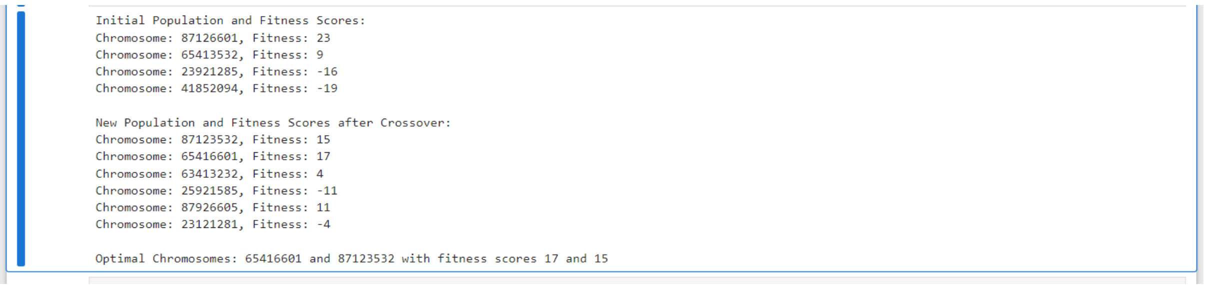
Q.9) Suppose a GA uses chromosome of the form x=abcdefgh , with a fixed length of 8 genes. Each gene can be any digit between 0 to 9. Let fitness function of the given statement f(x)=(a+b)-(c+d)+(e+f)-(g+h). Solve the GA problem to optimize this function with the given initial values of chromosomes, x1=65413532,x2=87126601,x3=23921285,x4=41852094, with following operators- i) evaluate the fitness of each individual, ii)crossover using one point at the middle between the 2 higher rank values, iii)crossover 2nd & 3rd rank value at the position of b & f, iv)cross the 1st & 3rd rank uniform. Do optimization of the function & print the optimal values or fitted values at last.

```
#phenotype GA optimization


import random  # Import the random module to use functions for generating random numbers


# Fitness function: f(x) = (a + b) - (c + d) + (e + f) - (g + h)
def fitness_fun(chromosome):
    # Convert each character in the chromosome string to an integer
    a, b, c, d, e, f, g, h = map(int, list(chromosome))
    # Compute fitness using the formula: (a + b) - (c + d) + (e + f) - (g + h)
    return (a + b) - (c + d) + (e + f) - (g + h)


# One-point crossover at the middle position
def one_point_crossover_middle(parent1, parent2):
    midpoint = len(parent1) // 2  # Calculate the midpoint of the chromosome length


    # Create two children by swapping halves of the parent chromosomes
    # `parent1[:midpoint]`: Take the first half of `parent1`
    # `parent2[midpoint:]`: Take the second half of `parent2`
    child1 = parent1[:midpoint] + parent2[midpoint:]


    # `parent2[:midpoint]`: Take the first half of `parent2`
    # `parent1[midpoint:]`: Take the second half of `parent1`
    child2 = parent2[:midpoint] + parent1[midpoint:]

    return child1, child2  # Return the two new child chromosomes

# One-point crossover at 'b' and 'f' positions
def one_point_crossover_b_f(parent1, parent2):
    pos_b = 2  # Position 'b' is the second gene (1-based index 2)
    pos_f = 6  # Position 'f' is the sixth gene (1-based index 6)

    # Create two children by swapping genes between the parents at positions 'b' and 'f'
    # `parent1[:pos_b - 1]`: Take the segment of `parent1` before position 'b'
    # `parent2[pos_b - 1]`: Add the gene from `parent2` at position 'b'
    # `parent1[pos_b:pos_f - 1]`: Add the segment of `parent1` from position 'b' to just before 'f'
    # `parent2[pos_f - 1]`: Add the gene from `parent2` at position 'f'
    # `parent1[pos_f:]`: Add the segment of `parent1` from position 'f' to the end
    child1 = parent1[:pos_b - 1] + parent2[pos_b - 1] + parent1[pos_b:pos_f - 1] + parent2[pos_f - 1] +
parent1[pos_f:]

    # `parent2[:pos_b - 1]`: Take the segment of `parent2` before position 'b'
    # `parent1[pos_b - 1]`: Add the gene from `parent1` at position 'b'
    # `parent2[pos_b:pos_f - 1]`: Add the segment of `parent2` from position 'b' to just before 'f'
    # `parent1[pos_f - 1]`: Add the gene from `parent1` at position 'f'
    # `parent2[pos_f:]`: Add the segment of `parent2` from position 'f' to the end
    child2 = parent2[:pos_b - 1] + parent1[pos_b - 1] + parent2[pos_b:pos_f - 1] + parent1[pos_f - 1] +
parent2[pos_f:]

    return child1, child2  # Return the two new child chromosomes

# Uniform crossover
def uniform_crossover(parent1, parent2):
    child1 = ''  # Initialize an empty string for the first child chromosome
    child2 = ''  # Initialize an empty string for the second child chromosome
    for i in range(len(parent1)):  # Loop through each gene position in the chromosomes
        if random.random() > 0.5:  # Randomly choose to take gene from parent1 or parent2
            child1 += parent1[i]  # Add gene from parent1 to child1
            child2 += parent2[i]  # Add gene from parent2 to child2
        else:
            child1 += parent2[i]  # Add gene from parent2 to child1
            child2 += parent1[i]  # Add gene from parent1 to child2
    return child1, child2  # Return the two new child chromosomes

# Calculate the difference between the two highest fitness scores
def calculate_difference(fitness_scores):
    sorted_scores = sorted(fitness_scores, reverse=True)  # Sort fitness scores in descending order
    return sorted_scores[0] - sorted_scores[1]  # Return the difference between the highest and the
second highest

# Main Genetic Algorithm function
def genetic_algorithm():
    # Initial chromosomes
    population = ['65413532', '87126601', '23921285', '41852094']  # List of initial chromosomes

    # Evaluate fitness of each chromosome
    fitness_scores = [fitness_fun(chrom) for chrom in population]  # Calculate fitness for each
chromosome

    # Rank chromosomes based on fitness (descending order)
    ranked_population = sorted(zip(population, fitness_scores), key=lambda x: x[1], reverse=True)
