VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Harshitha H G (1BM23CS108)

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 **Harshitha H G (1BM23CS108),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Sowmya T Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

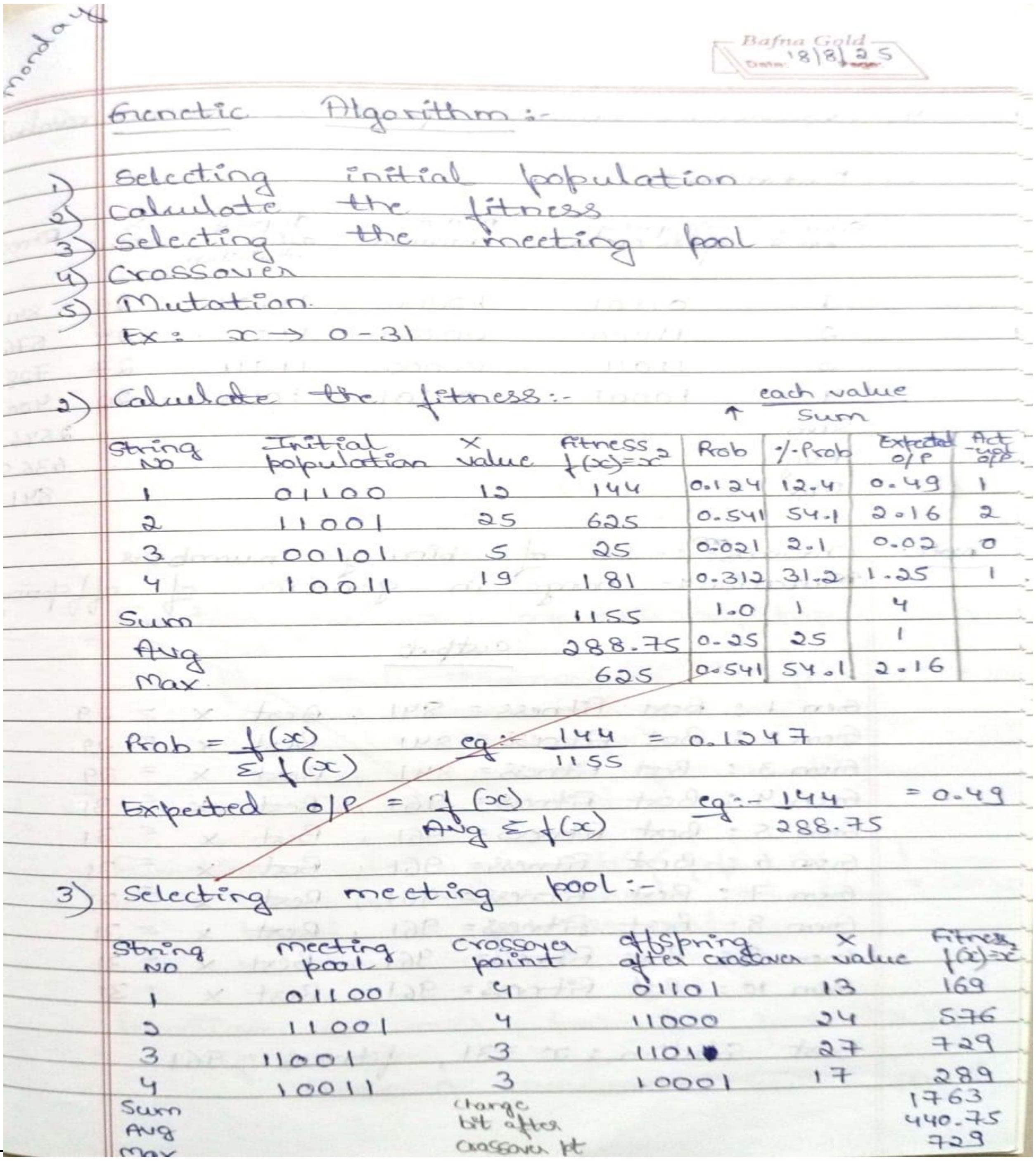
Github Link:
https://github.com/Harshithahrgopal/Bio-Inspired-System-BIS-

Program 1

Genetic Algorithm for Optimization Problems:

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. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:



monday

Bafna Gold Date: 18/8/25

Genetic Algorithm :-

1) Selecting initial population
2) calculate the fitness
3) Selecting the meeting pool
4) Crossover
5) Mutation

Ex : $x \rightarrow 0-31$

2) Calculate the fitness :-

↑ $\frac{\text{each value}}{\text{Sum}}$

String No	Initial population	x value	fitness $f(x)=x^2$	Prob	%Prob	Expected o/p	Actual o/p
1	01100	12	144	0.124	12.4	0.49	1
2	11001	25	625	0.541	54.1	2.16	2
3	00101	5	25	0.021	2.1	0.02	0
4	10011	19	181	0.312	31.2	1.25	1
Sum			1155	1.0	1	4	
Avg			288.75	0.25	25	1	
Max			625	0.541	54.1	2.16	

$\text{Prob} = \frac{f(x)}{\Sigma f(x)}$ eg:- $\frac{144}{1155} = 0.1247$

$\text{Expected o/p} = \frac{f(x)}{\text{Avg } \Sigma f(x)}$ eg:- $\frac{144}{288.75} = 0.49$

3) Selecting meeting pool :-

String No	meeting pool	crossover point	offspring after crossover	x value	Fitness $f(x)=x^2$
1	01100	4	01101	13	169
2	11001	4	11000	24	576
3	11001	3	1101●	27	729
4	10011	3	10001	17	289
Sum		change bit after crossover pt			1763
Avg					440.75
Max					729

4) Crossover :- Crossover point is choosen randomly

5) Mutation :-

String no	offspring after crossover	mutation chromosome changing pt	offspring after mutation	x value	fitness
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400
Sum					2546
Avg					636.5
Max					841

Note:- Chromosome :- Set of binary numbers
Mutation :- change in genetics of offspring

output

```
Gen 1 : Best Fitness = 841 , Best x = 29
Gen 2 : Best Fitness = 841 , Best x = 29
Gen 3 : Best Fitness = 841 , Best x = 29
Gen 4 : Best Fitness = 961 , Best x = 31
Gen 5 : Best Fitness = 961 , Best x = 31
Gen 6 : Best Fitness = 961 , Best x = 31
Gen 7 : Best Fitness = 961 , Best x = 31
Gen 8 : Best Fitness = 961 , Best x = 31
Gen 9 : Best Fitness = 961 , Best x = 31
Gen 10 : Best Fitness = 961 , Best x = 31

Best Solution : x = 31, fitness = 961
```

Bafna Gold
Date: Page:

1) Pseudo for Genetic Algorithm:-

```
function fitness (x):
    return x* x

function decode (chromosome):
    Convert binary list to decimal number
    return decimal_value

function create - population ():
    population = []
    for i = 1 to 10:
        chromose = random list of 5 bits
        Add chromosome to population
    return population.

function evaluate_population (population):
    fitness_list = []
    for each chromosome in population:
        x = decode (Chromosome)
        f = fitness (x)
        Add f to fitness_list
    return fitness list.

function select_parents (population, fitness list):
    Use roulette wheel Selection
    based on fitness values
    return selected_parents

function crossover (parent 1, parent 2):
    if random <0.7:
        chose a random crossover point
        child1 = first part of parent1 + second
                 part of parent2
        child 2 = first part of parent2 +
                  second part of parent1
    Else:
        child1 = copy of parent1
        child 2 = copy of parent 2
    return child1, child2

Function mutate (chromosome):
    for each bit in chromosome:
        if random < 0.1:
            flip bit (0 → 1, 1 → 0)
    return chromosome

Function genetic_algorithm():
    population = create_population()
    best chromosome = None
    best_fitness = - infinity

    For generation = 1 to 10:
        fitness_list = evaluate_population(population)
        find chromosome with highest fitness
        if this fitness > best_fitness:
            best_chromosome = that chromosome
            best_fitness = its fitness

        Print generation number, best x and best fitness

        Selected = Select_parents(population, fitness_list)

        next generation = []
```

