VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Vismitha Raj S Doshi (1WA23CS047)

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

CERTIFICATE

This is to certify that the Lab work entitled “Bio Inspired Systems (23CS5BSBIS)” carried out by **Vismitha Raj S Doshi (1WA23CS047),** 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.

Sandhya A Kulkarni Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

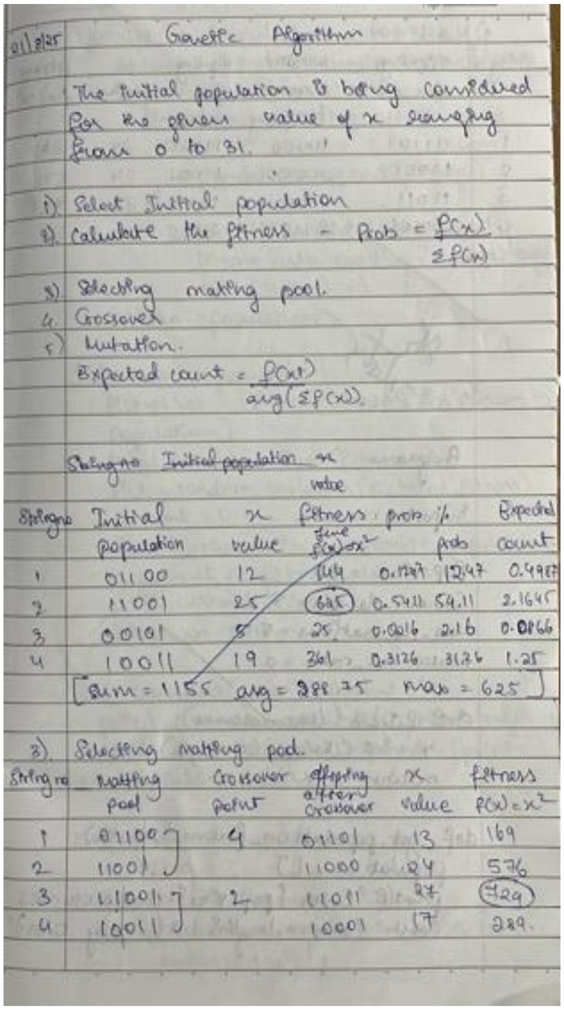
Github Link:
https://github.com/vismitharaj/BIS-Lab-

Program 1 : Genetic Algorithm

Problem statement:

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems.

Algorithm:



01/8/25 Genetic Algorithm

The initial population is being considered for the given value of x ranging from 0 to 31.

1) Select Initial population
2) Calculate the fitness - $Prob = \frac{f(x)}{\Sigma f(x)}$
3) Selecting mating pool.
4. Crossover
5) Mutation.

$Expected\ count = \frac{f(x)}{avg(\Sigma f(x))}$

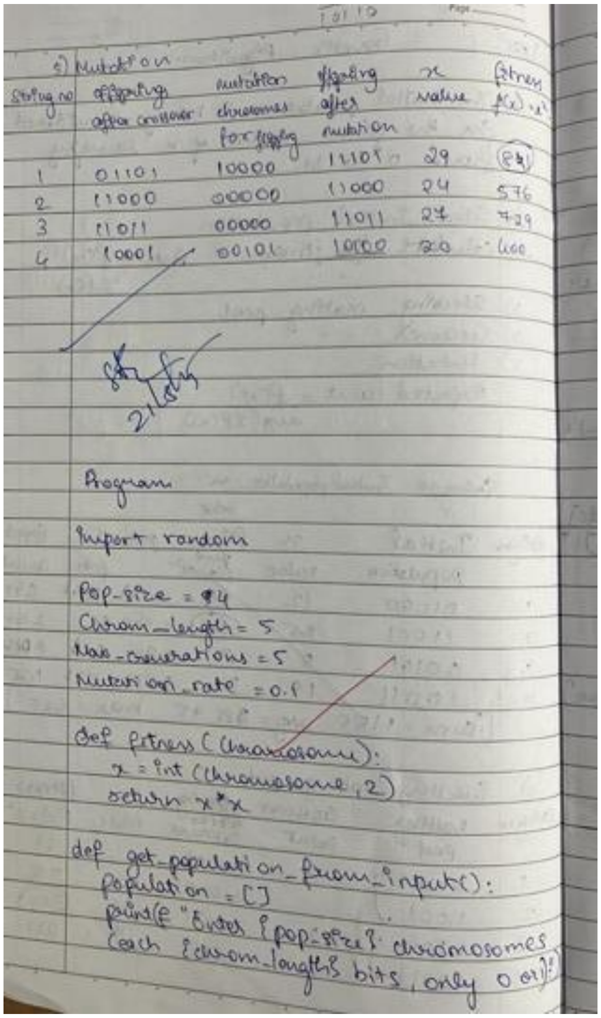
String no Initial population x value

String no	Initial population	x value	fitness func $f(x)=x^2$	prob	% prob	Expected count
1	01100	12	144	0.1247	12.47	0.4987
2	11001	25	625	0.5411	54.11	2.1645
3	00101	5	25	0.0216	2.16	0.0866
4	10011	19	361	0.3126	31.26	1.25

[sum = 1155 avg = 288.75 max = 625]

3) Selecting mating pool.

String no	Mating pool	Crossover point	offspring after crossover	x value	fitness $f(x)=x^2$
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289.



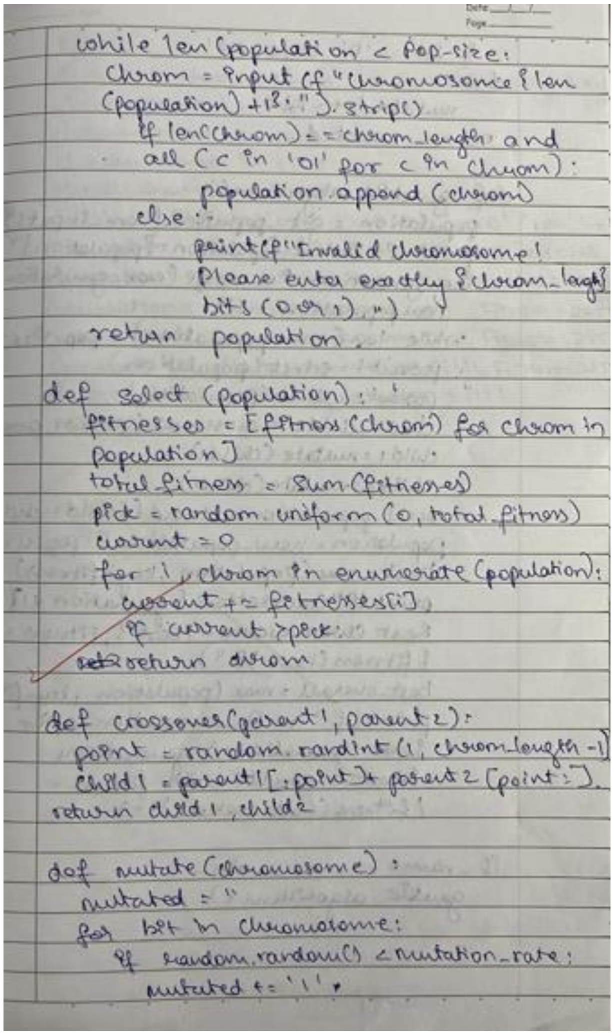
5) Mutation

String no	offspring after crossover	mutation chromosomes for flipping	offspring after mutation	x value	fitness $f(x)=x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400

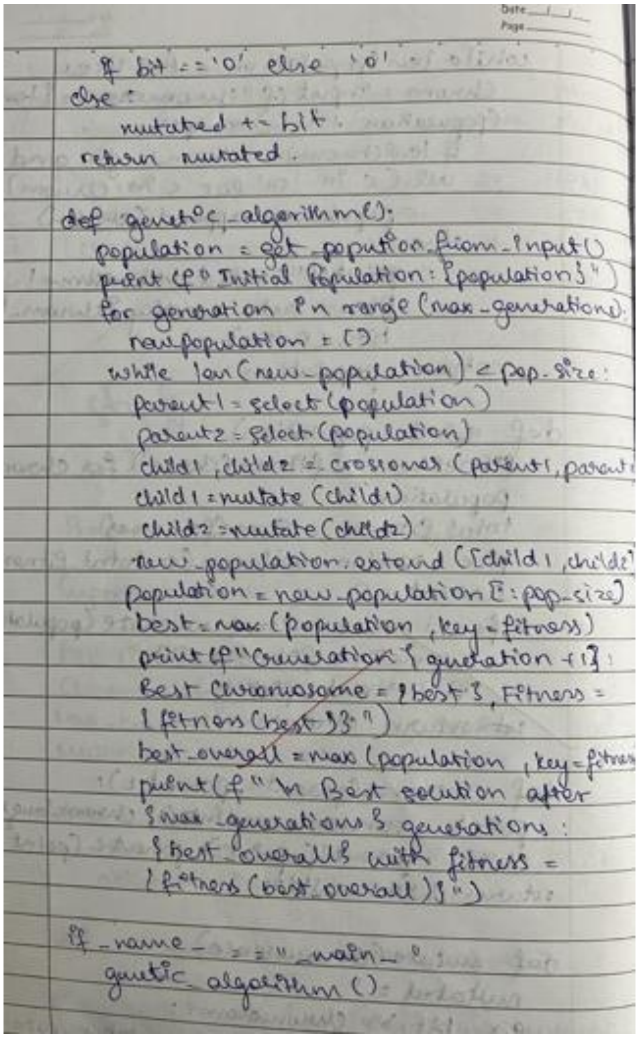
21/8

Program

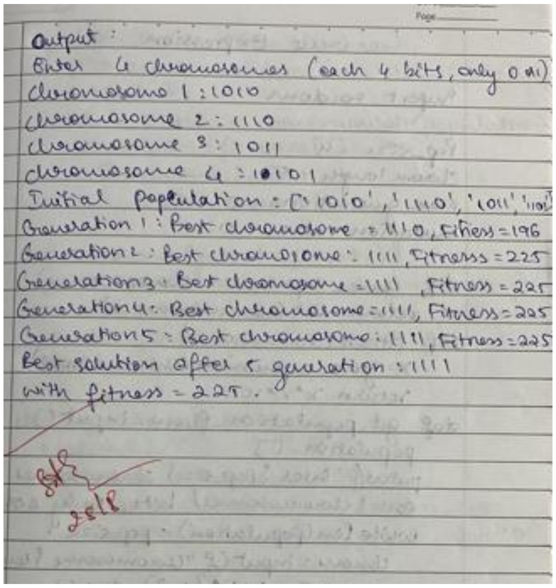
```
import random

pop-size = 4
Chrom_length = 5
Max_Generations = 5
Mutation_rate = 0.1

def fitness (chromosome):
    x = int (chromosome, 2)
    return x*x

def get_population_from_input():
    population = []
    print(f "Enter {pop_size} chromosomes
    (each {chrom_length} bits, only 0 or 1)!
```



```
while len(population < pop-size:
    chrom = input (f"chromosome {len
    (population) +1}: "). strip()
        if len(chrom) = = chrom_length and
        all (c in '01' for c in chrom):
            population append (chrom)
        else:
            print(f"Invalid chromosome!
            Please enter exactly {chrom_length}
            bits (0 or 1). ")
    return population.

def select (population):
    fitnesses = [fitness (chrom) for chrom in
    population]
    total_fitness = sum (fitnesses)
    pick = random. uniform (0, total_fitness)
    current = 0
    for i, chrom in enumerate (population):
        current += fitnesses[i]
        if current >pick:
    ~~set~~ return chrom

def crossover(parent1, parent2):
    point = random. randint (1, chrom_length -1)
    child1 = parent1[:point]+ parent2 [point:].
return child1, child2

def mutate (chromosome) :
    mutated = "
    for bit in chromosome:
        if random.random() < mutation_rate:
            mutated += '1'
```



```
        if bit = = '0' else '0'
    else :
        mutated += bit.
    return mutated.

def genetic_algorithm():
    population = get_popution_from_input()
    print (f" Initial Population: {population}")
    for generation in range (max_generations):
        newpopulation = []
        while len (new_population) < pop_size:
            parent1 = select (population)
            parent2 = select (population)
            child1, child2 = crossover (parent1, parent
            child1 = mutate (child1)
            child2 = mutate (child2)
            new_population. extend ([child1, child2
        population = new_population [:pop_size]
        best = max (population, key = fitness)
        print (f"Generation {generation +1}:
        Best Chromosome = {best}, Fitness =
        {fitness (best)}")
        best_overall = max (population, key=fitnes
        print (f" \n Best solution after
        {max_generations} generations:
        {best_overall} with fitness =
        {fitness (best_overall)}")

if _name_ = = "_main_":
    genetic_algorithm ()
```