    # `zip(population, fitness_scores)` pairs each chromosome with its fitness score.
    # `sorted(..., key=lambda x: x[1], reverse=True)` sorts these pairs by fitness score in descending
order.

    # Calculate the difference between the two highest fitness scores in the initial population
    initial_diff = calculate_difference(fitness_scores)
    # Print initial population and their fitness scores
    print("Initial Population and Fitness Scores:")
    for chrom, fitness in ranked_population:
        print(f"Chromosome: {chrom}, Fitness: {fitness}")

    # Perform one-point crossover at the middle position between the top 2 ranked chromosomes
    parent1, parent2 = ranked_population[0][0], ranked_population[1][0]  # Get the top 2 ranked
chromosomes
    child1, child2 = one_point_crossover_middle(parent1, parent2)  # Apply one-point crossover to
create 2 new children

    # Perform one-point crossover at 'b' and 'f' positions between the 2nd and 3rd ranked
chromosomes
    parent2, parent3 = ranked_population[1][0], ranked_population[2][0]  # Get the 2nd and 3rd
ranked chromosomes
    child3, child4 = one_point_crossover_b_f(parent2, parent3)  # Apply one-point crossover at 'b' and
'f' positions

    # Perform uniform crossover between the 1st and 3rd ranked chromosomes
    parent1, parent3 = ranked_population[0][0], ranked_population[2][0]  # Get the 1st and 3rd
ranked chromosomes
    child5, child6 = uniform_crossover(parent1, parent3)  # Apply uniform crossover to create 2 new
children

    # Combine all new children to form a new population
    new_population = [child1, child2, child3, child4, child5, child6]  # List of new chromosomes after
crossover

    # Evaluate fitness of new population
    new_fitness_scores = [fitness_fun(chrom) for chrom in new_population]  # Calculate fitness for
each new chromosome

    # Calculate the difference between the two highest fitness scores in the new population
    new_diff = calculate_difference(new_fitness_scores)

    # Rank new population based on fitness (descending order)
    ranked_new_population = sorted(zip(new_population, new_fitness_scores), key=lambda x: x[1],
reverse=True)  #ranks the new population by sorting chromosomes based on their fitness scores in
descending order

    # Print new population and their fitness scores
    print("\nNew Population and Fitness Scores after Crossover:")
    for chrom, fitness in zip(new_population, new_fitness_scores):
        print(f"Chromosome: {chrom}, Fitness: {fitness}")
    # `zip(new_population, new_fitness_scores)` pairs each new chromosome with its fitness score.
    # This allows for easy iteration to print out the chromosomes and their respective fitness scores.

    # Determine whether to select the new population or not based on the difference criterion
    if new_diff < initial_diff:  # If the difference between the top two fitness scores in the new
population is smaller
        # Print the optimal chromosomes from the new population based on the new_diff
        print(f"\nOptimal Chromosomes: {ranked_new_population[0][0]} and
{ranked_new_population[1][0]} with fitness scores {ranked_new_population[0][1]} and
{ranked_new_population[1][1]}")
    else:
        # Print the optimal chromosomes from the initial population based on the initial_diff
        print(f"\nOptimal Chromosomes: {ranked_population[0][0]} and {ranked_population[1][0]} with
fitness scores {ranked_population[0][1]} and {ranked_population[1][1]}")

# Run the genetic algorithm
genetic_algorithm()  # Call the main function to start the genetic algorithm process
```



```
Initial Population and Fitness Scores:
Chromosome: 87126601, Fitness: 23
Chromosome: 65413532, Fitness: 9
Chromosome: 23921285, Fitness: -16
Chromosome: 41852094, Fitness: -19

New Population and Fitness Scores after Crossover:
Chromosome: 87123532, Fitness: 15
Chromosome: 65416601, Fitness: 17
Chromosome: 63413232, Fitness: 4
Chromosome: 25921585, Fitness: -11
Chromosome: 87926605, Fitness: 11
Chromosome: 23121281, Fitness: -4

Optimal Chromosomes: 65416601 and 87123532 with fitness scores 17 and 15
```