VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Hitha Harish (1BM23CS115)

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 **Hitha Harish (1BM23CS115) ,** 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/HithaHarishCS/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:

18/8/25

1) GENETIC ALGORITHM

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

Eg: $x \to 0-31$

2) Calculate the fitness

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	

each value / sum

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

$$\text{Expected OP} = \frac{f(x)}{\text{Avg } \Sigma f(x)} \qquad \text{Eg: } \frac{144}{288.75} = 0.49$$

3) Selecting Meeting Pool.

String No.	Meeting Pool	Cross 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	11011	27	729
4	10011	3	10001	17	289

String No.	cross point change bit after crossover point	fitness $f(x)=x^2$
sum		1763
avg		440.75
max		729

4) Cross over: cross over point is chosen randomly

5) Mutation: -

String No	offspring after crossover	Mutation chromosome	offspring after mutation	x value	fitness
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	728
4	10001	00101	10100	20	400
sum					2546
Avg					636.5
Max					841

NOTE: Chromosome: Set of binary number.
Mutation: change in genetics of offspring

~~OUTPUT:~~
PSEUDOCODE:

Function fitness (x)
  Return x * x

Function decode (chromosome):
  Convert the binary list to decimal number
  Return decimal_value:

Function create_population ()
  population = []
  for i = 1 to 10:

chromosome = random list of 5 bits (0 or 1)
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_lists):
Use roulette wheel selection based on fitness value.
Return selected_parents.

Function crossover (parent1, parent2):
If random < 0.7:
Choose a random crossover point
child1 = first part of parent1 + second part of parent2.
child2 = first part of parent2 + second part of parent1
Else:
child1 = copy of parent1
child2 = copy of parent2.
Return child1, child2

Function mutate (chromosome):
For each bit in chromosome:
if random < 0.1:
flip the bit (0 becomes 1, 1 becomes 0).
return chromosome

Function genetic_algorithm ():
population = create_population ()
best_chromosome = None;
best_fitness = -infinity

For generation = 1 to 10
fitness_list = evaluate_population (pop-ulation)

Find chromosome with highest fitness
If this fitness > best_fitness
best_chromosome = that chromosome
best_fitness = its fitness.

Print generation number, bestx & 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 ()

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) GENE EXPRESSION ALGORITHM.

1. Define fitness function:
   fitness(x) = sum of squares of x's components

2. 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 = 30

3. Initialize population:
   For each individual in population_size:
   create a vector of num_genes random values between gene_min & gene_max.

4. 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 & save if improved.

   c. Print generation number, best fitness & 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 generation

-> After all generations:
Print best solution & its fitness

OUTPUTS :

7)

Gen	Best Fitness	Best Solution (genes)
1	7.626874	[-2.006 0.0863 -1.8552 -0.3412]
2	4.239207	[-0.4556 -1.6406 0.8779 0.7545]
3	4.239207	[-0.4556 -1.6406 0.8779 0.7545]
4	4.239207	[-0.4556 -1.6406 0.8779 0.7545]

1) 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

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:

Lab

URBAN EDGE 1, 9, 25

(2) PARTICLE SWARM OPTIMISATION ALGORITHM

ALGORITHM.

1. Initialise Parameters
- Define the objective function $f(\vec{x})$ to minimise or maximise
- Set:
- Number of particles : N
- Number of dimensions : D
- Maximum ~~number~~ of iterations: T
- Inertia weight : w
- cognitive coefficient : c_1
- social coefficient : c_2.

2. Initialise Each Particle.
For each particle $i \in \{1, 2 \dots N\}$:
- Randomly initialise:
- Position vector $\vec{x_i} \in \mathbb{R}^D$
- Velocity vector $\vec{v_i} \in \mathbb{R}^D$
- Evaluate the fitness $f(\vec{x_i})$
- set personal best:
- $\vec{p_i} \leftarrow \vec{x_i}$
- $f(\vec{p_i}) \leftarrow f(\vec{x_i})$

3. Initialise Global Best
- Find the best fitness among all particles:
$\vec{g} \leftarrow \vec{p_j}$ such that $f(\vec{p_j}) = \min_i f(\vec{p_i})$

4. Repeat for each iteration (until T or convergence)
for $t = 1$ to T:
for each particle $i \in \{1, 2 \dots N\}$
a. Update Velocity

$\vec{v_i} \leftarrow w \cdot \vec{v_i} + c_1 \cdot r_1 \cdot (\vec{p_i} - \vec{x_i}) + c_2 \cdot r_2 \cdot (\vec{g} - \vec{x_i})$

- r_1, r_2 are random vectors in $[0,1]^D$

b. Update Position

$\vec{x_i} \leftarrow \vec{x_i} + \vec{v_i}$

c. Evaluate Fitness.