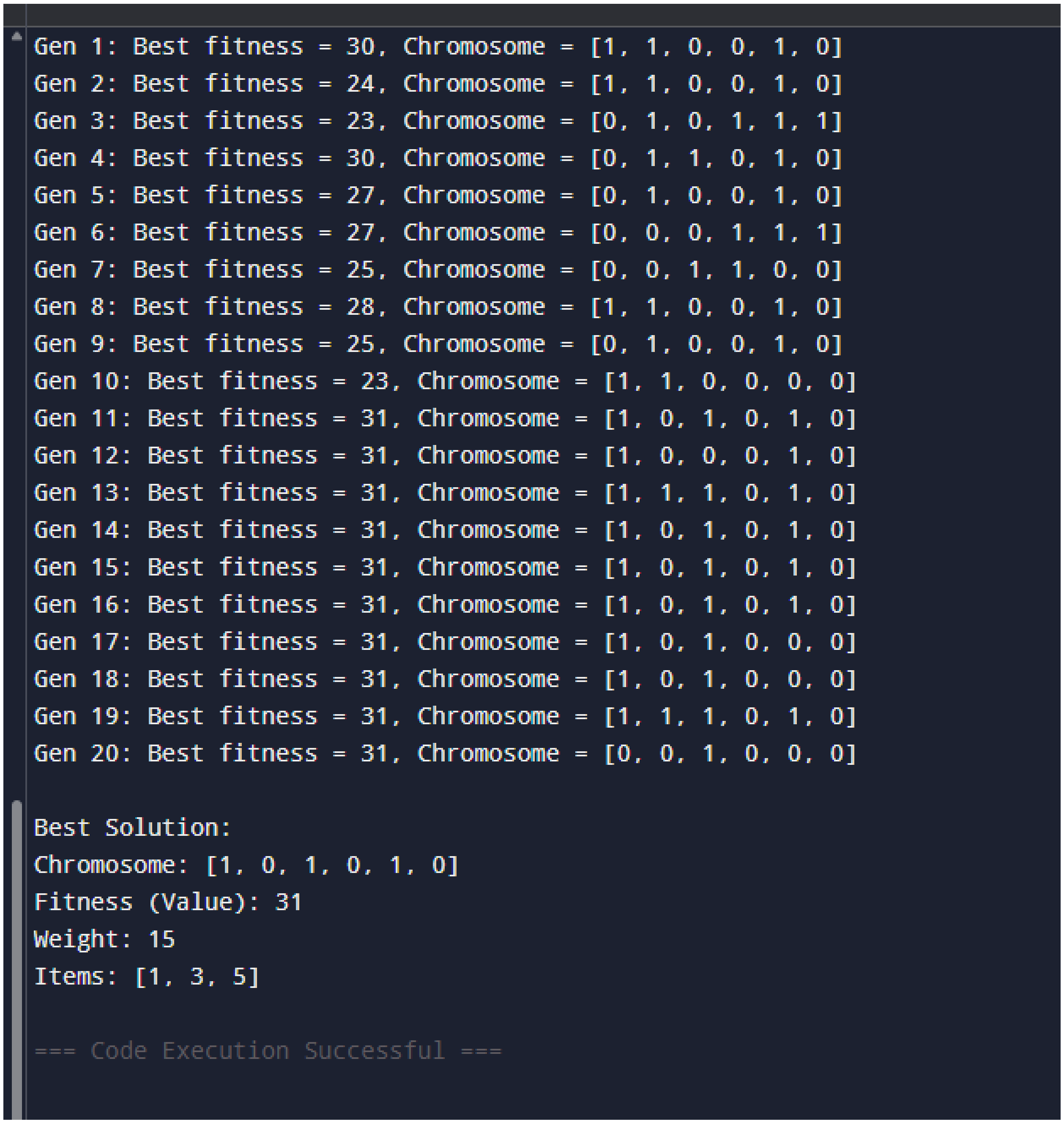
```
for i = 0 to population_size    Step 2:
    parent1 = Selected [i]
    parent2 = Selected [i+1]
    child1, child2 = crossover (parent1, parent2)
    child1 = mutate (child1)
    child2 = mutate (child2)
    Add child1 and child2 to next_generation

  population = next_generation
return decode (best_chromosome), best_fitness

Call genetic_algorithm ()
```

25/8/25

Code:

```
import random

# Items: (weight, value)
items = [
    (2, 6),   # Item1
    (5, 10),  # Item2
    (10, 18), # Item3
    (8, 12),  # Item4
    (3, 7),   # Item5
    (7, 14)   # Item6
]
capacity = 15

# Parameters
POP_SIZE = 6
GENS = 20
CROSS_RATE = 0.8
MUT_RATE = 0.1

# Generate initial population
def init_population():
    return [[random.randint(0, 1) for _ in range(len(items))] for _ in range(POP_SIZE)]

# Fitness function
def fitness(chromosome):
    total_weight, total_value = 0, 0
    for gene, (w, v) in zip(chromosome, items):
        if gene == 1:
            total_weight += w
            total_value += v
    return total_value if total_weight <= capacity else 0

# Roulette Wheel Selection
def selection(pop, fits):
    total_fit = sum(fits)
    if total_fit == 0:
        return random.choice(pop)
    pick = random.uniform(0, total_fit)
    current = 0
    for chromosome, fit in zip(pop, fits):
        current += fit
        if current > pick:
            return chromosome

# Crossover (single point)
def crossover(p1, p2):
    if random.random() < CROSS_RATE:
        point = random.randint(1, len(p1)-1)
        return p1[:point] + p2[point:], p2[:point] + p1[point:]
    return p1, p2

# Mutation (bit flip)
def mutate(chromosome):
    return [1-g if random.random() < MUT_RATE else g for g in chromosome]

# Run GA
def genetic_algorithm():
    population = init_population()

    for gen in range(GENS):
        fits = [fitness(ch) for ch in population]
        new_pop = []

        for _ in range(POP_SIZE // 2):
            p1, p2 = selection(population, fits), selection(population, fits)
            c1, c2 = crossover(p1, p2)
            new_pop.extend([mutate(c1), mutate(c2)])

        population = new_pop
        best_fit = max(fits)
        best_ch = population[fits.index(best_fit)]
        print(f"Gen {gen+1}: Best fitness = {best_fit}, Chromosome = {best_ch}")

    # Final best
    final_fits = [fitness(ch) for ch in population]
    best_fit = max(final_fits)
    best_ch = population[final_fits.index(best_fit)]
    print("\nBest Solution:")
    print("Chromosome:", best_ch)
    print("Fitness (Value):", best_fit)
    print("Weight:", sum(w for g,(w,_) in zip(best_ch, items) if g==1))
    print("Items:", [i+1 for i,g in enumerate(best_ch) if g==1])

# Run
if _name___== "_main_":
    genetic_algorithm()
```

Output:



```
Gen 1: Best fitness = 30, Chromosome = [1, 1, 0, 0, 1, 0]
Gen 2: Best fitness = 24, Chromosome = [1, 1, 0, 0, 1, 0]
Gen 3: Best fitness = 23, Chromosome = [0, 1, 0, 1, 1, 1]
Gen 4: Best fitness = 30, Chromosome = [0, 1, 1, 0, 1, 0]
Gen 5: Best fitness = 27, Chromosome = [0, 1, 0, 0, 1, 0]
Gen 6: Best fitness = 27, Chromosome = [0, 0, 0, 1, 1, 1]
Gen 7: Best fitness = 25, Chromosome = [0, 0, 1, 1, 0, 0]
Gen 8: Best fitness = 28, Chromosome = [1, 1, 0, 0, 1, 0]
Gen 9: Best fitness = 25, Chromosome = [0, 1, 0, 0, 1, 0]
Gen 10: Best fitness = 23, Chromosome = [1, 1, 0, 0, 0, 0]
Gen 11: Best fitness = 31, Chromosome = [1, 0, 1, 0, 1, 0]
Gen 12: Best fitness = 31, Chromosome = [1, 0, 0, 0, 1, 0]
Gen 13: Best fitness = 31, Chromosome = [1, 1, 1, 0, 1, 0]
Gen 14: Best fitness = 31, Chromosome = [1, 0, 1, 0, 1, 0]
Gen 15: Best fitness = 31, Chromosome = [1, 0, 1, 0, 1, 0]
Gen 16: Best fitness = 31, Chromosome = [1, 0, 1, 0, 1, 0]
Gen 17: Best fitness = 31, Chromosome = [1, 0, 1, 0, 0, 0]
Gen 18: Best fitness = 31, Chromosome = [1, 0, 1, 0, 0, 0]
Gen 19: Best fitness = 31, Chromosome = [1, 1, 1, 0, 1, 0]
Gen 20: Best fitness = 31, Chromosome = [0, 0, 1, 0, 0, 0]

Best Solution:
Chromosome: [1, 0, 1, 0, 1, 0]
Fitness (Value): 31
Weight: 15
Items: [1, 3, 5]

=== Code Execution Successful ===
```

Program 2:
Optimization via Gene Expression Algorithms:
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:

25/8/25

7) <u>Optimization</u> <u>via</u> <u>Gene</u> <u>Expression</u> <u>Algorithm</u>:

→ Define fitness function:
    fitness(x) = sum of squares of x

→ Initialize parameters:
```
    population_size = 20
    num_genes = 5
    gene_min = -5.0
    gene_max = 5.0
    mutation_rate = 0.1
    crossover_rate = 0.8
    generations = 5.
```

→ Initialize population:
```
    for each individual in population_size:
        Create a vector of num_genes
        random values between gene_min
        and gene_max
```

```
-> For generation = 1 to generations do:
    a. Evaluate fitness for all individuals:
        For each individual:
            Calculate fitness (individual)

    b. Find the best individual so far
       and save if improved.

    c. Print generation number, best fitness
       and best solution.

    d. Select parents:
        Use tournament selection based on
                                   fitness
    e. Generate next generation:
        For pairs of parents:
            Perform crossover with
            probability crossover_rate
            Perform mutation on children
            with mutation_rate
            Add children to next generation

    f. Replace population with next generat
                                         -on

-> After all generations:
    Print best solution and its fitness
```

Sra 25/8/25

output.

