**Experiment 5**

**AIM - Implement Genetics / Hill Climbing in Python.**

**Code – Genetics**

import numpy as np

population\_size = 6

gene\_length = 4

num\_generations = 100

crossover\_rate = 0.5

mutation\_rate = 0.1

num\_parents = int(population\_size / 2)

print("Niyati's Code for Genetics algorithm")

def initialize\_population(size, gene\_length):

return np.random.randint(0, 31, (size, gene\_length))

def calculate\_fitness(chromosome):

objective\_value = abs(sum([chromosome[i] \* (i+1) for i in range(len(chromosome))]) - 30)

fitness\_value = 1 / (1 + objective\_value) # Modified to match the provided logic

return fitness\_value

def select\_parents(population, fitness, num\_parents):

fitness\_sum = np.sum(fitness)

probability = fitness / fitness\_sum

chosen = set() # Set to keep track of which individuals have been chosen

parents = np.empty((num\_parents, population.shape[1]))

for parent\_num in range(num\_parents):

rand = np.random.rand()

cumulative\_probability = 0.0

for i in range(len(probability)):

if i not in chosen:

cumulative\_probability += probability[i]

if rand <= cumulative\_probability:

parents[parent\_num, :] = population[i, :]

chosen.add(i) # Mark this individual as chosen

break

return parents

def crossover(parents, offspring\_size, crossover\_rate):

offspring = np.empty(offspring\_size)

for k in range(offspring\_size[0]):

if np.random.rand() < crossover\_rate:

parent1\_idx = k % parents.shape[0]

parent2\_idx = (k+1) % parents.shape[0]

crossover\_point = np.random.randint(1, offspring\_size[1])

offspring[k, 0:crossover\_point] = parents[parent1\_idx, 0:crossover\_point]

offspring[k, crossover\_point:] = parents[parent2\_idx, crossover\_point:]

return offspring

def mutate(offspring\_crossover, mutation\_rate):

for idx in range(offspring\_crossover.shape[0]):

for gene in range(offspring\_crossover.shape[1]):

if np.random.rand() < mutation\_rate:

offspring\_crossover[idx, gene] = np.random.randint(0, 31)

return offspring\_crossover

population = initialize\_population(population\_size, gene\_length)

print("Initial Population:\n", population)

for generation in range(num\_generations):

fitness = np.array([calculate\_fitness(individual) for individual in population])

print(f"\nGeneration {generation} Fitness:\n", fitness)

parents = select\_parents(population, fitness, num\_parents)

print("Selected Parents:\n", parents)

offspring\_crossover = crossover(parents, (population\_size - num\_parents, gene\_length), crossover\_rate)

print("Crossover Offspring:\n", offspring\_crossover)

offspring\_mutation = mutate(offspring\_crossover, mutation\_rate)

print("Mutated Offspring:\n", offspring\_mutation)

population[:num\_parents, :] = parents

population[num\_parents:, :] = offspring\_mutation

break

def genetic\_algorithm(population, population\_size, gene\_length, num\_generations, crossover\_rate, mutation\_rate, num\_parents):

for generation in range(num\_generations):

fitness = np.array([calculate\_fitness(individual) for individual in population])

parents = select\_parents(population, fitness, num\_parents)

offspring\_crossover = crossover(parents, (population\_size - num\_parents, gene\_length), crossover\_rate)

offspring\_mutation = mutate(offspring\_crossover, mutation\_rate)

population[:num\_parents, :] = parents

population[num\_parents:, :] = offspring\_mutation

final\_fitness = np.array([calculate\_fitness(individual) for individual in population])

best\_index = np.argmax(final\_fitness) # Use argmax because we are using inverted fitness values