Compute $f(\vec{x_i})$

d. Update Personal Best

If $f(\vec{x_i}) < f(\vec{p_i})$:

$\vec{p_i} \leftarrow \vec{x_i}$

$f(\vec{p_i}) \leftarrow f(\vec{x_i})$

e. Update Global Best

If $f(\vec{p_i}) < f(\vec{g})$

- $\vec{g} \leftarrow \vec{p_i}$

5. After Final Iteration

- Return global best position $\vec{g}$
- Return global best fitness $f(\vec{g})$.

OUTPUT:

Iteration 1 | Best Fitness : 3.755997 | Best Position :
[-1.51561954 1.20784691]

Iteration 2 | Best Fitness : 2.411749 | Best Position :
[-1.16293536 1.02923795]

Iteration 3 | Best Fitness : 0.495682 | Best Position :
[-0.57128111 0.41148507]

Iteration 4 | Best Fitness : 0.209679 | Best Position :
[0.01041056 0.457789]

Iteration 5 | Best Fitness : 0.22462 | Best Position :
[0.0223314 0.1482007]

Iteration	Best Fitness	Best Position
6	0.022462	[0.0223314 0.14820007]
7	0.022462	[0.0223314 0.14820007]
8	0.022462	[0.0223314 0.14820007]
9	0.003205	[0.051932 0.022533]
10	0.001443	[0.011243 0.036279]
11	0.001443	[0.011243 0.036279]
12	0.001443	[0.11243 0.036279]
13	0.000038	[-0.001364 -0.00601]
14	0.000038	[-0.001364 -0.00601]
15	0.000038	[-0.001364 -0.00601]

Best ~~Optimizato~~ Position Found : [-0.00234074 -0.00064012]

Best Fitness Achieved : 0.000006.

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:

Output

```
Global Best solution: 26.806206878454258 Fitness Value: 802.9913478458512

=== 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:

LAB - 7 8|9|25

URBAN EDGE

ANT COLONY OPTIMISATION

ALGORITHM.

1) Initialise :
- Number of ants, cities, pheromone levels, etc.
- Distance between each city

2) Repeat for a number of iterations :
a) Each ant builds a solution (a complete path through the cities) using :
- Pheromone trial (learned knowledge)
- Heuristic information (Eg: inverse of distance)
- Transition probability.

b) Evaluate the solutions (calculate total path length).

c) Update pheromones:
- Evaporate some pheromone
- Deposite pheromone proportional to quality of solution

3) Return the best solution found.

OUTPUT:

Best path found :

City0 > City2 > City 3 > City4 > City1 > Ci

Total distance of the path : 9.00

INPUT :

dist. matrix = $\begin{bmatrix} np.inf & 2 & 2 & 5 & 7 \\ 2 & np.inf & 4 & 8 & 2 \\ 2 & 4 & np.inf & 1 & 3 \\ 5 & 8 & 1 & np.inf & 2 \\ 7 & 2 & 3 & 2 & np.inf \end{bmatrix}$

n_ants = 10
n_best = 3
n_iterations = 100
decay = 0.1
alpha = 1
beta = 2.

Sp.9
8/9/24

0

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:

```
Iteration 6: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 7: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 8: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 9: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 10: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 11: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 12: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 13: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 14: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 15: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 16: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 17: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 18: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 19: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 20: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 21: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 22: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 23: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 24: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 25: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 26: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 27: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 28: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 29: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 30: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 31: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 32: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 33: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 34: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 35: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 36: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 37: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 38: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 39: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 40: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 41: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 42: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 43: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 44: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 45: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 46: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 47: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 48: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 49: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]
Iteration 50: Best Length = 22.35103276995244, Best Tour = [3, 4, 2, 0, 1]

Final Best Tour: [3, 4, 2, 0, 1]
Final Best Length: 22.35103276995244

=== 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:

LAB-5 15/9/25

CUCKOO SEARCH ALGORITHM.

~~algorithm CuckooSearch():~~

~~# INPUT~~

~~# n = initial population size~~

~~# P_a = fraction of worse~~

ALGORITHM.

INPUT:

processing_times[]: A list of processing times of n

n_nests: Number of nests (candidate solution)

pa: Probability of abandoning a nest.

n_iterations: Number of iterations for the algorithm.