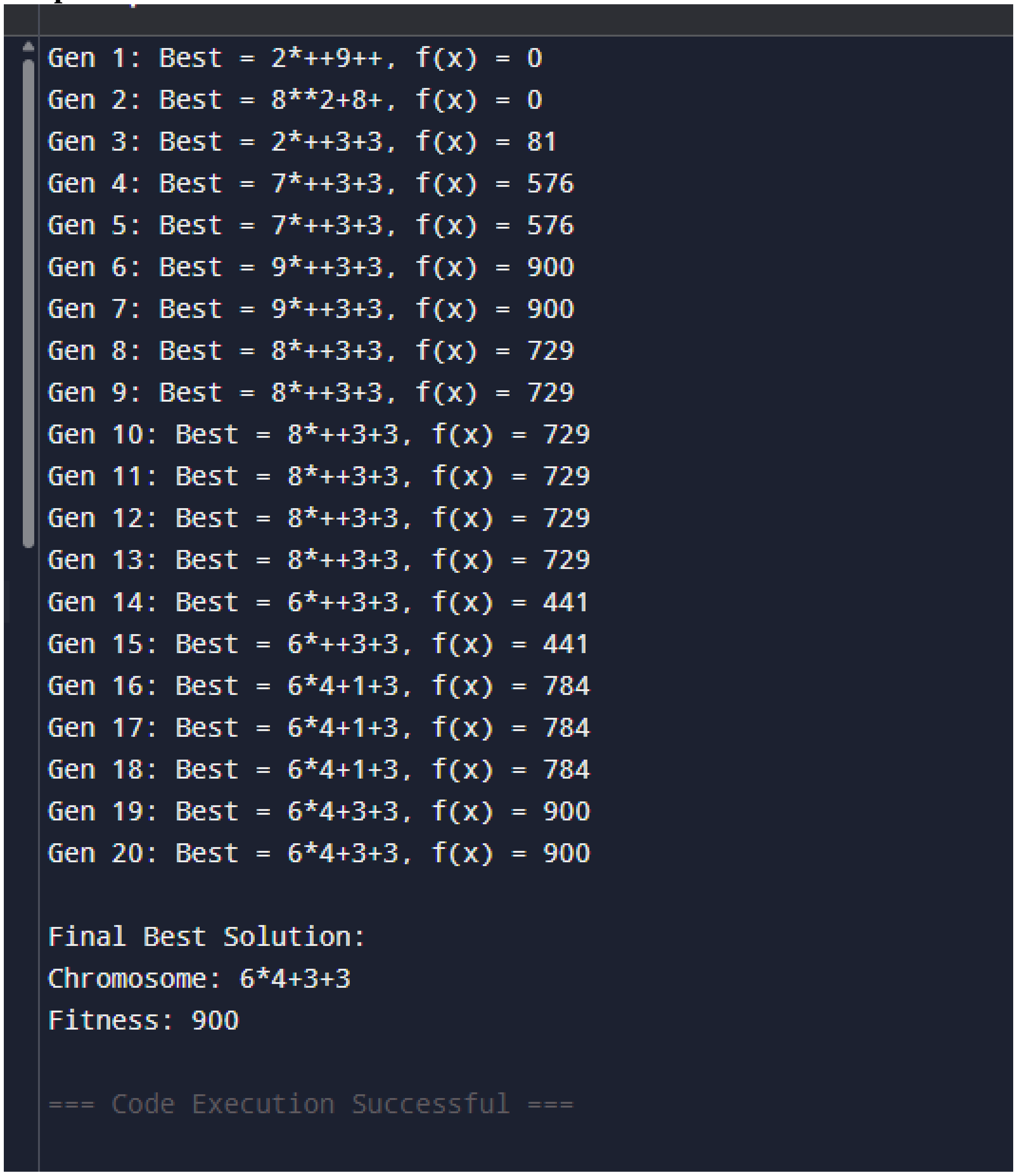
```
Gen   Best Fitness      Best solution (genes)
 1    13.136775         [ 2.3013   2.095  -1.865  -0.34]
 2     3.919895         [-0.4556  -1.64    0.877   0.75]
 3     3.919895         [-0.45    -1.64    0.877   0.75]
 4     3.919895         [-0.45    -1.64    0.877   0.75]
Best solution found: [-0.45 -1.64 0.87 0.75] with fitness 4.238
```

Code:

```
import random
import math

# -------------------------------
# PARAMETERS
# -------------------------------
POP_SIZE = 6
CHROM_LENGTH = 7          # length of genetic sequence
FUNCTIONS = ['+', '-', '*']
TERMINALS = [str(i) for i in range(10)]  # constants 0–9
CROSSOVER_RATE = 0.8
MUTATION_RATE = 0.1
GENERATIONS = 20

# -------------------------------
# HELPER FUNCTIONS
# -------------------------------

def random_gene():
    """Return a random gene (either function or terminal)."""
    if random.random() < 0.4:
        return random.choice(FUNCTIONS)
    return random.choice(TERMINALS)

def create_individual():
    """Generate a random chromosome (sequence)."""
    return [random_gene() for _ in range(CHROM_LENGTH)]

def decode_expression(chromosome):
    """Convert chromosome into a valid arithmetic expression."""
    expr = ""
    for gene in chromosome:
        expr += gene
    return expr

def evaluate(chromosome):
    """Evaluate chromosome by expressing it as integer x, then f(x)=x^2."""
    expr = decode_expression(chromosome)
    try:
        # Evaluate safely
        x_val = int(eval(expr))
    except Exception:
        return 0  # invalid expression
    if x_val < 0 or x_val > 31:  # constrain to problem domain
        return 0
    return x_val**2
```

```
def roulette_wheel_selection(pop, fitnesses):
    """Select one individual using roulette wheel."""
    total_fit = sum(fitnesses)
    if total_fit == 0:
        return random.choice(pop)
    pick = random.uniform(0, total_fit)
    current = 0
    for i, f in enumerate(fitnesses):
        current += f
        if current > pick:
            return pop[i]

def crossover(parent1, parent2):
    """Single point crossover."""
    if random.random() > CROSSOVER_RATE:
        return parent1[:], parent2[:]
    point = random.randint(1, CHROM_LENGTH - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

def mutate(chromosome):
    """Mutate chromosome by flipping a gene."""
    for i in range(len(chromosome)):
        if random.random() < MUTATION_RATE:
            chromosome[i] = random_gene()
    return chromosome

# -------------------------------
# MAIN LOOP
# -------------------------------

population = [create_individual() for _ in range(POP_SIZE)]

for gen in range(GENERATIONS):
    fitnesses = [evaluate(ind) for ind in population]
    best_index = fitnesses.index(max(fitnesses))
    best = population[best_index]
    print(f"Gen {gen+1}: Best = {decode_expression(best)}, f(x) = {max(fitnesses)}")

    # New population
    new_population = []
    while len(new_population) < POP_SIZE:
        p1 = roulette_wheel_selection(population, fitnesses)
        p2 = roulette_wheel_selection(population, fitnesses)
        c1, c2 = crossover(p1, p2)
        c1 = mutate(c1)
        c2 = mutate(c2)
        new_population.extend([c1, c2])
    population = new_population[:POP_SIZE]

# Final result
fitnesses = [evaluate(ind) for ind in population]
best_index = fitnesses.index(max(fitnesses))
best = population[best_index]
print("\nFinal Best Solution:")
print("Chromosome:", decode_expression(best))
print("Fitness:", max(fitnesses))
```

Output:



```
Gen 1: Best = 2*++9++, f(x) = 0
Gen 2: Best = 8**2+8+, f(x) = 0
Gen 3: Best = 2*++3+3, f(x) = 81
Gen 4: Best = 7*++3+3, f(x) = 576
Gen 5: Best = 7*++3+3, f(x) = 576
Gen 6: Best = 9*++3+3, f(x) = 900
Gen 7: Best = 9*++3+3, f(x) = 900
Gen 8: Best = 8*++3+3, f(x) = 729
Gen 9: Best = 8*++3+3, f(x) = 729
Gen 10: Best = 8*++3+3, f(x) = 729
Gen 11: Best = 8*++3+3, f(x) = 729
Gen 12: Best = 8*++3+3, f(x) = 729
Gen 13: Best = 8*++3+3, f(x) = 729
Gen 14: Best = 6*++3+3, f(x) = 441
Gen 15: Best = 6*++3+3, f(x) = 441
Gen 16: Best = 6*4+1+3, f(x) = 784
Gen 17: Best = 6*4+1+3, f(x) = 784
Gen 18: Best = 6*4+1+3, f(x) = 784
Gen 19: Best = 6*4+3+3, f(x) = 900
Gen 20: Best = 6*4+3+3, f(x) = 900

Final Best Solution:
Chromosome: 6*4+3+3
Fitness: 900

=== Code Execution Successful ===
```