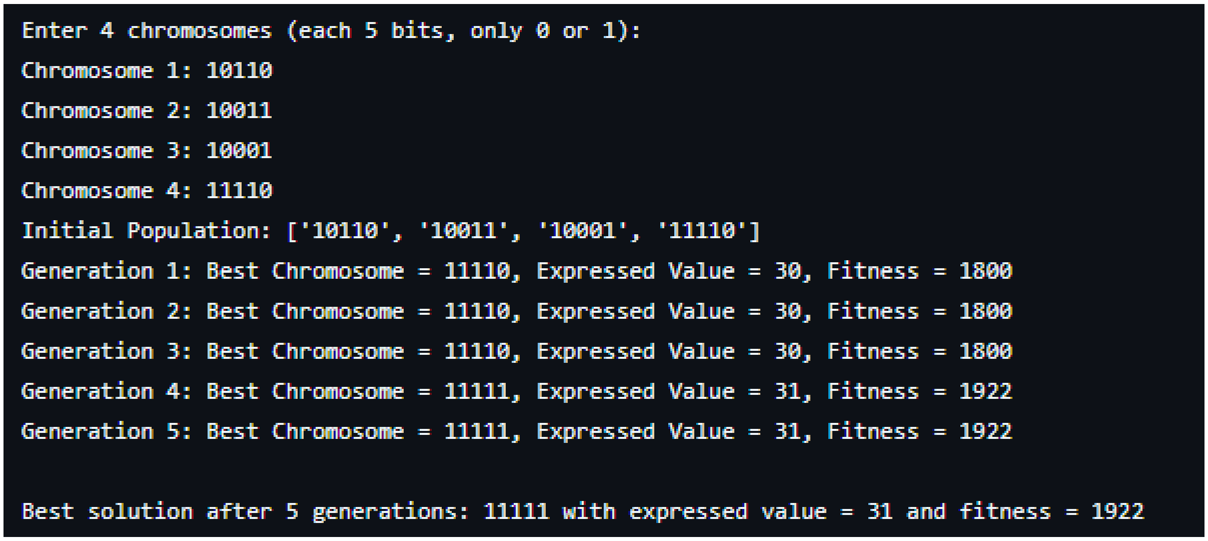


```
Output :
Enter 4 chromosomes (each 4 bits, only 0 or 1)
Chromosome 1 : 1010
Chromosome 2 : 1110
Chromosome 3 : 1011
Chromosome 4 : 10101
Initial population : ['1010', '1110', '1011', '1101']
Generation 1 : Best chromosome = 1110, Fitness = 196
Generation 2 : Best chromosome : 1111, Fitness = 225
Generation 3 : Best chromosome = 1111 , Fitness = 225
Generation 4 : Best chromosome = 1111, Fitness = 225
Generation 5 : Best chromosome : 1111, Fitness = 225
Best solution after 5 generation : 1111
with fitness = 225.
```

28/8

Code:

```
import random
def fitness(x):
    return x**2
def int_to_bin(x):
    return format(x, '05b')
def bin_to_int(b):
    return int(b, 2)
def tournament_selection(pop, k=3):
    selected = random.sample(pop, k)
    selected.sort(key=lambda x: fitness(x), reverse=True)
    return selected[0]
def crossover(p1, p2):
    b1, b2 = int_to_bin(p1), int_to_bin(p2)
    point = random.randint(1, 4)
    child1 = bin_to_int(b1[:point] + b2[point:])
    child2 = bin_to_int(b2[:point] + b1[point:])
    return child1, child2
def mutate(x, mutation_rate=0.1):
    if random.random() < mutation_rate:
        b = list(int_to_bin(x))
        pos = random.randint(0, 4)
        b[pos] = '1' if b[pos] == '0' else '0'
        return bin_to_int("".join(b))
    return x
def genetic_algorithm(initial_population=None, pop_size=6, generations=20,
crossover_rate=0.8,mutation_rate=0.1):
    if initial_population:
        population = initial_population[:pop_size]  # take only needed size
    else:
        population = [random.randint(0, 31) for _ in range(pop_size)]
    for gen in range(generations):
        population.sort(key=lambda x: fitness(x), reverse=True)
        best = population[0]
        print(f"Gen {gen}: Best x={best}, f(x)={fitness(best)}")
        new_pop = [best]
        while len(new_pop) < pop_size:
            parent1 = tournament_selection(population)
            parent2 = tournament_selection(population)
            if random.random() < crossover_rate:
                child1, child2 = crossover(parent1, parent2)
            else:
                child1, child2 = parent1, parent2
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)
            new_pop.extend([child1, child2])
        population = new_pop[:pop_size]
    population.sort(key=lambda x: fitness(x), reverse=True)
    best = population[0]
    print(f"\nBest Solution: x={best}, f(x)={fitness(best)}")
custom_population = [3, 7, 15, 20, 25, 30]
genetic_algorithm(initial_population=custom_population, generations=5)
```



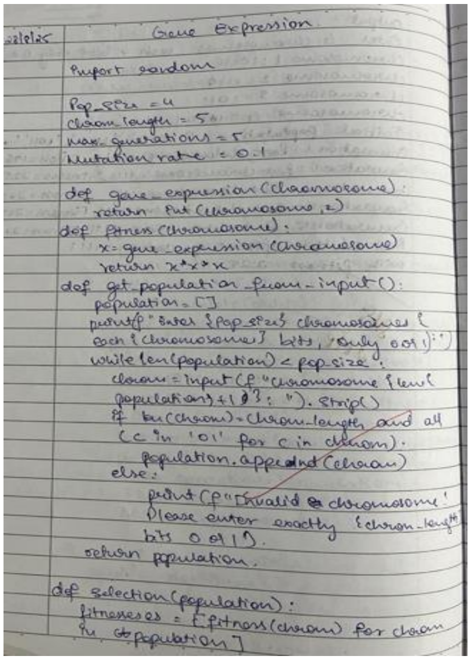
```
Enter 4 chromosomes (each 5 bits, only 0 or 1):
Chromosome 1: 10110
Chromosome 2: 10011
Chromosome 3: 10001
Chromosome 4: 11110
Initial Population: ['10110', '10011', '10001', '11110']
Generation 1: Best Chromosome = 11110, Expressed Value = 30, Fitness = 1800
Generation 2: Best Chromosome = 11110, Expressed Value = 30, Fitness = 1800
Generation 3: Best Chromosome = 11110, Expressed Value = 30, Fitness = 1800
Generation 4: Best Chromosome = 11111, Expressed Value = 31, Fitness = 1922
Generation 5: Best Chromosome = 11111, Expressed Value = 31, Fitness = 1922

Best solution after 5 generations: 11111 with expressed value = 31 and fitness = 1922
```

<u>Program 2 : Optimization via Gene expression</u>

Problem statement:

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

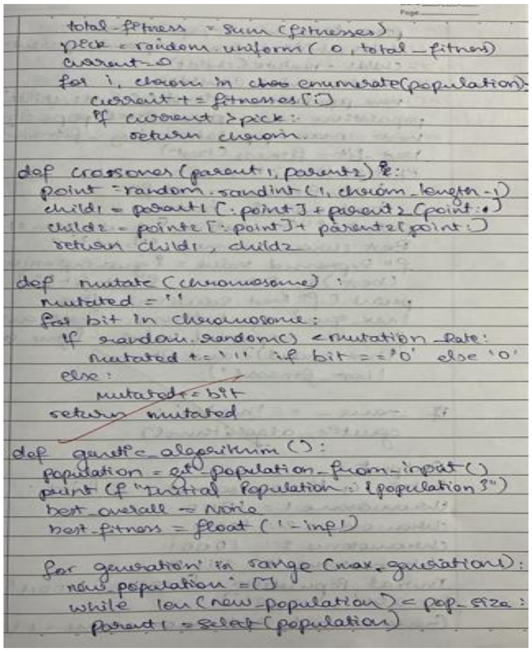
Algorithm:



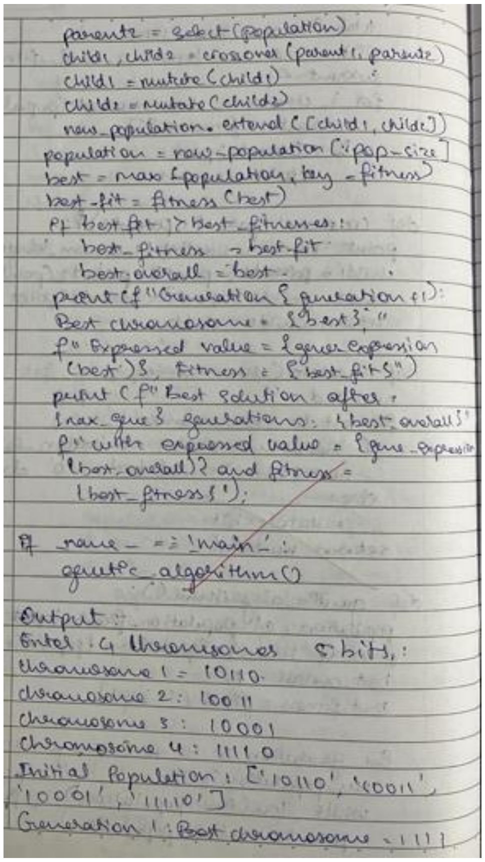
22|9|25

Gene Expression

```
import random

Pop_size = 4
chrom_length = 5
max_generations = 5
Mutation rate = 0.1

def gene_expression(chromosome):
    return int(chromosome, 2)
def fitness(chromosome):
    x = gene_expression(chromosome)
    return x*x*x
def get_population_from_input():
    population = []
    print(f"Enter {pop_size} chromosomes {
    each {chromosome} bits, only 0 or 1):")
    while len(population) < pop_size:
        chrom = input(f"chromosome {len(
        population)+1}: ").strip()
        if len(chrom) = chrom_length and all
        (c in '01' for c in chrom).
            population.append(chrom)
        else:
            print(f"Invalid chromosome!
            Please enter exactly {chrom_length}
            bits 0 or 1}).
    return population.

def selection(population):
    fitnesses = [fitness(chrom) for chrom
    in population]
```



