4. Suponga que desea utilizar Programación Genética para encontrar el diseño de un circuito lógico, tome como, ejemplo el codificador de 7 segmentos. Describa el conjunto de terminales, el conjunto de funciones y la función de aptitud. Use una librería de Python.

```
import random
import operator
```

En este bloque, se importan los módulos random y operator. random se utiliza para generar números aleatorios, y operator contiene operadores lógicos (AND, OR, NOT, XOR) que serán utilizados en operaciones lógicas en el código.

```
terminals = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

functions = {
    'AND': operator.and_,
    'OR': operator.or_,
    'NOT': operator.invert,
    'XOR': operator.xor
}
```

Se definen dos listas: terminals, que representa las entradas binarias del circuito, y functions, que almacena operaciones lógicas como funciones. En este caso, se asocian operadores lógicos de Python a las cadenas 'AND', 'OR', 'NOT' y 'XOR'.

```
def fitness(individual):
    try:
        expected_outputs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        actual_outputs = []

        num_inputs = len(terminals)
        for digit in range(10):
            inputs = [int(bit) for bit in format(digit, f'0{num_inputs}b')]
            output = evaluate_circuit(individual, inputs)
            actual_outputs.append(output)

        fitness_value = sum(abs(expected - actual) for expected, actual in zip(expe
        return fitness_value
    except ValueError:
        return 1e9
```

La función fitness calcula la aptitud de un individuo. Para ello, se compara la salida real del circuito representado por el individuo con salidas esperadas para todas las combinaciones de entradas binarias del 0 al 9. La función devuelve un valor de aptitud que mide la diferencia entre las salidas esperadas y reales. Si se produce una excepción (por ejemplo, debido a errores en la evaluación del circuito), se asigna un valor de aptitud alto (1e9) para penalizar el individuo.

```
def evaluate_circuit(individual, inputs):
    stack = []
    for gene in individual:
        if gene in terminals:
            stack.append(inputs[terminals.index(gene)])
        elif gene in functions:
            if gene == 'NOT':
                if len(stack) >= 1:
                    operand = stack.pop()
                    result = functions[gene](operand)
                    stack.append(result)
                else:
                    raise ValueError("Operación 'NOT' necesita un operando en la pi
            else:
                if len(stack) >= 2:
                    operand2 = stack.pop()
                    operand1 = stack.pop()
                    result = functions[gene](operand1, operand2)
                    stack.append(result)
                else:
                    raise ValueError("Operación necesita dos operandos en la pila."
    if len(stack) == 1:
        return stack[0]
    else:
        raise ValueError("El diseño del circuito no es válido. Falta operar en algu
```

La función evaluate_circuit toma un individuo (una secuencia de genes) y una lista de entradas binarias como argumentos. Luego, evalúa el circuito lógico representado por el individuo utilizando una pila para realizar operaciones lógicas. Si el circuito no se evalúa correctamente debido a errores, se lanzan excepciones. La función devuelve el resultado de la evaluación del circuito.

```
def create_individual():
    chromosome_length = random.randint(10, 20)
    return [random.choice(terminals + list(functions.keys())) for _ in range(chromo

population_size = 100
population = [create_individual() for _ in range(population_size)]

generations = 50
mutation_rate = 0.1
```

Se define una función create_individual que genera individuos aleatorios. Cada individuo tiene una longitud de cromosoma entre 10 y 20, y sus genes son seleccionados aleatoriamente de la lista de terminales y funciones. Luego, se crea una población inicial de 100 individuos utilizando esta función.

```
for generation in range(generations):
    fitness_scores = [fitness(individual) for individual in population]
    selected_indices = random.choices(range(population_size), k=population_size //
    selected_population = [population[i] for i in selected_indices]

    new_population = []
    for _ in range(population_size // 2):
        parent1, parent2 = random.choices(selected_population, k=2)
        crossover_point = random.randint(1, min(len(parent1), len(parent2)) - 1)
        child1 = parent1[:crossover_point] + parent2[crossover_point:]
        child2 = parent2[:crossover_point] + parent1[crossover_point:]

        if random.random() < mutation_rate:
            mutation_point = random.randint(0, len(child1) - 1)
            child1[mutation_point] = random.choice(terminals + list(functions.keys(
        if random.random() < mutation_rate:
            mutation_point = random.randint(0, len(child2) - 1)
            child2[mutation_point] = random.choice(terminals + list(functions.keys(

        new_population.extend([child1, child2])

    population = new_population
```

Encontramos el diseno optimo

```
best_individual = min(population, key=fitness)
best_fitness = fitness(best_individual)

print("Mejor diseño:", best_individual)
print("Aptitud del mejor diseño:", best_fitness)
```

```
Mejor diseño: ['F', 'G', 'B', 'OR', 'NOT', 'E', 'XOR', 'NOT', 'C', 'D', 'OR',
Aptitud del mejor diseño: 1000000000.0
```