Program 3:

Particle Swarm Optimization for Function Optimization:

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. Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:

3) Particle Swarm optimization ~~of~~ for Function optimization:-

```
→ Initialize a swarm of N particles;
  for each particle $i = 1$ to $N$:
    - Randomly initialize position
      $x_i \in [x_{min}, x_{max}]$
    - Randomly initialize velocity
      $v_i \in [-|x_{max} - x_{min}|, |x_{max} - x_{min}|]$
    - Set personal best position
      $p_{best_i} \leftarrow x_i$
    - Evaluate personal best score
      $S_{best_i} \leftarrow f(p_{best_i})$

→ Set global best position:
  - find particle with lowest score $S_{best_i}$
  - Set global best $g_{best} \leftarrow p_{best_i}$ with lowest $S_{best_i}$

→ Repeat for $t = 1$ to $T$ (maximum iterations):
  for each particle $i = 1$ to $N$:
    a. Evaluate current score $S_i \leftarrow f(x_i)$
    b. If $S_i < S_{best_i}$:
       - $p_{best_i} \leftarrow x_i$
       - $S_{best_i} \leftarrow S_i$

    c. If $S_i < f(g_{best})$:
       - $g_{best} \leftarrow x_i$
    d. update velocity:
       - Generate random number $r_1, r_2 \in [0,1]$
       - $v_i \leftarrow w * v_i + c_1 * r_1 * (p_{best_i} - x_i)$
             $+ c_2 * r_2 * (g\_best - x_i)$
    e. update position:
       - $x_i \leftarrow x_i + v_i$
```

f. Apply boundary constraints:
- if $x_i < x_{min}$: $x_i \leftarrow x_{min}$
- if $x_i > x_{max}$: $x_i \leftarrow x_{max}$

→ After T iterations:
- Return g_{best} as best solution
- Return $f_{g\text{-}best}$ as best Score

Output

```
Iteration 0 : Best Score = 14.294383
Iteration 10 : Best score = 0.000176
Iteration 20 : Best Score = 0.000172
Iteration 30 : Best Score = 0.00000
Iteration 40 : Best score = 0.00000
Iteration 50 : Best Score = 0. 00000
```

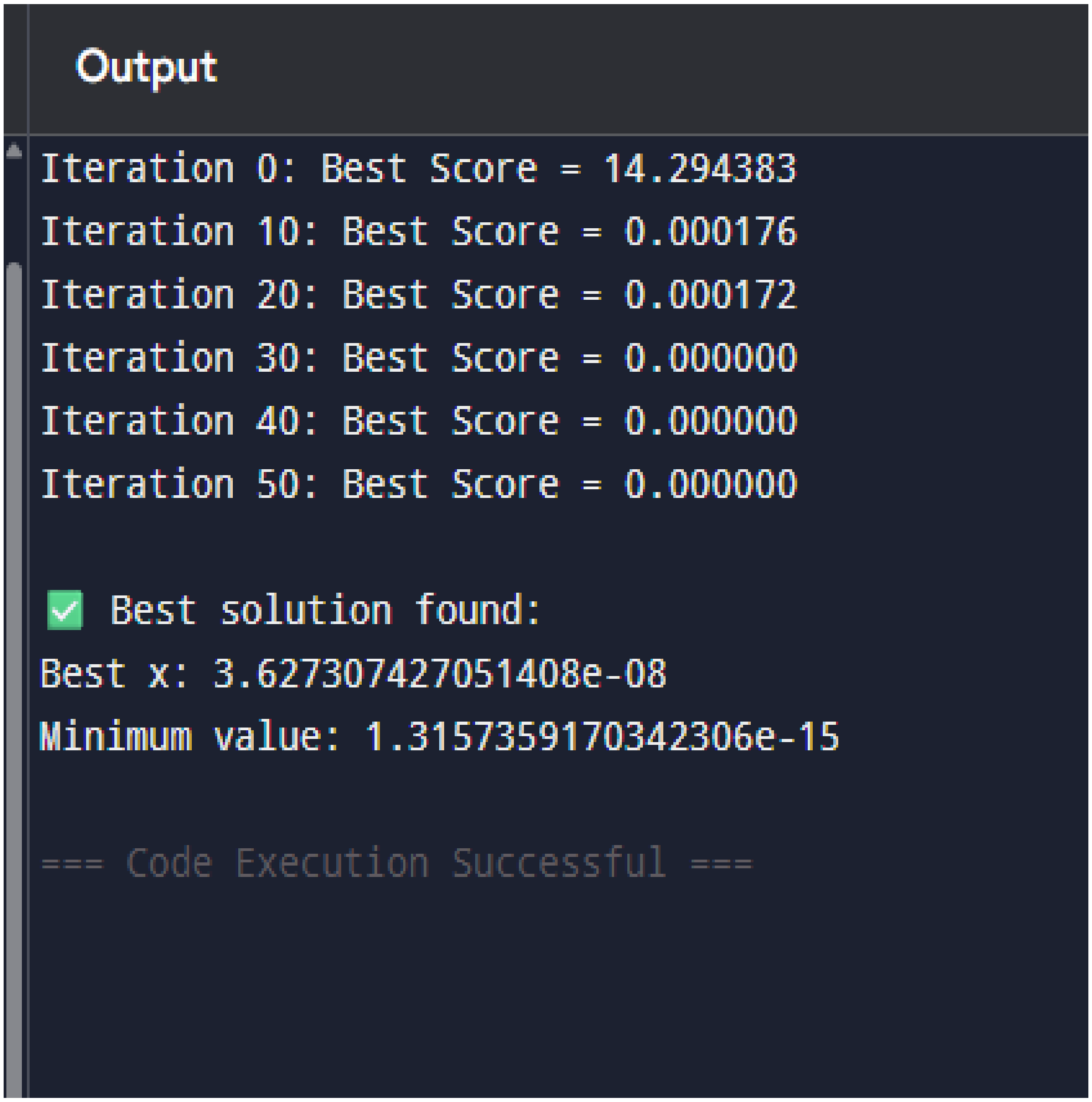
Best solution found:
Best x: 3.627307 [Global best position]
Minimum value: 1.3157359 [Global best score of that function]

[illegible] 1/9/25

Code:

```
import random

# Step 1: Define function (to maximize)
def f(x):
    return (x**2+3*x+4)   # simple quadratic

# Step 2: Parameters
num_particles = 10
iterations = 20
w = 0.4# inertia
c1, c2 = 1.5, 1

# Step 3: Initialize particles
positions = [random.uniform(-10, 10) for _ in range(num_particles)]
velocities = [random.uniform(-1, 1) for _ in range(num_particles)]
pbest = positions[:]
pbest_val = [f(x) for x in positions]
gbest = pbest[pbest_val.index(max(pbest_val))]

# Step 4–6: Iterate
for _ in range(iterations):
    for i in range(num_particles):
        # Update velocity
        velocities[i] = (w*velocities[i]
                         + c1*random.random()*(pbest[i]-positions[i])
                         + c2*random.random()*(gbest-positions[i]))
        # Update position
        positions[i] += velocities[i]

        # Update personal best
        val = f(positions[i])
        if val > pbest_val[i]:
            pbest[i] = positions[i]
            pbest_val[i] = val

    # Update global best
    gbest = pbest[pbest_val.index(max(pbest_val))]

# Step 7: Output
print("Global Best solution:", gbest, "Fitness Value:", f(gbest))
```

Output:



```
Iteration 0: Best Score = 14.294383
Iteration 10: Best Score = 0.000176
Iteration 20: Best Score = 0.000172
Iteration 30: Best Score = 0.000000
Iteration 40: Best Score = 0.000000
Iteration 50: Best Score = 0.000000

✅ Best solution found:
Best x: 3.627307427051408e-08
Minimum value: 1.3157359170342306e-15


=== Code Execution Successful ===
```

Program 4:
Ant Colony Optimization for the Traveling Salesman Problem:
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:

monday

Lab-program : 4

Bafna Gold Date 9/8/24 Page

4) Ant-Colony optimization for Travelling salesman Problem:-

Simple ACO algorithm for TSP:
1. Start : Define cities & calculate distance between every pair of cities.
2. Initialize : Set pheromone levels on all paths to the same initial-value
3. Repeat for many iterations :
→ For each ant:
a) Start from a random city:
b) Build a path by choosing next city based on how strong pheromone trail is & how close city is. Continue until all cities are visited and return to start city.
c) Calculate total length of each ant's path
d) Update pheromones on all path
→ Evaporate some pheromone from all paths
→ Add pheromone to paths used by ants with shorter routes.