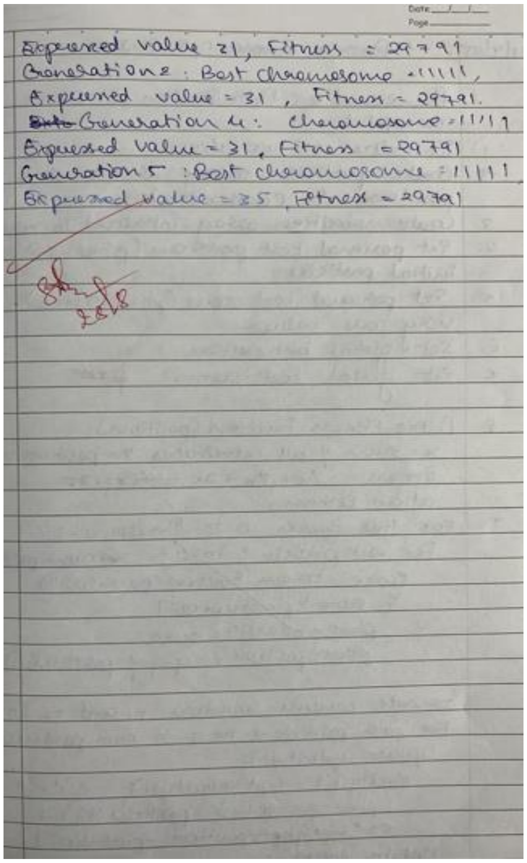
```
    total_fitness = sum(fitnesses)
    pick = random.uniform( 0, total_fitness)
    current = 0
    for i, chrom in enumerate(population):
        current += fitnesses[i]
        if current > pick:
            return chrom.

def crossover(parent1, parent2):
    point = random.randint(1, chrom_length-1)
    child1 = parent1[:point] + parent2(point:)
    child2 = pointe[:point]+ parent2[point:]
    return child1, child2

def mutate(chromosome):
    mutated = ''
    for bit in chromosome:
        if random.random() < mutation_Rate:
            mutated += '1' if bit == '0' else '0'
        else:
            mutated += bit
    return mutated

def genetic_algorithm():
    population = get_population_from_input()
    print(f"Initial Population : {population}")
    best_overall = None
    best_fitness = float('-inf')

    for generation in range(max_generations):
        new_population = []
        while len(new_population) < pop_size:
            parent1 = select(population)
```



```
            parent2 = select(population)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])
        population = new_population[:pop_size]
        best = max(population, key = fitness)
        best_fit = fitness(best)
        if best_fit > best_fitnesses:
            best_fitness = best_fit
            best_overall = best.
        print(f"Generation {generation+1}:
        Best chromosome = {best}, "
        f" Expressed value = {gene_expression
        (best)}, Fitness = {best_fit}")
    print(f" Best solution after
    {max_gene} generations: {best_overall}"
    f" with expressed value = {gene_expression
    (best_overall)} and fitness =
    {best_fitness}');

if _name_ == '_main_':
    genetic_algorithm()
```

Output:

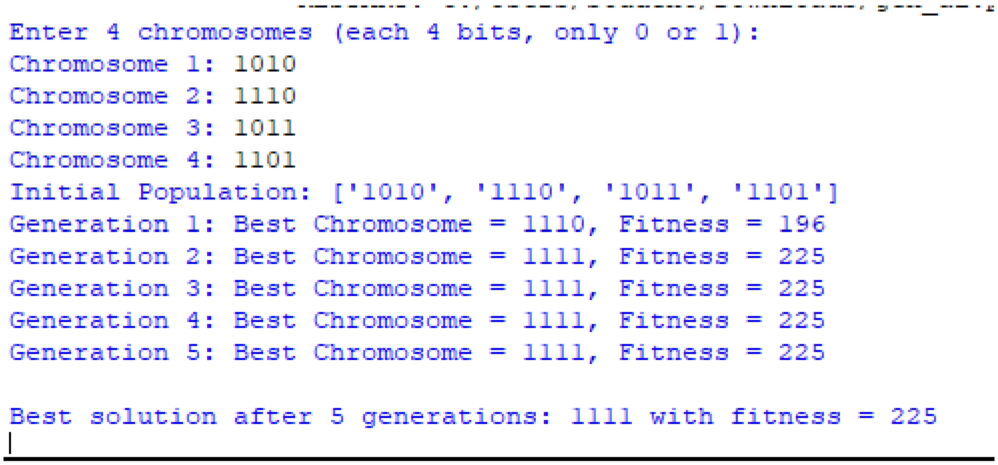
```
Enter 4 chromosomes 5 bits:
chromosome 1 = 10110.
chromosome 2: 10011
chromosome 3: 10001
chromosome 4: 1111 0
Initial Population: ['10110', '10011',
'10001', '11110']
Generation 1: Best chromosome = 1111
```



```
Expressed value 21, Fitness = 29791
Generation 2: Best chromosome = 11111,
Expressed value = 31, Fitness = 29791.
Generation 4: chromosome = 11111
Expressed value = 31, Fitness = 29791
Generation 5: Best chromosome = 11111
Expressed value = 35, Fitness = 29791
```

Code:

```
import random
import math
cities = [
    (0, 0), (1, 5), (5, 2), (6, 6), (8, 3),
    (2, 1), (7, 7), (3, 3), (4, 4), (9, 0)
]
def distance(a, b):
    return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)
def total_distance(tour):
    dist = 0
    for i in range(len(tour)):
        city_a = cities[tour[i]]
        city_b = cities[tour[(i+1) % len(tour)]]
        dist += distance(city_a, city_b)
    return dist
def create_individual(n):
    gene = list(range(n))
    random.shuffle(gene)
    return gene
def mutate(individual, rate=0.1):
    ind = individual[:]
    for i in range(len(ind)):
        if random.random() < rate:
            j = random.randint(0, len(ind)-1)
            ind[i], ind[j] = ind[j], ind[i]
    return ind
def crossover(parent1, parent2):
    size = len(parent1)
    a, b = sorted([random.randint(0, size-1) for _ in range(2)])
    child = [None]*size
    child[a:b+1] = parent1[a:b+1]
    p2_index = 0
    for i in range(size):
        if child[i] is None:
            while parent2[p2_index] in child:
                p2_index += 1
            child[i] = parent2[p2_index]
    return child
def genetic_algorithm(generations=100, pop_size=100, mutation_rate=0.1):
    num_cities = len(cities)
    population = [create_individual(num_cities) for _ in range(pop_size)]
    best = None
    best_dist = float('inf')
    for gen in range(generations):
        scored = [(ind, total_distance(ind)) for ind in population]
        scored.sort(key=lambda x: x[1])
        if scored[0][1] < best_dist:
            best = scored[0][0]
            best_dist = scored[0][1]
        new_pop = [best]
        while len(new_pop) < pop_size:
            p1 = random.choice(scored[:50])[0]
            p2 = random.choice(scored[:50])[0]
            child = crossover(p1, p2)
            child = mutate(child, mutation_rate)
            new_pop.append(child)
        population = new_pop
        if gen % 20 == 0:
            print(f"Gen {gen}: Best distance = {best_dist:.2f}")
    return best, best_dist
best_tour, best_dist = genetic_algorithm()
print("\nBest tour found:")
print(best_tour)
print(f"Total distance: {best_dist:.2f}")
```



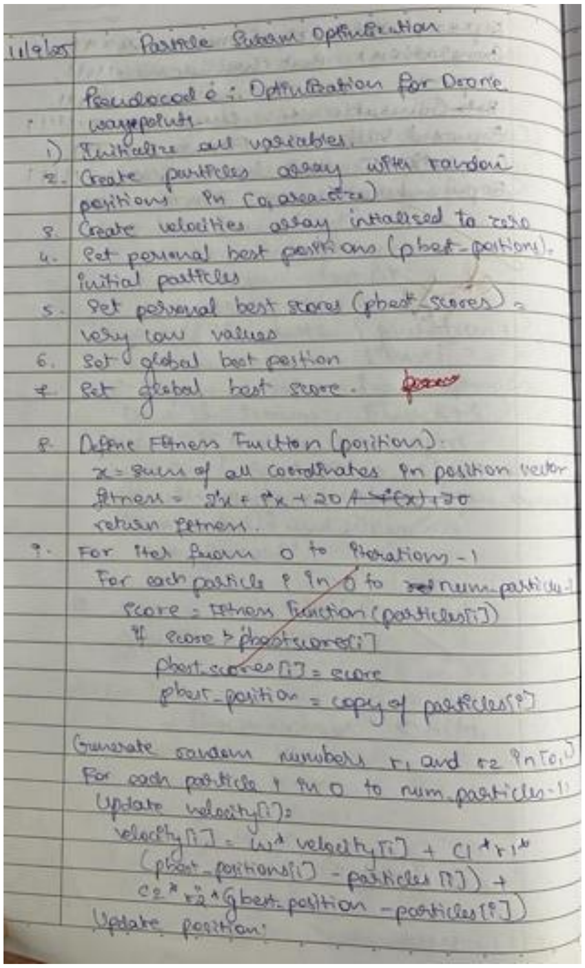
```
Enter 4 chromosomes (each 4 bits, only 0 or 1):
Chromosome 1: 1010
Chromosome 2: 1110
Chromosome 3: 1011
Chromosome 4: 1101
Initial Population: ['1010', '1110', '1011', '1101']
Generation 1: Best Chromosome = 1110, Fitness = 196
Generation 2: Best Chromosome = 1111, Fitness = 225
Generation 3: Best Chromosome = 1111, Fitness = 225
Generation 4: Best Chromosome = 1111, Fitness = 225
Generation 5: Best Chromosome = 1111, Fitness = 225

Best solution after 5 generations: 1111 with fitness = 225
```

Program 3 : Particle swarm Optimization

Problem statement:

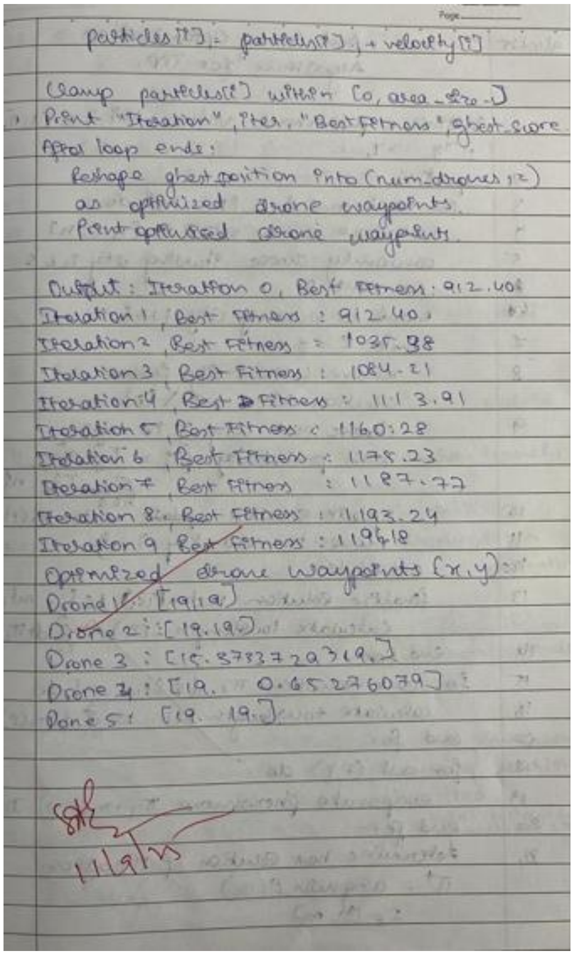
Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality.