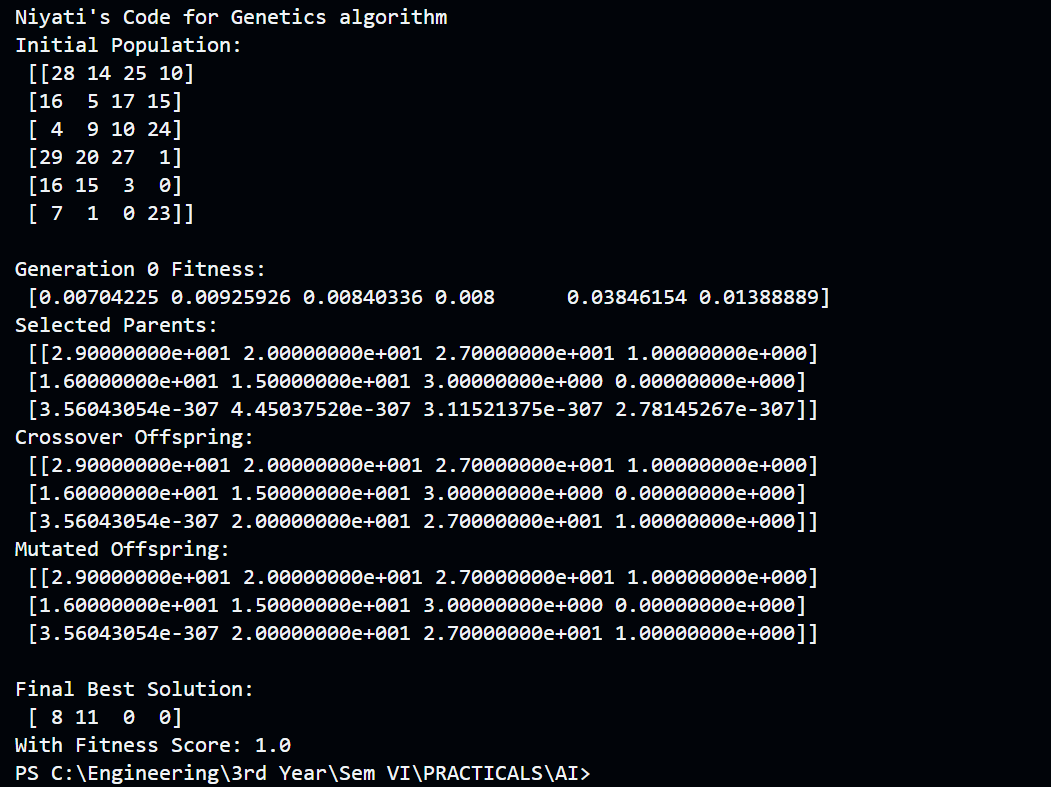
best\_solution = population[best\_index]

print("\nFinal Best Solution:\n", best\_solution)

print("With Fitness Score:", final\_fitness[best\_index])

genetic\_algorithm(population, population\_size, gene\_length, num\_generations, crossover\_rate, mutation\_rate,num\_parents)

**Output –**



**Code – Hill Climbing**

import random

# Objective function to be maximized

def objective\_function(x):

return -x \*\* 2

# Generate initial solution randomly

def generate\_initial\_solution():

return random.randint(-100, 100)

# Generate neighbour solutions

def generate\_neighbours(solution):

neighbours = []

for delta in [-1, 1]:

neighbours.append(solution + delta)

return neighbours

# Get highest quality neighbour of current solution

def get\_best\_neighbour(neighbours):

best\_neighbour = neighbours[0]

best\_quality = objective\_function(best\_neighbour)

for neighbour in neighbours[1:]:

neighbour\_quality = objective\_function(neighbour)

if neighbour\_quality > best\_quality:

best\_quality = neighbour\_quality

best\_neighbour = neighbour

return best\_neighbour

# Hill climbing algorithm

def hill\_climbing():

current\_solution = generate\_initial\_solution()

# print("Initial Solution: ", current\_solution)

while True:

neighbours = generate\_neighbours(current\_solution)

best\_neighbour = get\_best\_neighbour(neighbours)

if objective\_function(best\_neighbour) <= objective\_function(current\_solution):

return current\_solution

current\_solution = best\_neighbour

best\_solution = hill\_climbing()

print("Niyati's Code for Hill Climbing")

print("Best solution found:", best\_solution)

print("Objective function value:", objective\_function(best\_solution))

**Output –**