4. Keep track of shortest path found so far
5. After all iterations, output the best path and its total distance.

Note: (α) → Pheromone (chemical released from ant): Trail strength / edge weight.
(β) → Heuristic info: Desirability (1/distance) To find shortest path b/w cities.
(ρ) → Evaporation rate: How fast pheromones fades, this occurs when that path is not reinforced by newer ants.

output

Input: City Coordinates
City 0 : (0,0)
City 1 : (2,6)
City 2 : (5,3)
City 3 : (6,7)
City 4 : (8,2)

output : Best Tour Found
→ Tour path (city numbers) :
$2 \rightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 0 \longrightarrow 2$

→ Edges and weights :
$2 \rightarrow 4$: 3.16
$4 \rightarrow 3$: 5.39
$3 \rightarrow 1$: 4.12
$1 \rightarrow 0$: 6.32
$0 \rightarrow 2$: 5.83

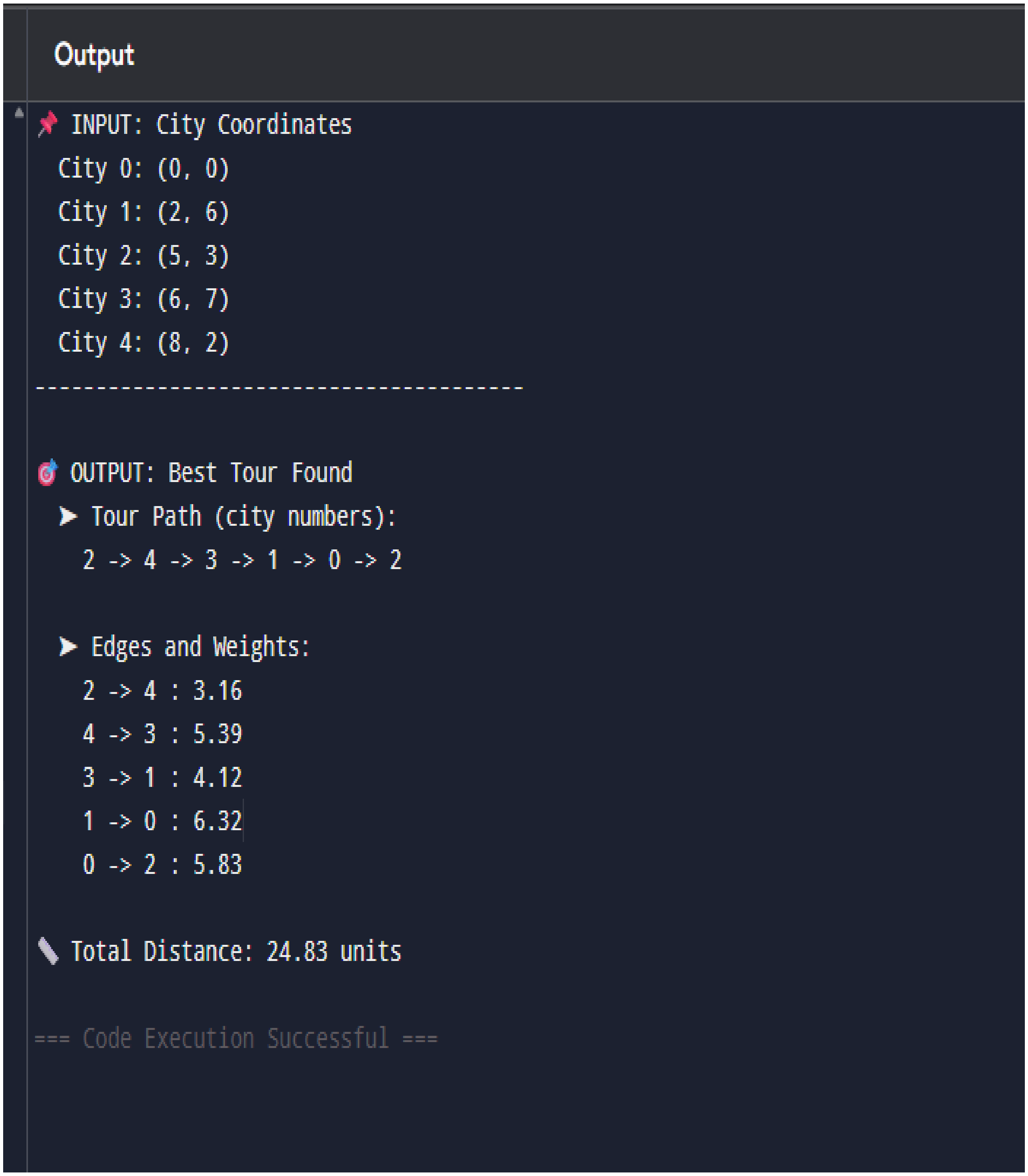
Total Distance : 24.83 units.

[illegible]
8/9/25

Code:

```
import random
import math

# ----------------------------
# Problem Setup (TSP Cities)
# ----------------------------
cities = {
    0: (0, 0),
    1: (1, 5),
    2: (5, 2),
    3: (6, 6),
    4: (8, 3)
}

num_cities = len(cities)

# Distance matrix
distances = [[0] * num_cities for _ in range(num_cities)]
for i in range(num_cities):
    for j in range(num_cities):
        xi, yi = cities[i]
        xj, yj = cities[j]
        distances[i][j] = math.sqrt((xi - xj) ** 2 + (yi - yj) ** 2)

# ----------------------------
# Parameters
# ----------------------------
num_ants = 10
iterations = 50
alpha = 0      # pheromone importance
beta = 5.0       # heuristic importance
rho = 0.5        # evaporation rate
Q = 5        # pheromone deposit factor
initial_pheromone = 1.0

# ----------------------------
# Initialize pheromones
# ----------------------------
pheromone = [[initial_pheromone] * num_cities for _ in range(num_cities)]

# ----------------------------
# Helper Functions
# ----------------------------
def tour_length(tour):
    length = 0
    for i in range(len(tour) - 1):
        length += distances[tour[i]][tour[i + 1]]
    length += distances[tour[-1]][tour[0]]  # return to start
    return length

def select_next_city(current, unvisited):
    probabilities = []
    denom = sum((pheromone[current][j] ** alpha) * ((1 / distances[current][j]) ** beta) for j in
unvisited)
    for j in unvisited:
        prob = (pheromone[current][j] ** alpha) * ((1 / distances[current][j]) ** beta) / denom
        probabilities.append((j, prob))

    # Roulette wheel selection
    r = random.random()
    cumulative = 0
    for city, prob in probabilities:
        cumulative += prob
        if r <= cumulative:
            return city
    return unvisited[-1]

# ----------------------------
# Main ACO Loop
# ----------------------------
best_length = float("inf")
best_tour = None

for it in range(iterations):
    all_tours = []
    all_lengths = []

    for ant in range(num_ants):
        start = random.randint(0, num_cities - 1)
        tour = [start]
        unvisited = list(set(range(num_cities)) - {start})

        while unvisited:
            current = tour[-1]
            next_city = select_next_city(current, unvisited)
            tour.append(next_city)
            unvisited.remove(next_city)

        length = tour_length(tour)
        all_tours.append(tour)
        all_lengths.append(length)

        if length < best_length:
            best_length = length
            best_tour = tour[:]

    # Evaporate pheromones
    for i in range(num_cities):
        for j in range(num_cities):
            pheromone[i][j] *= (1 - rho)

    # Deposit pheromones (each ant contributes)
    for tour, length in zip(all_tours, all_lengths):
        for i in range(len(tour) - 1):
            a, b = tour[i], tour[i + 1]
            pheromone[a][b] += Q / length
            pheromone[b][a] += Q / length
        # close the tour
        pheromone[tour[-1]][tour[0]] += Q / length
        pheromone[tour[0]][tour[-1]] += Q / length

    print(f"Iteration {it+1}: Best Length = {best_length}, Best Tour = {best_tour}")

# ----------------------------
# Final Result
# ----------------------------
print("\nFinal Best Tour:", best_tour)
print("Final Best Length:", best_length)
```

Output:


Output

```
📌 INPUT: City Coordinates
  City 0: (0, 0)
  City 1: (2, 6)
  City 2: (5, 3)
  City 3: (6, 7)
  City 4: (8, 2)
----------------------------------------


🎯 OUTPUT: Best Tour Found
  ➤ Tour Path (city numbers):
    2 -> 4 -> 3 -> 1 -> 0 -> 2

  ➤ Edges and Weights:
    2 -> 4 : 3.16
    4 -> 3 : 5.39
    3 -> 1 : 4.12
    1 -> 0 : 6.32
    0 -> 2 : 5.83

📏 Total Distance: 24.83 units


=== Code Execution Successful ===
```