Algorithm:



11/9/25 Particle Swarm Optimization

```
Pseudocode : Optimization for Drone waypoints
1) Initialize all variables.
2. Create particles array with random positions in [0, area_size)
3. Create velocities array initialised to zero
4. Set personal best positions (pbest_positions) = initial particles
5. Set personal best scores (pbest_scores) = very low values
6. Set global best position
7. Set global best score.

8. Define Fitness Function (position):
    x = sum of all coordinates in position vector
    fitness = 3*x + 9*x + 20 ~~+ f(x)+20~~
    return fitness.
9. For iter from 0 to iterations - 1
    For each particle i in 0 to ~~n~~ num_particles-1
        score = fitness function (particles[i])
        if score > pbest_scores[i]
            pbest_scores[i] = score
            pbest_position = copy of particles[i]

Generate random numbers r1 and r2 in [0,1]
For each particle i in 0 to num_particles-1
    Update velocity[i]:
        velocity[i] = w * velocity[i] + c1 * r1 *
            (pbest_positions[i] - particles[i]) +
            c2 * r2 * (gbest_position - particles[i])
    Update position:
```



```
        particles[i] = particles[i] + velocity[i]

    Clamp particles[i] within [0, area_size]
    Print "Iteration", iter, "Best Fitness", gbest_score
After loop ends:
    Reshape gbest_position into (num_drones, 2)
    as optimized drone waypoints
    Print optimized drone waypoints
```

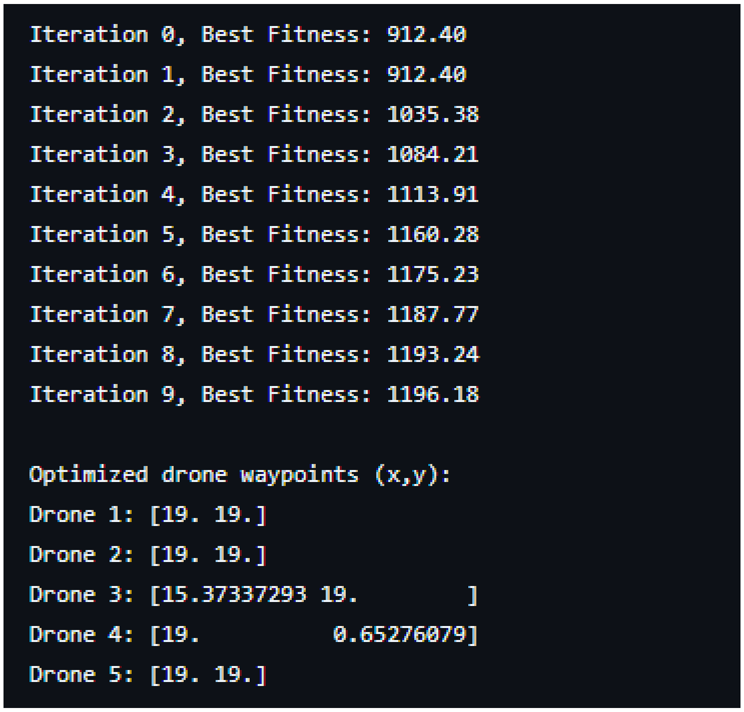
```
Output : Iteration 0, Best Fitness : 912.40
Iteration 1, Best Fitness : 912.40
Iteration 2, Best Fitness : 1035.38
Iteration 3, Best Fitness : 1084.21
Iteration 4, Best Fitness : 1113.91
Iteration 5, Best Fitness : 1160.28
Iteration 6, Best Fitness : 1178.23
Iteration 7, Best Fitness : 1187.77
Iteration 8, Best Fitness : 1193.24
Iteration 9, Best Fitness : 1196.18
Optimized drone waypoints (x, y):
Drone 1: [19, 19.]
Drone 2: [19. 19.]
Drone 3 : [15.57337293 19.]
Drone 4 : [19.    0.65276079]
Drone 5: [19.  19.]
```

11/9/25

Code:

```
import numpy as np
x_data = np.array([1, 2, 3, 4, 5])
y_data = np.array([3, 5, 7, 9, 11])
def objective_function(theta):
    theta_0, theta_1 = theta
    predictions = theta_0 + theta_1 * x_data
    errors = y_data - predictions
    return np.sum(errors**2)
num_particles = 30
num_iterations = 10
w = 0.7
```

```
c1 = 1.5
c2 = 2.1
bounds = [(-10, 10), (-10, 10)]
positions = np.array([np.random.uniform(low, high, num_particles) for low, high in bounds]).T
velocities = np.random.uniform(-1, 1, (num_particles, 2))
personal_best_positions = np.copy(positions)
personal_best_values = np.array([objective_function(p) for p in personal_best_positions])
best_particle_index = np.argmin(personal_best_values)
global_best_position = personal_best_positions[best_particle_index]
global_best_value = personal_best_values[best_particle_index]
for iteration in range(num_iterations):
    for i in range(num_particles):
        fitness = objective_function(positions[i])
        if fitness < personal_best_values[i]:
            personal_best_values[i] = fitness
            personal_best_positions[i] = positions[i]
        if fitness < global_best_value:
            global_best_value = fitness
            global_best_position = positions[i]
    for i in range(num_particles):
        r1 = np.random.rand(2)
        r2 = np.random.rand(2)
        cognitive = c1 * r1 * (personal_best_positions[i] - positions[i])
        social = c2 * r2 * (global_best_position - positions[i])
        velocities[i] = w * velocities[i] + cognitive + social
        positions[i] += velocities[i]
        for dim in range(2):
            positions[i, dim] = np.clip(positions[i, dim], bounds[dim][0], bounds[dim][1])
    print(f"Iteration {iteration+1}/{num_iterations}, Best SSE: {global_best_value:.5f}")
print("\nBest parameters found:")
print("theta_0 =", global_best_position[0])
print("theta_1 =", global_best_position[1])
print("Minimum sum of squared errors:", global_best_value)
```



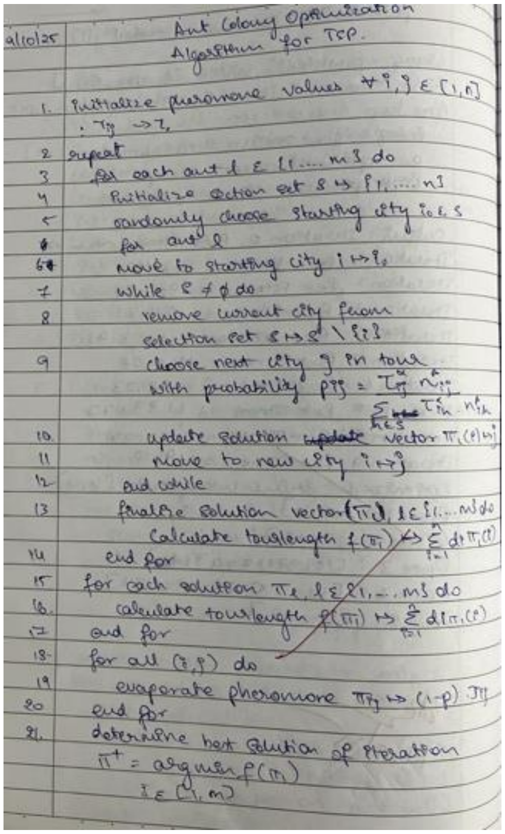
```
Iteration 0, Best Fitness: 912.40
Iteration 1, Best Fitness: 912.40
Iteration 2, Best Fitness: 1035.38
Iteration 3, Best Fitness: 1084.21
Iteration 4, Best Fitness: 1113.91
Iteration 5, Best Fitness: 1160.28
Iteration 6, Best Fitness: 1175.23
Iteration 7, Best Fitness: 1187.77
Iteration 8, Best Fitness: 1193.24
Iteration 9, Best Fitness: 1196.18

Optimized drone waypoints (x,y):
Drone 1: [19. 19.]
Drone 2: [19. 19.]
Drone 3: [15.37337293 19.        ]
Drone 4: [19.          0.65276079]
Drone 5: [19. 19.]
```

Program 4 : Ant Colony Optimization

Problem statement:

The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

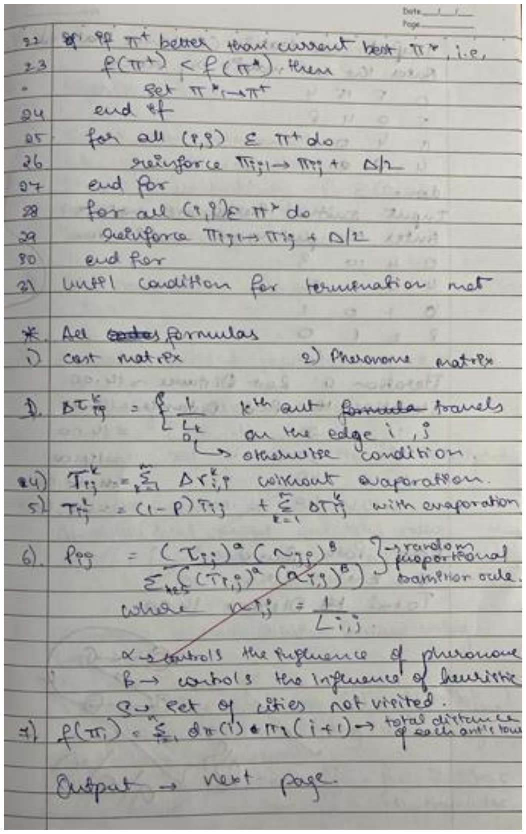
Algorithm:



9/10/25

Ant Colony Optimization Algorithm for TSP.

```
Initialize pheromone values $\forall i, j \in [1,n]$ : $\tau_{ij} \to \tau_0$
repeat
  for each ant $l \in \{1, \dots, m\}$ do
    Initialize selection set $S \mapsto \{1, \dots, n\}$
    randomly choose starting city $i_0 \in S$
    for ant $l$
    move to starting city $i \mapsto i_0$
    while $S \neq \phi$ do
      remove current city from
      selection set $S \mapsto S \setminus \{i\}$
      choose next city $j$ in tour
      with probability $p_{ij} = \frac{\tau_{ij}^{\alpha} \eta_{ij}^{\beta}}{\sum_{h \in S} \tau_{ih}^{\alpha} \eta_{ih}^{\beta}}$
      update solution ~~update~~ vector $\pi_l(i) \mapsto j$
      move to new city $i \mapsto j$
    end while
    finalize solution vector $(\pi_l)$, $l \in \{1, \dots, m\}$ do
      Calculate tourlength $f(\pi_l) \mapsto \sum_{i=1}^{n} d_{i\pi_l(i)}$
  end for
  for each solution $\pi_l$, $l \in \{1, \dots, m\}$ do
    calculate tourlength $f(\pi_l) \mapsto \sum_{i=1}^{n} d_{i\pi_l(i)}$
  end for
  for all $(i,j)$ do
    evaporate pheromone $\pi_{ij} \mapsto (1-\rho)\tau_{ij}$
  end for
  determine best solution of iteration
  $\pi^+ = \arg\min_{l \in [1,m]} f(\pi_l)$
```



```
  if $\pi^+$ better than current best $\pi^*$, i.e.,
    $f(\pi^+) < f(\pi^*)$ then
      set $\pi^* \mapsto \pi^+$
  end if
  for all $(i,j) \in \pi^+$ do
      reinforce $\pi_{ij} \mapsto \pi_{ij} + \Delta/2$
  end for
  for all $(i,j) \in \pi^*$ do
    reinforce $\pi_{ij} \mapsto \pi_{ij} + \Delta/2$
  end for
until condition for termination met
```

* All ~~[illegible]~~ formulas

1) cost matrix 2) Pheromone matrix

3) $\Delta\tau_{ij}^{k} = \left\{ \begin{array}{l} \frac{1}{L_k} \\ 0 \end{array} \right.$ kth ant ~~formula~~ travels on the edge i, j

→ otherwise condition.