1) Initialise nests:

- Create a population of n_nests nests (candidate solutions), each representing a permutation of job orders.
- Each nest is initialised as a continuous vector with random values between 0 & 1 to represent the job order.

2) Evaluate fitness:

- For each nest (solution), calculate the total completion time for the job sequence defined by the permutation

3) Set best solution

- Identify the nest with the best (minimum fitness value, & set it as the best nest

4) Main loop (for each iteration):
- Repeat steps 5-7 for n-iteration times.

5) Generate new nests using Levy flight:
- For each nest i:
  1) Calculate a Lévy flight step based on a random walk.
  2) Update the current nest's solution by adding a step ~~the~~ size ~~valid~~ calculated ~~[0, 1]~~ using ~~the~~ Levy continuous vector flight formula.
  3) Apply bounds to ensure the new solution is within the valid range [0, 1] (since it's a continuous vector).
  4) Decode the updated solution (continuous vector) into a job sequence (permutation) using sorting
  5) Calculate the total completion time for the new solution

6) Greedy replacement.
- If the new solution's fitness (total completion time) is better than the current nest's fitness, relpace the current nest with the new solution & update its fitness.
- If the new solution is better than the global best solution (best_nest), update (best_nest).

7) ~~Upd~~ Abandon worse nests

- Identify the pa * n nests worst nests
- Replace worst nests with random solutions
- Recalculate the fitness of the new nests

8. Output the best solution.
best_nest.

INPUT :

processing times = [3 , 1 , 7 , 5 , 2]

OUTPUT :

Iteration 1 : Best total completion time : 40
Iteration 100 : " : 39
Iteration 200 : " : 39
Iteration 300 : " : 39
Iteration 400 : " : 39
Iteration 500 : " : 39

Best job order (0-indexed) : [1 4 0 3 2]
Best total completion time : 39.

22/9/25

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

```
Iteration 6: Best Fitness = 3.194164
Iteration 7: Best Fitness = 3.194164
Iteration 8: Best Fitness = 3.194164
Iteration 9: Best Fitness = 1.983263
Iteration 10: Best Fitness = 1.983263
Iteration 11: Best Fitness = 0.816409
Iteration 12: Best Fitness = 0.735204
Iteration 13: Best Fitness = 0.735204
Iteration 14: Best Fitness = 0.735204
Iteration 15: Best Fitness = 0.735204
Iteration 16: Best Fitness = 0.310402
Iteration 17: Best Fitness = 0.310402
Iteration 18: Best Fitness = 0.310402
Iteration 19: Best Fitness = 0.310402
Iteration 20: Best Fitness = 0.299307
Iteration 21: Best Fitness = 0.251654
Iteration 22: Best Fitness = 0.251654
Iteration 23: Best Fitness = 0.206784
Iteration 24: Best Fitness = 0.206784
Iteration 25: Best Fitness = 0.206784
Iteration 26: Best Fitness = 0.206784
Iteration 27: Best Fitness = 0.206784
Iteration 28: Best Fitness = 0.206784
Iteration 29: Best Fitness = 0.206784
Iteration 30: Best Fitness = 0.075533
Iteration 31: Best Fitness = 0.075533
Iteration 32: Best Fitness = 0.075533
Iteration 33: Best Fitness = 0.075533
Iteration 34: Best Fitness = 0.075533
Iteration 35: Best Fitness = 0.075533
Iteration 36: Best Fitness = 0.075533
Iteration 37: Best Fitness = 0.075533
Iteration 38: Best Fitness = 0.075533
Iteration 39: Best Fitness = 0.075533
Iteration 40: Best Fitness = 0.075533
Iteration 41: Best Fitness = 0.075533
Iteration 42: Best Fitness = 0.075533
Iteration 43: Best Fitness = 0.075533
Iteration 44: Best Fitness = 0.075533
Iteration 45: Best Fitness = 0.017961
Iteration 46: Best Fitness = 0.017961
Iteration 47: Best Fitness = 0.017961
Iteration 48: Best Fitness = 0.017961
Iteration 49: Best Fitness = 0.017961
Iteration 50: Best Fitness = 0.017961

Best Solution: [-0.13401788 -0.00026016]
Best Value: 0.017960859555768195

=== 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:

29/9/25

GREY WOLF OPTIMISER

ALGORITHM.

Input : Cities with distance matrix
Output : Best tour (shortest path)

Initialise population of wolves (candidate tours)
Evaluate fitness of all tours

Identify alpha (best), beta (2nd best), delta (3rd best)
Assign remaining wolves as omega

Repeat until max iterations :
  For each omega wolf:
    For each city position in the tour:
      Update based on alpha, beta, delta influence :
        D-alpha = | C1 * Alpha - position|
        D-beta = | C2 * Beta - position|
        D-delta = | C3 * Delta - position|
        New-Pos = (Alpha - A1 * D-alpha + Beta -
A2 * D-beta + Delta - A3 * D-delta) / 3
      Repair tour (ensure valid permutation)
    Evaluate fitness
  Update alpha, beta, delta (best 3)
  Remaining wolves stay as omega

Return alpha as best solution

OUTPUT :

Best Tour : [1, 0, 2, 4, 3]
Best Distance : 22.35103276995244

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:

Output