Program 5:
Cuckoo Search (CS):
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:

Monday

lab: program - 5

Bafna Gold
Date: 15/9/25

5) Algorithm: Knapsack Problem using Cuckoo Search

1. Input:
→ No of items, their values & weights
→ Knapsack capacity
→ Algo parameters: no of nests n, probability of discovery pa, no of iterations max_iter

2. Initialize Nests:
Create n random solutions [Eg:- [1 0 1 0], means pick items 1 and 3].

3. Evaluate fitness:
for each solution
→ Calculate total wt & total value
→ If wt ≤ capacity → fitness = total value
→ Else → fitness = 0 (invalid)

4. Find Best Nest: Select the solution with highest fitness.

5. Generate New Solutions (via Levy flights)
for each nest:
→ Modify solution slightly using levy flight
→ Evaluate new fitness
→ If new solution is better → replace old one

6. Abandon Worst Nests:
With probability pa, replace some worst solutions with new random ones

7) Update Best
→ check if the new solutions are better than global best solution.
→ If yes → update best solution.
→ Continue steps 5-7 until max_iter are done

8) output
Print best solution found that is selected items, total value, total weight

output

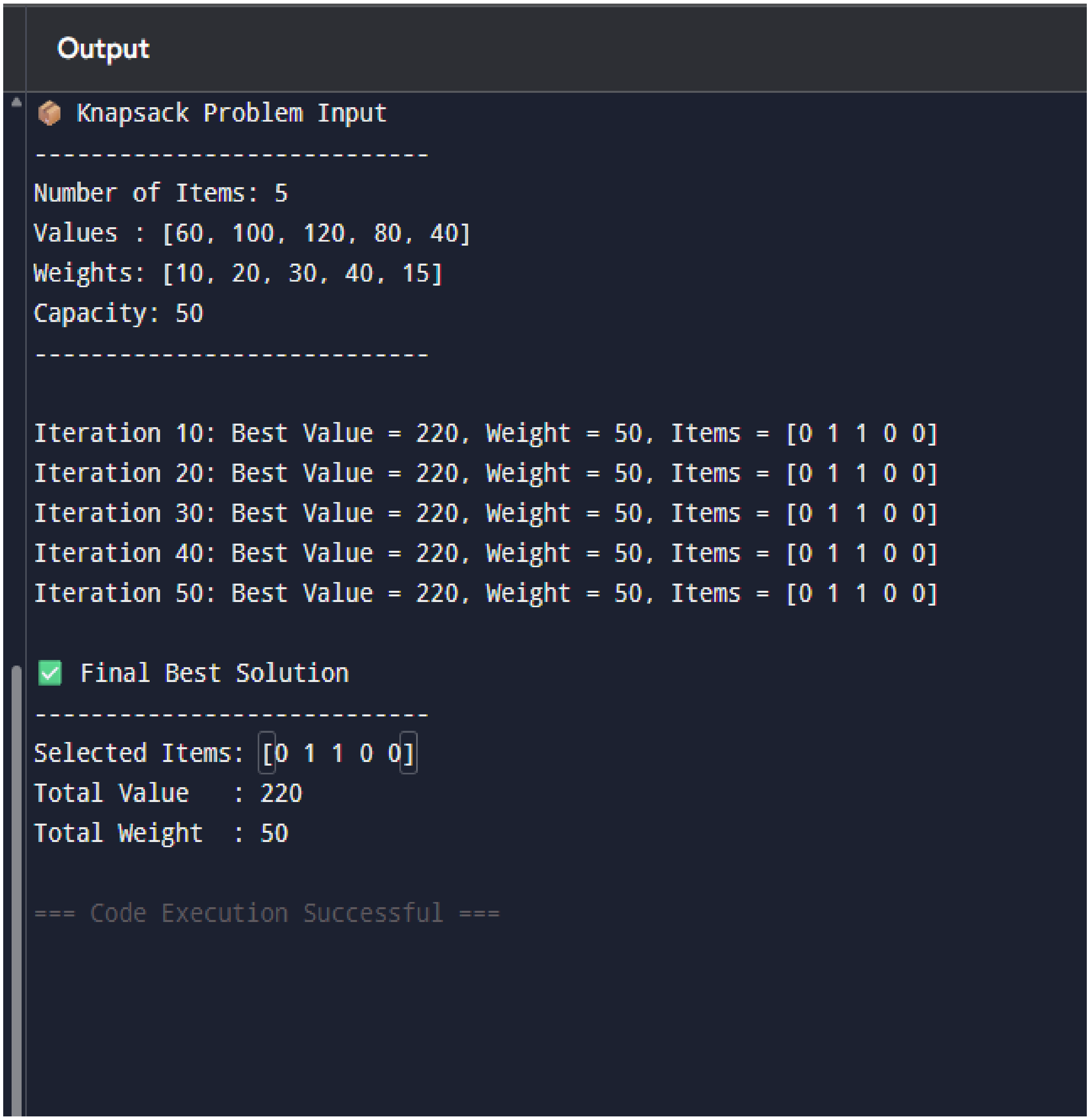
```
Knapsack Problem Input
Number of Items : 5
Values : [60, 100, 120, 80, 40]
weights : [10, 20, 30, 40, 15]
Capacity : 50

Iteration 10 : Best value = 220, weight = 50,
Items = [0 1 1 0 0]
Iteration 20 : Best value = 220, weight = 50, Items = [0 1 1 0 0]
Iteration 30 : Best value = 220, weight = 50, Items = [0 1 1 0 0]
Iteration 40 : Best value = 220, weight = 50, Items = [0 1 1 0 0]
Iteration 50 : Best value = 220, weight = 50, Items = [0 1 1 0 0]
```

Summary:
Cuckoo Bird lay their eggs in nearby suitable nest of someother bird.
Levy flight → It is a random walk approach used by CB : to find suitable nest for laying egg
Discovery Prob → Host bird can find CB egg & throw away.

Code:

```
import numpy as np
import math

# ------------------------------
# Objective Function (minimize)
# ------------------------------
def objective_function(x):
    return np.sum(x**2)  # Sphere function example

# ------------------------------
# Lévy flight step
# ------------------------------
def levy_flight(Lambda, size):
    sigma = (math.gamma(1+Lambda) * math.sin(math.pi*Lambda/2) /
             (math.gamma((1+Lambda)/2) * Lambda * 2**((Lambda-1)/2)))**(1/Lambda)
    u = np.random.normal(0, sigma, size)
    v = np.random.normal(0, 1, size)
    return u / (np.abs(v)**(1/Lambda))

# ------------------------------
# Cuckoo Search Algorithm
# ------------------------------
def cuckoo_search(n=10, dim=2, lb=-10, ub=10, pa=0.25, max_iter=50):
    # Initialize nests randomly
    nests = np.random.uniform(lb, ub, (n, dim))
    fitness = np.array([objective_function(x) for x in nests])
    best_nest = nests[np.argmin(fitness)].copy()
    best_fitness = np.min(fitness)

    for t in range(max_iter):
        # Generate new solutions via Lévy flights
        new_nests = nests + levy_flight(1.5, (n, dim)) * (nests - best_nest)
        new_nests = np.clip(new_nests, lb, ub)  # keep within bounds
        new_fitness = np.array([objective_function(x) for x in new_nests])

        # Replace nests if better
        for i in range(n):
            if new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]

        # Abandon some nests (with probability pa)
        abandon = np.random.rand(n) < pa
        nests[abandon] = np.random.uniform(lb, ub, (np.sum(abandon), dim))
        fitness[abandon] = [objective_function(x) for x in nests[abandon]]
        # Update global best
        if np.min(fitness) < best_fitness:
            best_fitness = np.min(fitness)
            best_nest = nests[np.argmin(fitness)].copy()

        print(f"Iteration {t+1}: Best Fitness = {best_fitness:.6f}")

    return best_nest, best_fitness

# -------------------------------
# Run the algorithm
# -------------------------------
best_solution, best_value = cuckoo_search()
print("\nBest Solution:", best_solution)
print("Best Value:", best_value)
```

Output:


Output

```
📦 Knapsack Problem Input
----------------------------
Number of Items: 5
Values : [60, 100, 120, 80, 40]
Weights: [10, 20, 30, 40, 15]
Capacity: 50
----------------------------

Iteration 10: Best Value = 220, Weight = 50, Items = [0 1 1 0 0]
Iteration 20: Best Value = 220, Weight = 50, Items = [0 1 1 0 0]
Iteration 30: Best Value = 220, Weight = 50, Items = [0 1 1 0 0]
Iteration 40: Best Value = 220, Weight = 50, Items = [0 1 1 0 0]
Iteration 50: Best Value = 220, Weight = 50, Items = [0 1 1 0 0]

✅ Final Best Solution
----------------------------
Selected Items: [0 1 1 0 0]
Total Value   : 220
Total Weight  : 50


=== Code Execution Successful ===
```

Program 6:
Grey Wolf Optimizer (GWO):
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 c

Algorithm:

Monday

Lab-program: 6

29/9/2025

6) Algorithm: Grey wolf optimizer (GWO) on Job Scheduling Problem:-

Step-1: Initialize wolves.
→ For each wolf (solution) generate a random list of n numbers which decide the job order (lower number = higher priority)

Step-2: Convert to Job Sequence
→ For each wolf, sort its number using argsort() which gives job sequence.
→ Calculate total completion time (fitness) for the sequence.