4) $\tau_{ij}^{k} = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$ without evaporation.

5) $\tau_{ij}^{k} = (1-\rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$ with evaporation

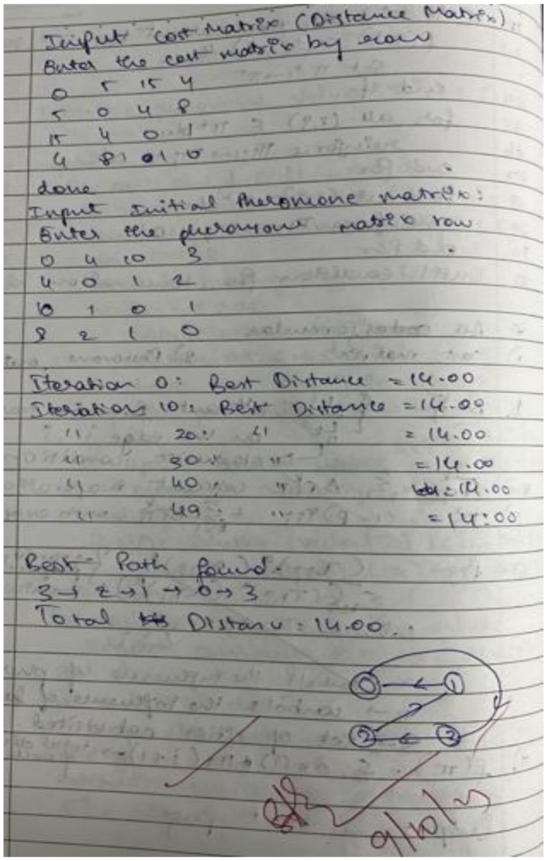
6) $p_{ij} = \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum_{h \in S}((\tau_{ij})^{\alpha}(\alpha_{ij})^{\beta})}$ } → random proportional transition rule.

where $\eta_{ij} = \frac{1}{L_{ij}}$

α → controls the influence of pheromone
β → controls the influence of heuristic
S → set of cities not visited.

7) $f(\pi_l) = \sum_{i=1}^{n} d_{\pi(i)\pi_l(i+1)}$ → total distance of each ant's tour

Output → next page.

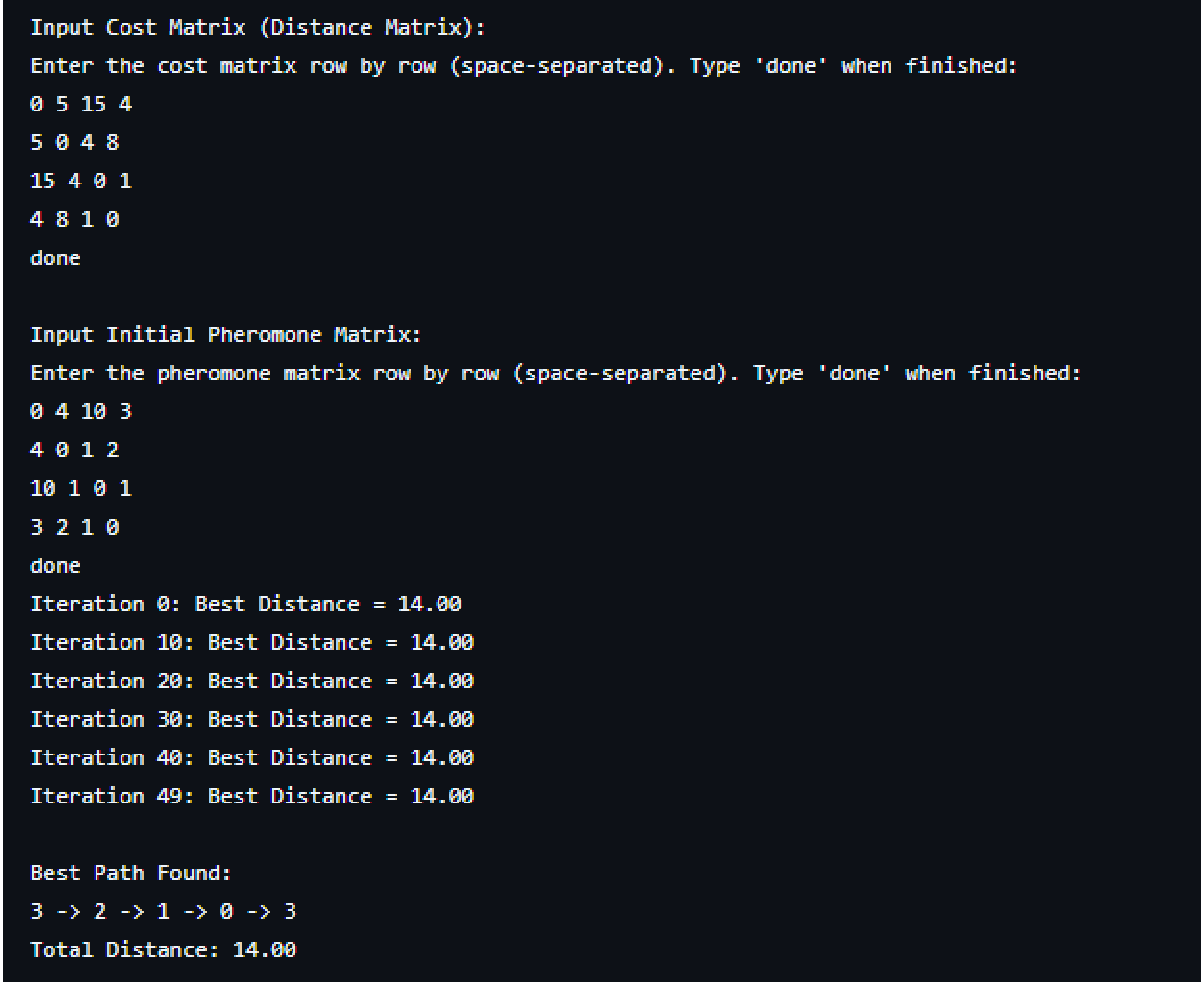


```
Input cost Matrix (Distance Matrix)
Enter the cost matrix by row
0  5  15  4
5  0  4   8
15 4  0   1
4  8  1   0
done
Input Initial Pheromone matrix:
Enter the pheromone matrix row
0  4  10  3
4  0  1   2
10 1  0   1
3  2  1   0

Iteration 0: Best Distance = 14.00
Iteration 10: Best Distance = 14.00
   "      20:    "          = 14.00
   "      30:    "          = 14.00
   "      40:    "          = 14.00
   "      49:    "          = 14.00

Best Path found:
3 → 2 → 1 → 0 → 3
Total ~~[illegible]~~ Distance : 14.00
```

9/10/25

Code:

```
import numpy as np
import random
NUM_CITIES = 10
NUM_ANTS = 20
NUM_ITERATIONS = 100
ALPHA = 1.0
BETA = 5.0
EVAPORATION = 0.5
Q = 100
np.random.seed(42)
cities = np.random.rand(NUM_CITIES, 2) * 100
dist_matrix = np.sqrt(((cities[:, np.newaxis, :] - cities[np.newaxis, :, :]) ** 2).sum(axis=2))
pheromone = np.ones((NUM_CITIES, NUM_CITIES))
best_distance = float('inf')
best_path = []
for iteration in range(NUM_ITERATIONS):
    all_paths = []
    all_distances = []
    for ant in range(NUM_ANTS):
        path = [random.randint(0, NUM_CITIES - 1)]
        while len(path) < NUM_CITIES:
            current_city = path[-1]
            probabilities = []
            for next_city in range(NUM_CITIES):
                if next_city not in path:
                    tau = pheromone[current_city][next_city] ** ALPHA
                    eta = (1 / dist_matrix[current_city][next_city]) ** BETA
                    probabilities.append(tau * eta)
                else:
                    probabilities.append(0)
            probabilities = np.array(probabilities)
            probabilities /= probabilities.sum()
            next_city = np.random.choice(range(NUM_CITIES), p=probabilities)
            path.append(next_city)
        path.append(path[0])  # Return to starting city
        distance = sum(dist_matrix[path[i]][path[i + 1]] for i in range(NUM_CITIES))
        all_paths.append(path)
        all_distances.append(distance)
        if distance < best_distance:
            best_distance = distance
            best_path = path
    pheromone *= (1 - EVAPORATION)
    for i in range(NUM_ANTS):
        for j in range(NUM_CITIES):
            from_city = all_paths[i][j]
            to_city = all_paths[i][j + 1]
            pheromone[from_city][to_city] += Q / all_distances[i]
            pheromone[to_city][from_city] += Q / all_distances[i]
    if iteration % 10 == 0 or iteration == NUM_ITERATIONS - 1:
        print(f"Iteration {iteration}: Best Distance = {best_distance:.2f}")
print("\nBest Path Found:")
print(" -> ".join(map(str, best_path)))
print(f"Total Distance: {best_distance:.2f}")
```



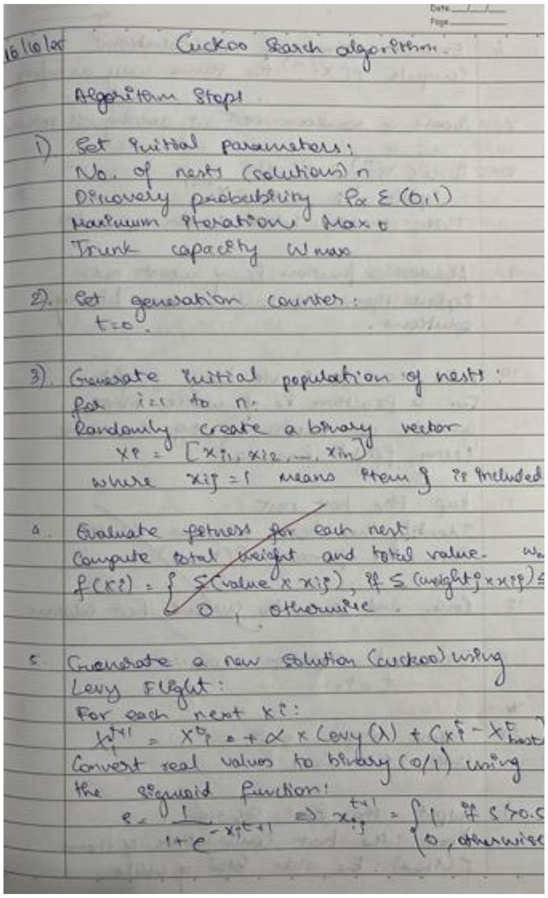
```
Input Cost Matrix (Distance Matrix):
Enter the cost matrix row by row (space-separated). Type 'done' when finished:
0 5 15 4
5 0 4 8
15 4 0 1
4 8 1 0
done

Input Initial Pheromone Matrix:
Enter the pheromone matrix row by row (space-separated). Type 'done' when finished:
0 4 10 3
4 0 1 2
10 1 0 1
3 2 1 0
done
Iteration 0: Best Distance = 14.00
Iteration 10: Best Distance = 14.00
Iteration 20: Best Distance = 14.00
Iteration 30: Best Distance = 14.00
Iteration 40: Best Distance = 14.00
Iteration 49: Best Distance = 14.00

Best Path Found:
3 -> 2 -> 1 -> 0 -> 3
Total Distance: 14.00
```

Program 5 : Cuckoo search Optimization

Problem statement:

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Algorithm:


16/10/25 Cuckoo Search algorithm.