```
Iteration 6: Best Fitness = 0.001403
Iteration 7: Best Fitness = 0.001236
Iteration 8: Best Fitness = 0.000012
Iteration 9: Best Fitness = 0.000012
Iteration 10: Best Fitness = 0.000003
Iteration 11: Best Fitness = 0.000000
Iteration 12: Best Fitness = 0.000000
Iteration 13: Best Fitness = 0.000000
Iteration 14: Best Fitness = 0.000000
Iteration 15: Best Fitness = 0.000000
Iteration 16: Best Fitness = 0.000000
Iteration 17: Best Fitness = 0.000000
Iteration 18: Best Fitness = 0.000000
Iteration 19: Best Fitness = 0.000000
Iteration 20: Best Fitness = 0.000000
Iteration 21: Best Fitness = 0.000000
Iteration 22: Best Fitness = 0.000000
Iteration 23: Best Fitness = 0.000000
Iteration 24: Best Fitness = 0.000000
Iteration 25: Best Fitness = 0.000000
Iteration 26: Best Fitness = 0.000000
Iteration 27: Best Fitness = 0.000000
Iteration 28: Best Fitness = 0.000000
Iteration 29: Best Fitness = 0.000000
Iteration 30: Best Fitness = 0.000000
Iteration 31: Best Fitness = 0.000000
Iteration 32: Best Fitness = 0.000000
Iteration 33: Best Fitness = 0.000000
Iteration 34: Best Fitness = 0.000000
Iteration 35: Best Fitness = 0.000000
Iteration 36: Best Fitness = 0.000000
Iteration 37: Best Fitness = 0.000000
Iteration 38: Best Fitness = 0.000000
Iteration 39: Best Fitness = 0.000000
Iteration 40: Best Fitness = 0.000000
Iteration 41: Best Fitness = 0.000000
Iteration 42: Best Fitness = 0.000000
Iteration 43: Best Fitness = 0.000000
Iteration 44: Best Fitness = 0.000000
Iteration 45: Best Fitness = 0.000000
Iteration 46: Best Fitness = 0.000000
Iteration 47: Best Fitness = 0.000000
Iteration 48: Best Fitness = 0.000000
Iteration 49: Best Fitness = 0.000000
Iteration 50: Best Fitness = 0.000000

Best Position: [-3.53962974e-16 -3.09844340e-16]
Best Fitness: 2.2129330195336707e-31
```

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:

Noisy Image (pixel values):

```
[[136 149 126 112 140 105 118 136 155 116]
 [120 108 108 121 150 133 137 121 100 119]
 [150  99 121 148 127 135  99 157 118 130]
 [109 155 119 141 122 146 124 156 139 125]
 [157 113 112 144 148 141 152 149 154 100]
 [134 148 104 118 106 136 115 101 122 157]
 [111 147 155 106 123 150  99 117 125 154]
 [157 104 141 105 146 132 111 114 133 147]
 [137 101  99 103 151 139 101 151 126 115]
 [123 141 131 107 133 111 128 145 112 105]]
```

Smoothed Image (pixel values):

```
[[133 131 127 125 122 123 124 130 130 125]
 [131 128 126 125 125 126 127 127 127 124]
 [127 127 126 126 127 128 128 128 126 125]
 [131 127 126 127 128 129 129 128 127 124]
 [130 128 126 126 127 128 128 128 128 127]
 [130 128 126 126 126 126 127 127 129 131]
 [130 127 125 124 124 125 125 126 128 136]
 [130 126 124 123 123 123 124 124 126 130]
 [131 126 124 122 122 123 124 124 123 122]
 [128 128 124 122 120 123 125 125 121 113]]
```

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:

main.py

```python
    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
    (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)
```