Step-3: Identify Alpha, Beta, Delta, & Omega
→ Alpha: best (lowest time)
→ Beta: Second best
→ Delta: third best
→ Omega: Remaining solutions.

Step-4: update wolves (Repeat for MaxIter times)
for each iteration:
→ Update control parameter
$a = 2 - (2 \times \text{current iteration} / \text{MaxIter})$
→ For each wolf & each value update its alpha, Beta, Delta & omega wolves co-efficient
→ clip the values to stay b/w 0 and 1
→ Convert new Values to job sequence using argsort()
→ Calculate fitness (completion time)
→ If this new wolf is better than Alpha, Beta or Delta update them.

Step 5: Return Result
→ After all iterations return Alpha wolf's job sequence (best order) & its total completion time (minimum)

Output

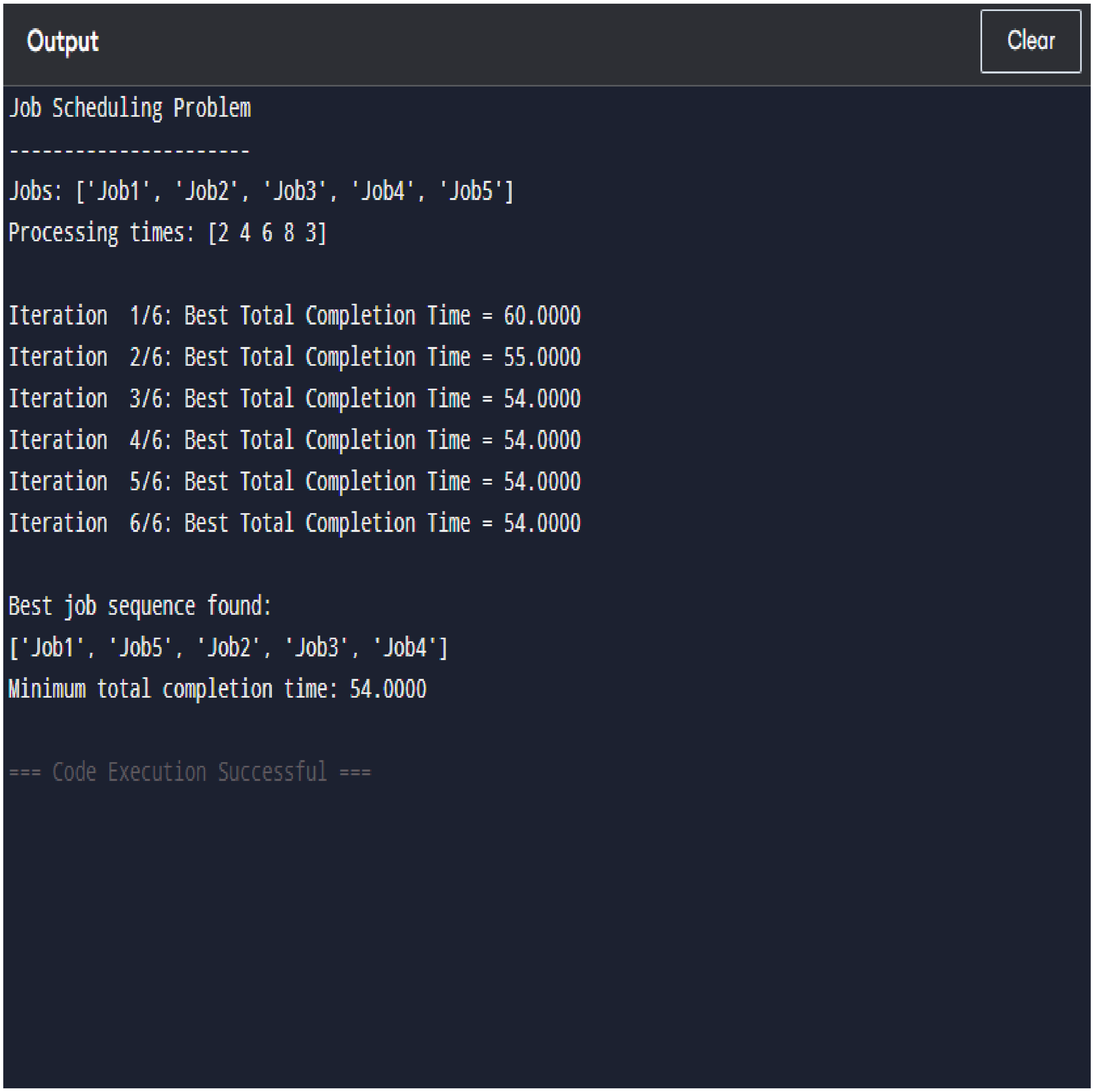
```
Job Scheduling Problem
Jobs : ['Job1', 'Job2', 'Job3', 'Job4', 'Job5']
Processing times : [2 4 6 8 3]

Iteration 1/6 : Total Completion Time = 60.0000
Iteration 2/6 : Total Completion Time = 55.0000
Iteration 3/6 : Total Completion Time = 54.0000
Iteration 4/6 : Total Completion Time = 54.0000
Iteration 5/6 : Total Completion Time = 54.0000
Iteration 6/6 : Total Completion Time = 54.0000

Best job sequence found :
['Job1', 'Job 5', 'Job 2', 'Job 3', 'Job4']
Minimum total completion time : 54.0000
```

[illegible] 29/9/25

Code:

```
import numpy as np

# Objective Function (Sphere function)
def objective_function(x):
    return np.sum(x**2)

# Grey Wolf Optimizer
def grey_wolf_optimizer(obj_func, dim=2, search_agents=10, max_iter=50, lb=-10, ub=10):
    # Initialize wolf positions randomly
    wolves = np.random.uniform(lb, ub, (search_agents, dim))
    fitness = np.array([obj_func(w) for w in wolves])

    # Identify alpha, beta, delta wolves
    alpha, beta, delta = np.zeros(dim), np.zeros(dim), np.zeros(dim)
    alpha_score, beta_score, delta_score = float("inf"), float("inf"), float("inf")

    # Find initial alpha, beta, delta
    for i in range(search_agents):
        if fitness[i] < alpha_score:
            alpha_score = fitness[i]
            alpha = wolves[i].copy()
        elif fitness[i] < beta_score:
            beta_score = fitness[i]
            beta = wolves[i].copy()
        elif fitness[i] < delta_score:
            delta_score = fitness[i]
            delta = wolves[i].copy()

    # Main loop
    for t in range(max_iter):
        a = 2 - t * (2 / max_iter)  # linearly decreases from 2 to 0

        for i in range(search_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] = (X1 + X2 + X3) / 3  # update wolf position

            # Boundaries
            wolves[i] = np.clip(wolves[i], lb, ub)

            # Fitness evaluation
            score = obj_func(wolves[i])

            if score < alpha_score:
                delta_score, delta = beta_score, beta.copy()
                beta_score, beta = alpha_score, alpha.copy()
                alpha_score, alpha = score, wolves[i].copy()
            elif score < beta_score:
                delta_score, delta = beta_score, beta.copy()
                beta_score, beta = score, wolves[i].copy()
            elif score < delta_score:
                delta_score, delta = score, wolves[i].copy()

        print(f"Iteration {t+1}: Best Fitness = {alpha_score:.6f}")

    return alpha, alpha_score

# Run GWO
best_position, best_value = grey_wolf_optimizer(objective_function, dim=2, search_agents=15,
max_iter=50)
print("\nBest Position:", best_position)
print("Best Fitness:", best_value)
```

Output:



```
Job Scheduling Problem
----------------------
Jobs: ['Job1', 'Job2', 'Job3', 'Job4', 'Job5']
Processing times: [2 4 6 8 3]

Iteration  1/6: Best Total Completion Time = 60.0000
Iteration  2/6: Best Total Completion Time = 55.0000
Iteration  3/6: Best Total Completion Time = 54.0000
Iteration  4/6: Best Total Completion Time = 54.0000
Iteration  5/6: Best Total Completion Time = 54.0000
Iteration  6/6: Best Total Completion Time = 54.0000

Best job sequence found:
['Job1', 'Job5', 'Job2', 'Job3', 'Job4']
Minimum total completion time: 54.0000

=== Code Execution Successful ===
```