Algorithm Steps

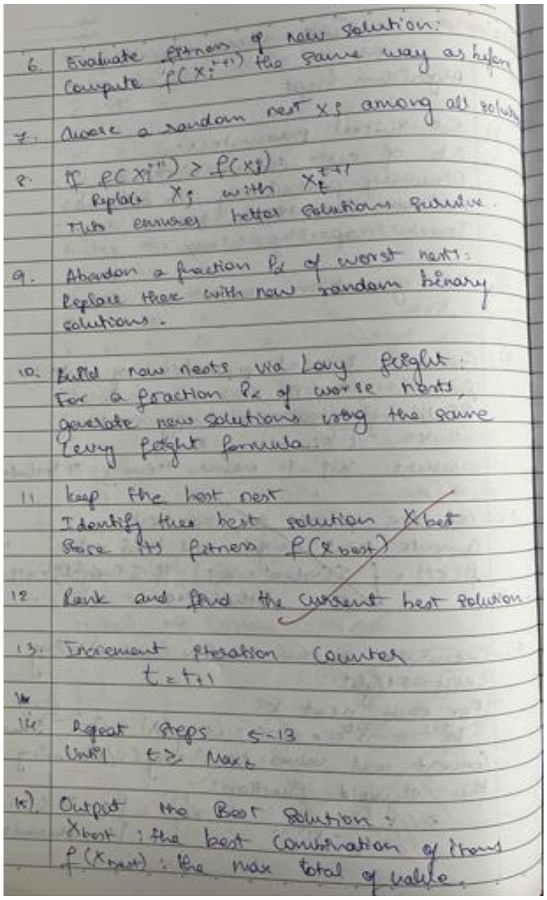
1) Set initial parameters;
No. of nests (solutions) n
Discovery probability $P_a \in (0,1)$
Maximum iteration Max_t
Trunk capacity W_{max}

2). Set generation counter:
$t=0$.

3). Generate initial population of nests:
for $i=1$ to n:
Randomly create a binary vector
$X_i = [x_{i1}, x_{i2}, \ldots, x_{in}]$
where $x_{ij} = 1$ means item j is included

4. Evaluate fitness for each nest.
Compute total weight and total value. W
$f(X_i) = \begin{cases} \sum(value \times x_{ij}), & \text{if } \sum(weight_j \times x_{ij}) \leq \\ 0, & \text{otherwise} \end{cases}$

5. Generate a new solution (cuckoo) using Levy Flight:
For each nest X_i:
$X_i^{t+1} = X_i^t + \alpha \times Levy(\lambda) + (X_i^t - X_{best}^t)$
Convert real values to binary (0/1) using the sigmoid function:
$s = \frac{1}{1+e^{-x_i^{t+1}}} \Rightarrow x_{ij}^{t+1} = \begin{cases} 1 & \text{if } s > 0.5 \\ 0, & \text{otherwise} \end{cases}$


6. Evaluate fitness of new solution:
Compute $f(X_i^{t+1})$ the same way as before

7. Choose a random nest X_j among all solutions

8. If $f(X_i^{t+1}) > f(X_j)$:
Replace X_j with X_i^{t+1}
This ensures better solutions survive.

9. Abandon a fraction P_a of worst nests:
Replace these with new random binary solutions.

10. Build new nests via Levy flight:
For a fraction P_a of worse nests,
generate new solutions using the same Levy flight formula.

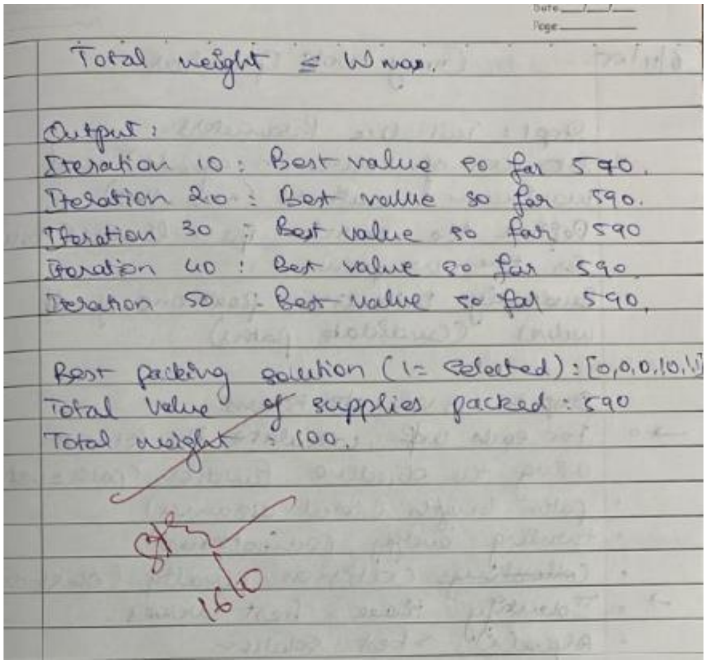
11. keep the best nest
Identify the best solution X_{best}
Store its fitness $f(X_{best})$

12. Rank and find the current best solution

13. Increment iteration counter
$t = t+1$

14. Repeat steps 5-13
Until $t \geq Max_t$

15). Output the Best Solution:
X_{best} : the best combination of items
$f(X_{best})$: the max total of value.


Total weight ≤ W max.

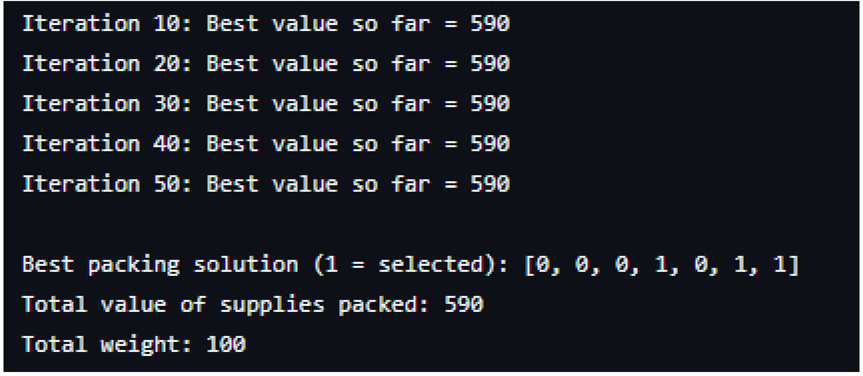
Output:
Iteration 10 : Best value so far 590.
Iteration 20 : Best value so far 590.
Iteration 30 : Best value so far 590
Iteration 40 : Best value so far 590.
Iteration 50 : Best value so far 590.

Best packing solution (1= Selected) : [0,0,0,1,0,1,1]
Total Value of supplies packed : 590
Total weight : 100.

16/10

Code:

```
import random
import math
weights = [10, 20, 30, 40, 15, 25, 35]
values = [60, 100, 120, 240, 80, 150, 200]
capacity = 100  # Max weight capacity of the truck
n_items = len(weights)
n_nests = 15
max_iter = 50
pa = 0.25
def fitness(solution):
    total_weight = sum(w for w, s in zip(weights, solution) if s == 1)
    total_value = sum(v for v, s in zip(values, solution) if s == 1)
    if total_weight > capacity:
        return 0  # Penalize overweight solutions
    else:
        return total_value
def generate_nest():
    return [random.randint(0, 1) for _ in range(n_items)]
def levy_flight(Lambda=1.5):
    sigma_u = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
              (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    u = random.gauss(0, sigma_u)
    v = random.gauss(0, 1)
    step = u / (abs(v) ** (1 / Lambda))
    return step
def get_cuckoo(nest, best_nest):
    new_nest = []
    for xi, bi in zip(nest, best_nest):
        step = levy_flight()
        val = xi + step * (xi - bi)
        s = 1 / (1 + math.exp(-val))
        new_val = 1 if s > 0.5 else 0
        new_nest.append(new_val)
    return new_nest
def cuckoo_search():
    nests = [generate_nest() for _ in range(n_nests)]
    fitness_values = [fitness(nest) for nest in nests]
    best_index = fitness_values.index(max(fitness_values))
    best_nest = nests[best_index][:]
    best_fitness = fitness_values[best_index]
    for _ in range(max_iter):
        for i in range(n_nests):
            new_nest = get_cuckoo(nests[i], best_nest)
            new_fitness = fitness(new_nest)
            if new_fitness > fitness_values[i]:
                nests[i] = new_nest
                fitness_values[i] = new_fitness
        for i in range(n_nests):
            if random.random() < pa:
                nests[i] = generate_nest()
                fitness_values[i] = fitness(nests[i])
        current_best_index = fitness_values.index(max(fitness_values))
        current_best_fitness = fitness_values[current_best_index]
        if current_best_fitness > best_fitness:
            best_fitness = current_best_fitness
            best_nest = nests[current_best_index][:]
    return best_nest, best_fitness
if _name____== "_main_":
    best_solution, best_value = cuckoo_search()
    total_weight = sum(w for w, s in zip(weights, best_solution) if s == 1)
    print(f"Best packing solution (1 = selected): {best_solution}")
    print(f"Total value of supplies packed: {best_value}")
    print(f"Total weight: {total_weight}")
```



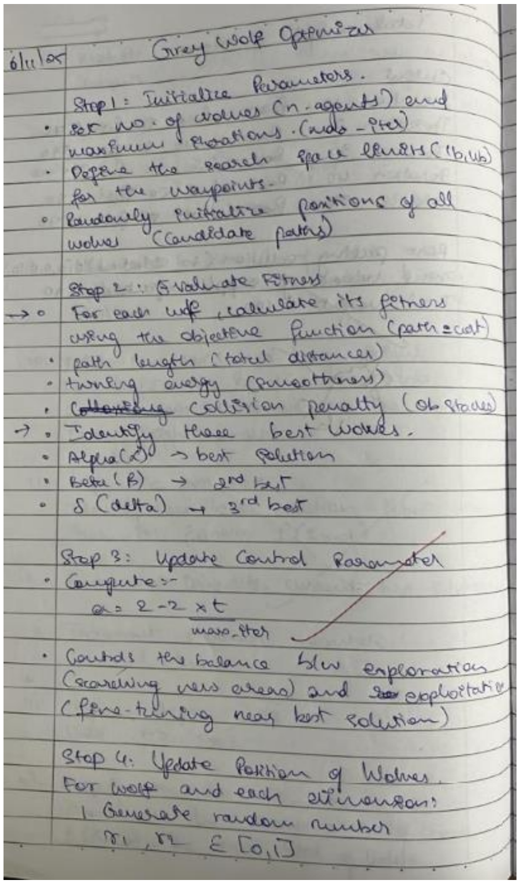
```
Iteration 10: Best value so far = 590
Iteration 20: Best value so far = 590
Iteration 30: Best value so far = 590
Iteration 40: Best value so far = 590
Iteration 50: Best value so far = 590

Best packing solution (1 = selected): [0, 0, 0, 1, 0, 1, 1]
Total value of supplies packed: 590
Total weight: 100
```

Program 6 : Grey Wolf Optimization

Problem statement:

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta,delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

Algorithm:



6/11/25 Grey Wolf Optimizer

Step 1: Initialize Parameters.

