_drones():
    """Initialize drones at random grid positions."""
    drones = []
    for _ in range(NUM_DRONES):
        x = random.randint(0, GRID_SIZE - 1)
        y = random.randint(0, GRID_SIZE - 1)
        drones.append(Drone(x, y))
    return drones

# ---------------------------------------------
# FITNESS EVALUATION
# ---------------------------------------------
def evaluate_drones(drones):
    """Evaluate the total fitness (search efficiency) of all drones."""
    total_fitness = 0
    for drone in drones:
        total_fitness += drone.search_area()
    return total_fitness

# ---------------------------------------------
# NEIGHBOR DETECTION
# ---------------------------------------------
def get_neighbors(drone, drones):
    """Find neighboring drones within 1 cell in any direction."""
    neighbors = []
```

```python
    for other in drones:
        if other != drone:
            if abs(drone.x - other.x) <= 1 and abs(drone.y - other.y) <= 1:
                neighbors.append(other)
    return neighbors

# ---------------------------------------------
# POSITION UPDATE STRATEGY
# ---------------------------------------------
def update_drone_position(drone, neighbors):
    """Move drone toward the best neighbor (highest fitness)."""
    if neighbors:
        best_neighbor = max(neighbors, key=lambda n: n.fitness)
        if best_neighbor.x != drone.x or best_neighbor.y != drone.y:
            # Move one step toward best neighbor
            dx = np.sign(best_neighbor.x - drone.x)
            dy = np.sign(best_neighbor.y - drone.y)
            drone.update_position(drone.x + dx, drone.y + dy)

# ---------------------------------------------
# MAIN SEARCH AND RESCUE LOOP
# ---------------------------------------------
def search_and_rescue():
    drones = initialize_drones()
    best_fitness = 0

    for iteration in range(NUM_ITERATIONS):
        total_fitness = evaluate_drones(drones)

        if total_fitness > best_fitness:
            best_fitness = total_fitness
            print(f"Iteration {iteration + 1}: Best Fitness = {best_fitness}")

        # Update drone positions based on neighbors
        for drone in drones:
            neighbors = get_neighbors(drone, drones)
            update_drone_position(drone, neighbors)

    return best_fitness

# ---------------------------------------------
# RUN THE SIMULATION
# ---------------------------------------------
if __name__ == "__main__":
    best_fitness = search_and_rescue()
    print(f"\nBest Fitness Achieved: {best_fitness}")
```