Q.8) Solve the given maximization function, f(x)=x^2, where x ranges from 0 to 31, by using GA with applying, i) encoding technique (binary,5 length chromosome), ii)selection operator(roulette wheel), iii)crossover(one point). Find the value of x(best value) for which f(x) is max. Take initial 4 values & they are x=13,24,8,19. Solve this problem using python.

```
#genotype GA max

import pandas as pd  # Import pandas for data manipulation and analysis
import random  # Import random for generating random numbers

# Function to evaluate: f(x) = x^2
def fitness_fun(x):
    return x ** 2  # Returns the square of the input value x as its fitness

# Function to convert a decimal integer to a 5-bit binary string
def decimal_to_binary(x):
    return format(x, '05b')  # Converts a decimal number to a binary string of length 5

# Function to decode a binary string back to a decimal integer
def decode(binary_str):
    return int(binary_str, 2)  # Converts a binary string to a decimal integer

# Roulette wheel selection function
def roulette_wheel_selection(population, fitness_scores):
    total_fitness = sum(fitness_scores)  # Calculate the total fitness of the population
    selection_probs = [f / total_fitness for f in fitness_scores]  # Calculate selection probability for each chromosome
    selected_index = random.choices(range(len(population)), weights=selection_probs, k=1)[0]  # Randomly select one chromosome based on its selection probability
    return population[selected_index]  # Return the selected chromosome
```

```
# One-point crossover function

def one_point_crossover(parent1, parent2):

    crossover_point = random.randint(1, len(parent1) - 1)  # Randomly select a crossover point between 1 and length of chromosome - 1

    child1 = parent1[:crossover_point] + parent2[crossover_point:]  # Create the first child by combining segments from both parents

    child2 = parent2[:crossover_point] + parent1[crossover_point:]  # Create the second child by combining segments from both parents

    return child1, child2  # Return the two children


# # Mutation function

# def mutate(chromosome, mutation_rate=0.01):

#     """Randomly flips bits in the chromosome with a certain mutation rate."""

#     new_chromosome = ""  # Initialize an empty string to store the mutated chromosome

#     for gene in chromosome:

#         if random.random() < mutation_rate:  # Decide whether to mutate this gene

#             # Flip the gene (0 becomes 1, 1 becomes 0)

#             new_chromosome += '0' if gene == '1' else '1'

#         else:

#             new_chromosome += gene  # Keep the gene as it is

#     return new_chromosome  # Return the mutated chromosome


# Main Genetic Algorithm function

def genetic_algorithm():

    # Initial values of x

    x = [13, 24, 8, 19]  # Initial population of four individuals with given x values


    # Encoding and Initial Setup

    string = []  # To store labels for chromosomes

    initial_population = []  # To store binary representation of chromosomes

    fitness = []  # To store fitness values of chromosomes
```

```
    for i in range(len(x)):
        string.append('s{}'.format(i + 1))  # Create labels for each chromosome starting from s1, s2, etc.
        initial_population.append(decimal_to_binary(x[i]))  # Convert each x value to its binary representation
        fitness.append(fitness_fun(x[i]))  # Calculate fitness for each x value

    # Calculating total and average fitness
    total_val_fitness = sum(fitness)  # Sum of all fitness values
    avg_val_fitness = total_val_fitness / len(x)  # Average fitness value

    # Calculating probability and expected count
    probability = []  # To store probability for each chromosome
    expected_count = []  # To store expected count for each chromosome

    for i in fitness:
        probability.append(i / total_val_fitness)  # Calculate the probability of selection for each chromosome
        expected_count.append(i / avg_val_fitness)  # Calculate the expected count for each chromosome

    # Create a DataFrame to store the population and related statistics
    df = pd.DataFrame({
        'String': string,
        'Initial_Population': initial_population,
        'x': x,
        'Fitness': fitness,
        'Probability': probability,
        'Expected_Count': expected_count,
        'Actual_Count': [round(ec) for ec in expected_count]  # Round the expected count to get the actual count
    })
```

```
# Display the initial population DataFrame
print("Initial Population:")
print(df)

# Selection and crossover
num_generations = 5  # Number of generations to evolve
population_size = len(x)  # Size of the population

for generation in range(num_generations):
    # Selection using Roulette Wheel
    new_population = []  # To store new population after selection and crossover
    for _ in range(population_size // 2):  # Select pairs for crossover
        parent1 = roulette_wheel_selection(df['Initial_Population'].tolist(), df['Fitness'].tolist())  #
Select first parent
        parent2 = roulette_wheel_selection(df['Initial_Population'].tolist(), df['Fitness'].tolist())  #
Select second parent

        # Crossover (one-point)
        child1, child2 = one_point_crossover(parent1, parent2)  # Perform one-point crossover to
generate two children

        # # Mutation (optional)
        # child1 = mutate(child1)  # Apply mutation to the first child
        # child2 = mutate(child2)  # Apply mutation to the second child

        # Add offspring to the new population
        new_population.extend([child1, child2])  # Add the two children to the new population

    # Update population
    df['Initial_Population'] = new_population  # Update DataFrame with new population
```

```python
        df['x'] = [decode(chrom) for chrom in new_population]  # Decode binary strings to decimal x values
        df['Fitness'] = [fitness_fun(chrom) for chrom in df['x']]  # Recalculate fitness values

        # Calculate new probabilities and expected counts
        total_val_fitness = sum(df['Fitness'])  # Recalculate total fitness
        avg_val_fitness = total_val_fitness / len(df['x'])  # Recalculate average fitness
        df['Probability'] = df['Fitness'] / total_val_fitness  # Update probabilities
        df['Expected_Count'] = df['Fitness'] / avg_val_fitness  # Update expected counts
        df['Actual_Count'] = [round(ec) for ec in df['Expected_Count']]  # Update actual counts

        # Print best solution of this generation
        best_individual = df.loc[df['Fitness'].idxmax()]  # Identify the individual with the highest fitness
        print(f"Generation {generation + 1}: Best x = {best_individual['x']}, f(x) = {best_individual['Fitness']}")  # Output best individual

    # Final best solution
    final_best = df.loc[df['Fitness'].idxmax()]  # Identify the best solution after all generations
    print(f"Best value of x after {num_generations} generations: x = {final_best['x']}, f(x) = {final_best['Fitness']}")  # Output the final best solution

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

```
Initial Population:
  String Initial_Population   x  Fitness  Probability  Expected_Count  \
0     s1              01101  13      169     0.144444        0.577778
1     s2              11000  24      576     0.492308        1.969231
2     s3              01000   8       64     0.054701        0.218803
3     s4              10011  19      361     0.308547        1.234188

   Actual_Count
0             1
1             2
2             0
3             1
Generation 1: Best x = 27, f(x) = 729
Generation 2: Best x = 26, f(x) = 676
Generation 3: Best x = 26, f(x) = 676
Generation 4: Best x = 26, f(x) = 676
Generation 5: Best x = 27, f(x) = 729
Best value of x after 5 generations: x = 27, f(x) = 729
```