- Set no. of wolves (n-agents) and maximum iterations. (max-iter)
- Define the search space limits (lb, ub) for the waypoints.
- Randomly initialize positions of all wolves (candidate paths)

Step 2: Evaluate Fitness

→ For each wolf, calculate its fitness using the objective function (path = cost)

- path length (total distances)
- turning energy (smoothness)
- ~~Colliding~~ collision penalty (obstacles)

→ Identify three best wolves.

- Alpha (α) → best solution
- Beta (β) → 2nd best
- δ (delta) → 3rd best

Step 3: Update Control Parameter

- Compute:-

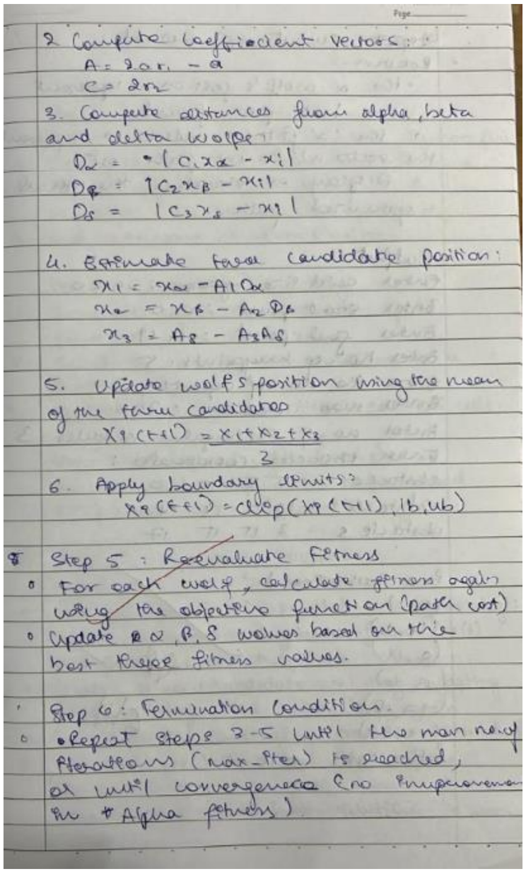
$$a = 2 - 2\frac{\times t}{max\_iter}$$

- Controls the balance b/w exploration (searching new areas) and exploitation (fine-tuning near best solution)

Step 4: Update Position of Wolves.

For wolf and each dimension:

1 Generate random number $r_1, r_2 \in [0,1]$



2 Compute coefficient vectors:

$A = 2a r_1 - a$

$C = 2r_2$

3. Compute distances from alpha, beta and delta wolfe

$D_\alpha = |C_1 x_\alpha - x_i|$

$D_\beta = |C_2 x_\beta - x_i|$

$D_\delta = |C_3 x_\delta - x_i|$

4. Estimate three candidate position:

$x_1 = x_\alpha - A_1 D_\alpha$

$x_2 = x_\beta - A_2 D_\beta$

$x_3 = A_\delta - A_3 A_\delta$

5. Update wolf's position using the mean of the three candidates

$$X_i(t+1) = \frac{x_1 + x_2 + x_3}{3}$$

6. Apply boundary limits:

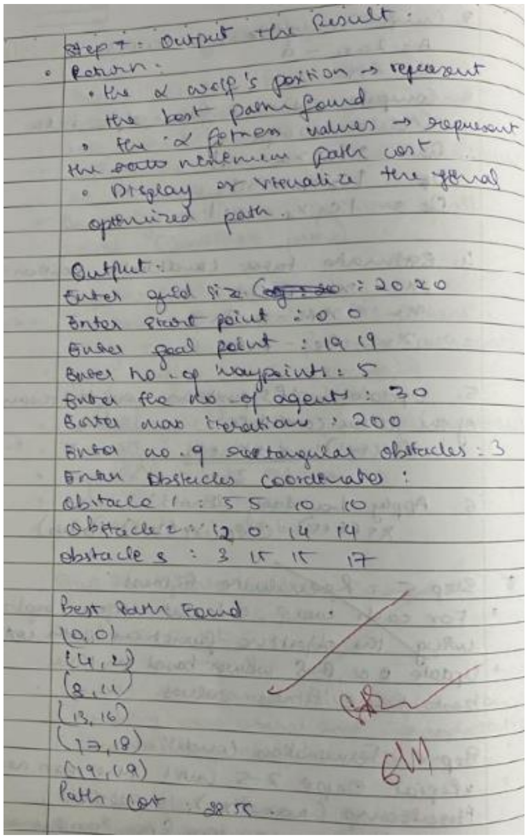
$X_i(t+1) = clip(X_i(t+1), lb, ub)$

Step 5: Reevaluate Fitness

- For each wolf, calculate fitness again using the objective function (path cost)
- Update α, β, δ wolves based on the best three fitness values.

Step 6: Termination condition.

- Repeat steps 3-5 until the max no. of iterations (max_iter) is reached, or until convergence (no improvement in Alpha fitness)



Step 7: Output the Result:

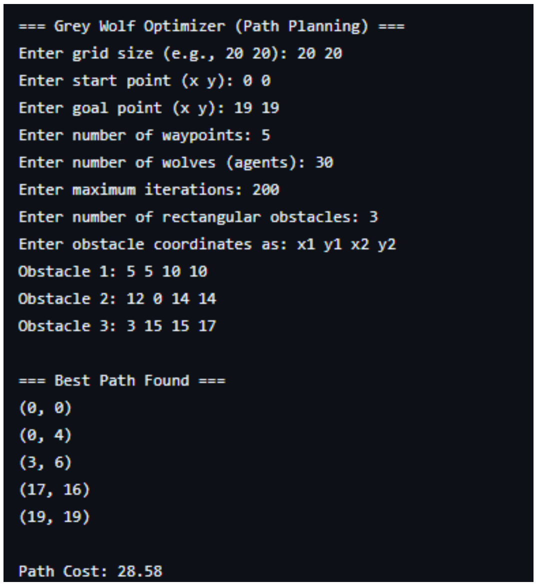
- Return:
  - the α wolf's position → represent the best path found
  - the α fitness values → represent the minimum path cost
  - Display or visualize the final optimized path.

Output:

```
Enter grid size (~~eg: 20~~) : 20 20
Enter start point : 0 0
Enter goal point : 19 19
Enter no. of waypoints : 5
Enter the no. of agents : 30
Enter max iterations : 200
Enter no. of rectangular obstacles : 3
Enter obstacles coordinates :
obstacle 1 : 5 5 10 10
obstacle 2 : 12 0 14 14
obstacle 3 : 3 15 15 17

Best Path Found :
(0,0)
(4,2)
(8,11)
(13,16)
(17,18)
(19,19)
Path cost : 28.55.
```

Code:

```
import numpy as np
def gwo(obj_func, dim, search_space, n_agents=20, max_iter=100):
    lb, ub = search_space
    wolves = np.random.uniform(lb, ub, (n_agents, dim))
    alpha, beta, delta = None, None, None
    alpha_score, beta_score, delta_score = float("inf"), float("inf"), float("inf")
    for t in range(max_iter):
        for i in range(n_agents):
            fitness = obj_func(wolves[i])
            if fitness < alpha_score:
                delta_score, delta = beta_score, beta
                beta_score, beta = alpha_score, alpha
                alpha_score, alpha = fitness, wolves[i].copy()
            elif fitness < beta_score:
                delta_score, delta = beta_score, beta
                beta_score, beta = fitness, wolves[i].copy()
            elif fitness < delta_score:
                delta_score, delta = fitness, wolves[i].copy()
        a = 2 - 2 * (t / max_iter)
        for i in range(n_agents):
            for j in range(dim):
                r1, r2 = np.random.rand(), np.random.rand()
                A1, C1 = 2 * a * r1 - a, 2 * r2
                D_alpha = abs(C1 * alpha[j] - wolves[i][j])
                X1 = alpha[j] - A1 * D_alpha
                r1, r2 = np.random.rand(), np.random.rand()
                A2, C2 = 2 * a * r1 - a, 2 * r2
                D_beta = abs(C2 * beta[j] - wolves[i][j])
                X2 = beta[j] - A2 * D_beta
                r1, r2 = np.random.rand(), np.random.rand()
                A3, C3 = 2 * a * r1 - a, 2 * r2
                D_delta = abs(C3 * delta[j] - wolves[i][j])
                X3 = delta[j] - A3 * D_delta
                wolves[i][j] = np.clip((X1 + X2 + X3) / 3, lb, ub)
    return alpha, alpha_score
grid_size = (20, 20)
start, goal = np.array([0, 0]), np.array([19, 19])
obstacles = [
    (5, 5, 10, 10),
    (12, 0, 14, 14),
    (3, 15, 15, 17)
]
def is_collision(point):
    x, y = point.astype(int)
    if x < 0 or y < 0 or x >= grid_size[0] or y >= grid_size[1]:
        return True
    for ox1, oy1, ox2, oy2 in obstacles:
        if ox1 <= x <= ox2 and oy1 <= y <= oy2:
            return True
    return False
    waypoints = waypoints.reshape(-1, 2)
    path = [start] + [w.astype(int) for w in waypoints] + [goal]
    total_dist, penalty = 0, 0
    for i in range(len(path) - 1):
        dist = np.linalg.norm(path[i + 1] - path[i])
        total_dist += dist
        if is_collision(path[i + 1]):
            penalty += 100
    energy = 0
    for i in range(1, len(path) - 1):
        v1 = path[i] - path[i - 1]
        v2 = path[i + 1] - path[i]
        if np.linalg.norm(v1) > 0 and np.linalg.norm(v2) > 0:
            cos_angle = np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))
            angle = np.arccos(np.clip(cos_angle, -1, 1))
            energy += angle
    return total_dist + energy * 5 + penalty
n_waypoints = 5  # intermediate waypoints
dim = n_waypoints * 2
best_path, best_score = gwo(path_cost, dim, (0, grid_size[0]-1), n_agents=30, max_iter=200)
best_waypoints = best_path.reshape(-1, 2).astype(int)
final_path = np.vstack([start, best_waypoints, goal]) clean_path = []
for p in final_path:
    pt = tuple(map(int, p))
    if len(clean_path) == 0 or pt != clean_path[-1]:
        clean_path.append(pt)
print("Best Path Found:")
for p in clean_path:
    print(p)
print("\nPath Cost:", round(best_score, 2))
```