Problem 7:
Parallel Cellular Algorithms and Programs:
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:

Monday

Lab-program-7   13/10/25

7) Parallel Cellular Algorithm for Traffic and Crowd Simulation:-

```
a) Initialize GRID with Cars and Pedestrians
b) best_blocked ← ∞
   Skip ← false

c) for step = 1 to N do
      blocked ← count blocked agents:
         for each car at (r,c):
               blocked ++ if GRID[r][(c+1) mod w] ≠ empty
         for each pedestrians at (r,c):
               blocked ++ if GRID[(r-1) mod H][c] ≠ empty
      If blocked < best_blocked then save
      GRID as best_grid
      Print GRID and blocked
      new_GRID ← empty

      for each CAR:
         if next cell empty move right
         else stay

      for each Pedestrian:
         if next cell empty move up
         else stay
      GRID ← NEW_GRID
   End for

d) Print best_grid and best blocked
```

Output:- Step 1 | Blocked Agents : 3

```
.  .  .  .  .
C  C  .  C  .
.  .  .  C  C
.  .  P  P  .
```

Step 2 | Blocked Agents : 2.

```
.  .  .  .  .
C  .  C  .  C
.  .  .  .  .
C  .  P  C  .
.  .  .  P  .
```

Step 3 | Blocked Agents : 2

```
.  .  .  .  .
C  .  C  .  C
C  .  P  C  .
.  .  .  P  .
```

Step 4 | Blocked Agents : 1

```
.  .  .  .  .
.  C  .  C  C
.  .  P  .  .
.  C  .  .  C
.  .  .  P  .
```

Final Best Result with Minimum Blocked Agents: 1

```
.  .  .  .  .
.  C  P  C  C
.  .  P  .  .
.  C  .  .  C
.  .  .  P  .
```

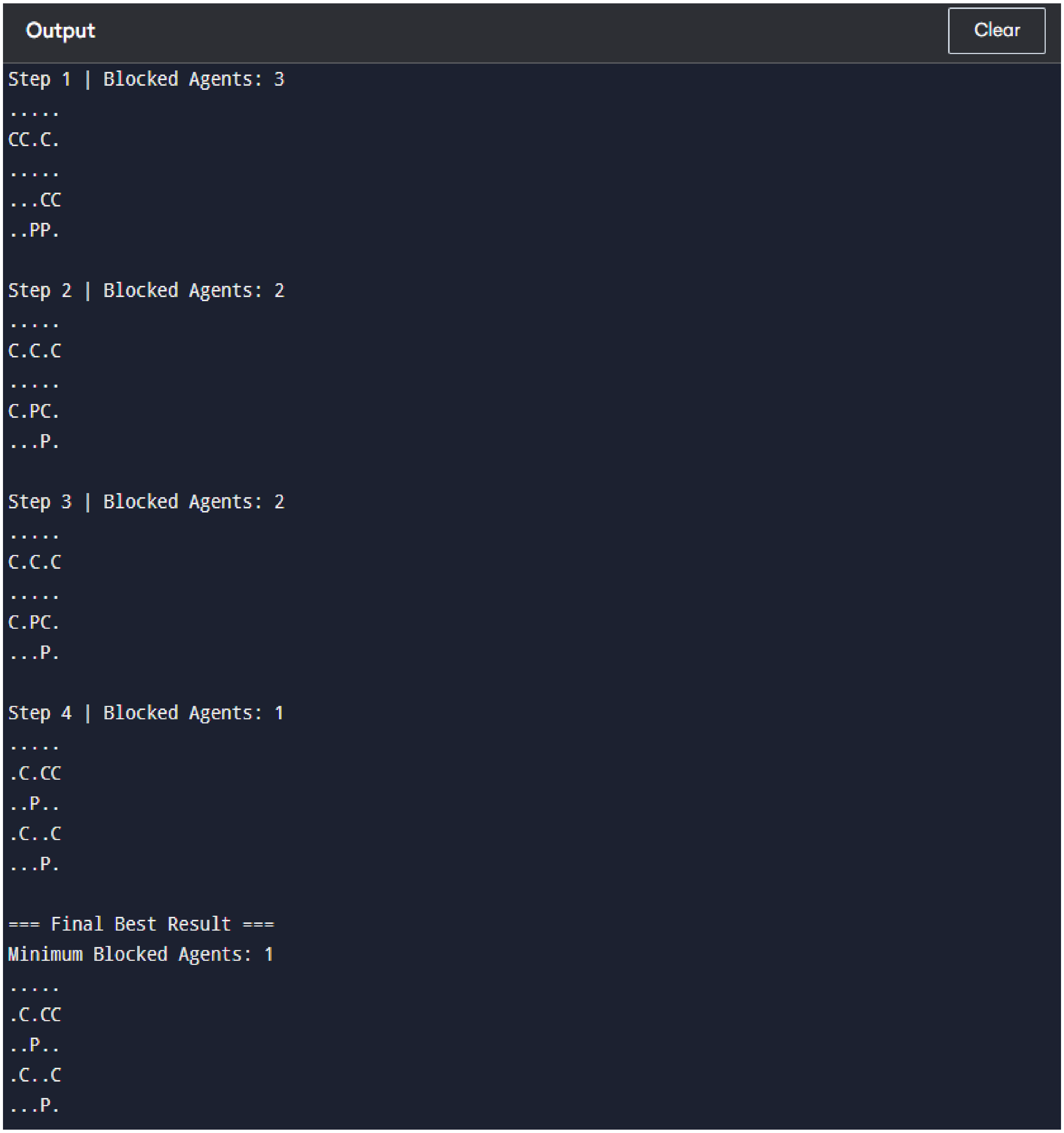
[illegible]
13/10/25

Code:

```
import numpy as np

# ----------------------------
# Cellular Automata Parameters
# ----------------------------
n = 8  # number of cells in CA
rule_number = 30  # Wolfram rule 30

# ----------------------------
# Apply CA rule
# ----------------------------
def apply_rule(left, center, right, rule):
    index = (left << 2) | (center << 1) | right
    return (rule >> index) & 1

# ----------------------------
# Generate CA key stream
# ----------------------------
def generate_key(seed, rule, length):
    state = seed.copy()
    key_stream = []
    for _ in range(length):
        key_stream.append(state[-1])  # output last cell
        new_state = np.zeros_like(state)
        for i in range(len(state)):
            left = state[i-1] if i>0 else state[-1]
            center = state[i]
            right = state[i+1] if i<len(state)-1 else state[0]
            new_state[i] = apply_rule(left, center, right, rule)
        state = new_state
    return np.array(key_stream)

# ----------------------------
# XOR for encryption/decryption
# ----------------------------
def xor_bits(data_bits, key_bits):
    return np.array([d ^ k for d, k in zip(data_bits, key_bits)])

# ----------------------------
# Convert string to bits and back
# ----------------------------
def string_to_bits(s):
    bits = []
    for char in s:
        bits.extend([int(b) for b in format(ord(char), '08b')])
    return bits
```

```
def bits_to_string(bits):
    chars = []
    for i in range(0, len(bits), 8):
        byte = bits[i:i+8]
        chars.append(chr(int(''.join(map(str, byte)), 2)))
    return ''.join(chars)

# ----------------------------
# Main
# ----------------------------
plaintext_str = "HELLO"
plaintext_bits = string_to_bits(plaintext_str)
print("Plaintext:", plaintext_str)
print("Plaintext bits:", plaintext_bits)

# Random CA seed of 8 bits
seed = np.random.randint(0, 2, n)
print("Initial CA Seed:   ", seed.tolist())

# Repeat key stream to match plaintext length
key_stream = np.tile(generate_key(seed, rule_number, n), len(plaintext_bits)//n + 1)[:len(plaintext_bits)]
print("Generated Key Bits:", key_stream.tolist())

# Encrypt
ciphertext_bits = xor_bits(plaintext_bits, key_stream)
print("Ciphertext bits:   ", ciphertext_bits.tolist())

# Decrypt
decrypted_bits = xor_bits(ciphertext_bits, key_stream)
decrypted_str = bits_to_string(decrypted_bits)
print("Decrypted bits:    ", decrypted_bits.tolist())
print("Decrypted Text:    ", decrypted_str)
```

Output:



```
Step 1 | Blocked Agents: 3
.....
CC.C.
.....
...CC
..PP.

Step 2 | Blocked Agents: 2
.....
C.C.C
.....
C.PC.
...P.

Step 3 | Blocked Agents: 2
.....
C.C.C
.....
C.PC.
...P.

Step 4 | Blocked Agents: 1
.....
.C.CC
..P..
.C..C
...P.

=== Final Best Result ===
Minimum Blocked Agents: 1
.....
.C.CC
..P..
.C..C
...P.
```