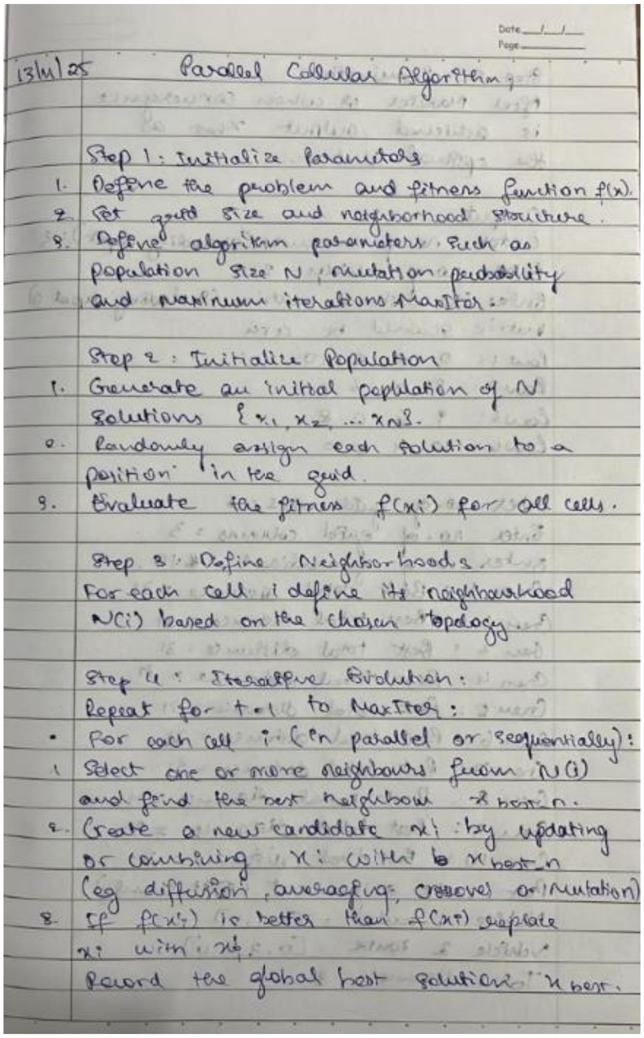
```
=== Grey Wolf Optimizer (Path Planning) ===
Enter grid size (e.g., 20 20): 20 20
Enter start point (x y): 0 0
Enter goal point (x y): 19 19
Enter number of waypoints: 5
Enter number of wolves (agents): 30
Enter maximum iterations: 200
Enter number of rectangular obstacles: 3
Enter obstacle coordinates as: x1 y1 x2 y2
Obstacle 1: 5 5 10 10
Obstacle 2: 12 0 14 14
Obstacle 3: 3 15 15 17

=== Best Path Found ===
(0, 0)
(0, 4)
(3, 6)
(17, 16)
(19, 19)

Path Cost: 28.58
```

Program 7 : Parallel cellular Optimization

Problem statement:

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Algorithm:



Date__/__/__
Page____

13/11/25 Parallel Cellular Algorithm

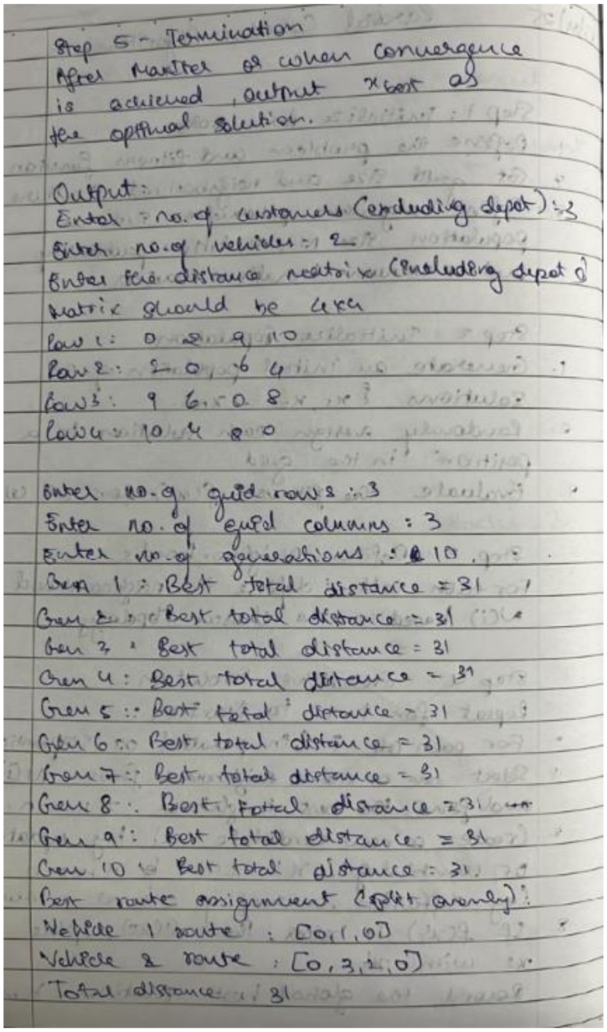
Step 1: Initialize Parameters
1. Define the problem and fitness function $f(x)$.
2. Set grid size and neighborhood structure.
3. Define algorithm parameters such as population size N, mutation probability and maximum iterations MaxIter.

Step 2: Initialize Population
1. Generate an initial population of N solutions $\{x_1, x_2, \dots x_N\}$.
2. Randomly assign each solution to a position in the grid.
3. Evaluate the fitness $f(x_i)$ for all cells.

Step 3: Define Neighborhoods
For each cell i define its neighbourhood N(i) based on the chosen topology.

Step 4: Iterative Evolution:
Repeat for t=1 to MaxIter:
- For each cell i (in parallel or sequentially):
1. Select one or more neighbours from N(i) and find the best neighbour x_{best_n}.
2. Create a new candidate x_i' by updating or combining x_i with x_{best_n} (eg diffusion, averaging, crossover or mutation)
3. If $f(x_i')$ is better than $f(x_i)$ replace x_i with x_i'.

Record the global best solution x_{best}.



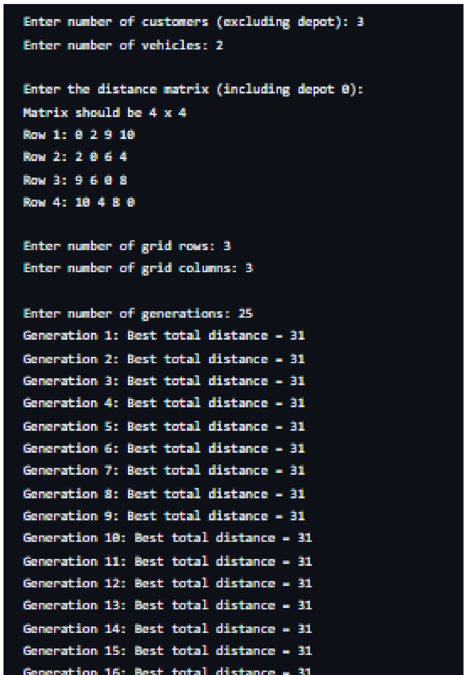
Step 5 - Termination
After MaxIter or when convergence is achieved, output x_{best} as the optimal solution.

Output:

```
Enter no. of customers (excluding depot): 3
Enter no. of vehicles: 2
Enter the distance matrix (including depot 0)
Matrix should be 4x4
Row 1: 0 2 9 10
Row 2: 2 0 6 4
Row 3: 9 6 0 8
Row 4: 10 4 8 0

Enter no. of grid rows: 3
Enter no. of grid columns: 3
Enter no. of generations: 10
Gen 1: Best total distance = 31
Gen 2: Best total distance = 31
Gen 3: Best total distance = 31
Gen 4: Best total distance = 31
Gen 5: Best total distance = 31
Gen 6: Best total distance = 31
Gen 7: Best total distance = 31
Gen 8: Best total distance = 31
Gen 9: Best total distance = 31
Gen 10: Best total distance = 31
Best route assignment (split evenly):
Vehicle 1 route: [0, 1, 0]
Vehicle 2 route: [0, 3, 2, 0]
Total distance: 31
```

Code:

```
import numpy as np
import random
from itertools import permutations
distance_matrix = np.array([
    [0, 2, 9, 10],
    [2, 0, 6, 4],
    [9, 6, 0, 8],
    [10, 4, 8, 0]
])
num_customers = distance_matrix.shape[0] - 1
population_size = 9
grid_dim = (3, 3)
num_vehicles = 2
def generate_individual():
    perm = list(range(1, num_customers + 1))
    random.shuffle(perm)
    return perm
population = [generate_individual() for _ in range(population_size)]
def fitness(individual):
    split_points = np.linspace(0, num_customers, num_vehicles + 1, dtype=int)
    total_distance = 0
    for i in range(num_vehicles):
        route = [0] + individual[split_points[i]:split_points[i+1]] + [0]  # depot at start and end
        for j in range(len(route) - 1):
            total_distance += distance_matrix[route[j], route[j+1]]
    return total_distance
def get_neighbors(idx):
    r, c = divmod(idx, grid_dim[1])
    neighbors = []
    for dr in [-1, 0, 1]:
        for dc in [-1, 0, 1]:
            nr, nc = r + dr, c + dc
            if 0 <= nr < grid_dim[0] and 0 <= nc < grid_dim[1]:
                n_idx = nr * grid_dim[1] + nc
                if n_idx != idx:
                    neighbors.append(n_idx)
    return neighbors
def crossover(parent1, parent2):
    size = len(parent1)
    a, b = sorted(random.sample(range(size), 2))
    child = [None] * size
    child[a:b] = parent1[a:b]
    pointer = b
    for gene in parent2[b:] + parent2[:b]:
        if gene not in child:
            if pointer == size:
                pointer = 0
            child[pointer] = gene
            pointer += 1
    return child
def mutate(individual):
    a, b = random.sample(range(len(individual)), 2)
    individual[a], individual[b] = individual[b], individual[a]
    return individual
def pca_iteration(pop):
    new_pop = pop.copy()
    for idx in range(len(pop)):
        neighbors = get_neighbors(idx)
        partner_idx = random.choice(neighbors)
        parent1 = pop[idx]
        parent2 = pop[partner_idx]
        child = crossover(parent1, parent2)
        if random.random() < 0.2:
            child = mutate(child)
        if fitness(child) < fitness(pop[idx]):
            new_pop[idx] = child
    return new_pop
num_generations = 25
for gen in range(num_generations):
    population = pca_iteration(population)
    best_fitness = min(fitness(ind) for ind in population)
    print(f"Generation {gen+1}: Best total distance = {best_fitness}")
best_individual = min(population, key=fitness)
print("\nBest route assignment (split evenly):")
split_points = np.linspace(0, num_customers, num_vehicles + 1, dtype=int)
for i in range(num_vehicles):
    route = [0] + best_individual[split_points[i]:split_points[i+1]] + [0]
    print(f"Vehicle {i+1} route: {route}")
print(f"Total distance: {fitness(best_individual)}")
```



```
Enter number of customers (excluding depot): 3
Enter number of vehicles: 2

Enter the distance matrix (including depot 0):
Matrix should be 4 x 4
Row 1: 0 2 9 10
Row 2: 2 0 6 4
Row 3: 9 6 0 8
Row 4: 10 4 8 0

Enter number of grid rows: 3
Enter number of grid columns: 3

Enter number of generations: 25
Generation 1: Best total distance = 31
Generation 2: Best total distance = 31
Generation 3: Best total distance = 31
Generation 4: Best total distance = 31
Generation 5: Best total distance = 31
Generation 6: Best total distance = 31
Generation 7: Best total distance = 31
Generation 8: Best total distance = 31
Generation 9: Best total distance = 31
Generation 10: Best total distance = 31
Generation 11: Best total distance = 31
Generation 12: Best total distance = 31
Generation 13: Best total distance = 31
Generation 14: Best total distance = 31
Generation 15: Best total distance = 31
Generation 16: Best total distance = 31
```