```
#ALGORITHM

import random

# Define the problem's fitness function (f(x) = x^2)
def fitness(x):
    return x ** 2

# Create an initial population
def create_population(pop_size, x_bounds):
    return [random.randint(x_bounds[0], x_bounds[1]) for _ in range(pop_size)]

# Select two parents using roulette wheel selection
def select_parents(population, fitness_values):
    total_fitness = sum(fitness_values)
    probabilities = [f / total_fitness for f in fitness_values]
    parent1 = random.choices(population, probabilities)[0]
    parent2 = random.choices(population, probabilities)[0]

    # Ensure the parents are distinct
    while parent2 == parent1:
        parent2 = random.choices(population, probabilities)[0]

    return parent1, parent2

# Crossover function (single-point crossover)
def crossover(parent1, parent2):
    # Choose a random point for crossover
    crossover_point = random.randint(1, 9)  # Changed to 9 for a 10-bit number
    # Convert parents to binary
    parent1_bin = bin(parent1)[2:].zfill(10)  # zfill(10) for a 10-bit number
    parent2_bin = bin(parent2)[2:].zfill(10)

    # Swap bits after crossover point
    offspring1_bin = parent1_bin[:crossover_point] + parent2_bin[crossover_point:]
    offspring2_bin = parent2_bin[:crossover_point] + parent1_bin[crossover_point:]

    # Convert back to integers
    offspring1 = int(offspring1_bin, 2)
    offspring2 = int(offspring2_bin, 2)
    return offspring1, offspring2

# Mutation function (flip a random bit)
def mutate(offspring, mutation_rate):
    if random.random() < mutation_rate:
        mutation_point = random.randint(0, 9)  # Changed to 9 for a 10-bit number
        offspring_bin = bin(offspring)[2:].zfill(10)  # zfill(10) for a 10-bit number
        offspring_bin = list(offspring_bin)
        offspring_bin[mutation_point] = '1' if offspring_bin[mutation_point] == '0' else '0'
        offspring = int("".join(offspring_bin), 2)
    return offspring

# Run the genetic algorithm
def genetic_algorithm(pop_size, x_bounds, generations, mutation_rate):
    population = create_population(pop_size, x_bounds)
    for generation in range(generations):
        # Evaluate fitness for each individual
        fitness_values = [fitness(x) for x in population]

        # Select the best individuals (elite selection)
        new_population = []
        for _ in range(pop_size // 2):
            parent1, parent2 = select_parents(population, fitness_values)
            offspring1, offspring2 = crossover(parent1, parent2)
            new_population.extend([mutate(offspring1, mutation_rate), mutate(offspring2, mutation_rate)])

        # Replace the population with the new one
        population = new_population

        # Optionally print the best individual of each generation
        best_individual = max(population, key=fitness)
        print(f"Generation {generation}: Best Individual = {best_individual}, Fitness = {fitness(best_individual)}")

    return max(population, key=fitness)

# Parameters
```

```
population_size = 10
x_bounds = (0, 1023)  # Search space for x (e.g., range of 10-bit integers)
generations = 50
mutation_rate = 0.1

# Run the genetic algorithm
best_solution = genetic_algorithm(population_size, x_bounds, generations, mutation_rate)
print(f"Best Solution: {best_solution}, Fitness: {fitness(best_solution)}")
```

```
Generation 0: Best Individual = 991, Fitness = 982081
Generation 1: Best Individual = 967, Fitness = 935089
Generation 2: Best Individual = 1014, Fitness = 1028196
Generation 3: Best Individual = 1014, Fitness = 1028196
Generation 4: Best Individual = 1014, Fitness = 1028196
Generation 5: Best Individual = 1012, Fitness = 1024144
Generation 6: Best Individual = 1012, Fitness = 1024144
Generation 7: Best Individual = 1012, Fitness = 1024144
Generation 8: Best Individual = 1012, Fitness = 1024144
Generation 9: Best Individual = 1012, Fitness = 1024144
Generation 10: Best Individual = 1020, Fitness = 1040400
Generation 11: Best Individual = 1020, Fitness = 1040400
Generation 12: Best Individual = 1020, Fitness = 1040400
Generation 13: Best Individual = 1012, Fitness = 1024144
Generation 14: Best Individual = 1020, Fitness = 1040400
Generation 15: Best Individual = 1020, Fitness = 1040400
Generation 16: Best Individual = 1020, Fitness = 1040400
Generation 17: Best Individual = 1020, Fitness = 1040400
Generation 18: Best Individual = 1020, Fitness = 1040400
Generation 19: Best Individual = 1012, Fitness = 1024144
Generation 20: Best Individual = 1012, Fitness = 1024144
Generation 21: Best Individual = 1020, Fitness = 1040400
Generation 22: Best Individual = 1020, Fitness = 1040400
Generation 23: Best Individual = 1020, Fitness = 1040400
Generation 24: Best Individual = 1020, Fitness = 1040400
Generation 25: Best Individual = 1012, Fitness = 1024144
Generation 26: Best Individual = 1014, Fitness = 1028196
Generation 27: Best Individual = 1012, Fitness = 1024144
Generation 28: Best Individual = 1014, Fitness = 1028196
Generation 29: Best Individual = 1013, Fitness = 1026169
Generation 30: Best Individual = 1013, Fitness = 1026169
Generation 31: Best Individual = 1013, Fitness = 1026169
Generation 32: Best Individual = 1013, Fitness = 1026169
Generation 33: Best Individual = 1013, Fitness = 1026169
Generation 34: Best Individual = 1022, Fitness = 1044484
Generation 35: Best Individual = 1022, Fitness = 1044484
Generation 36: Best Individual = 1021, Fitness = 1042441
Generation 37: Best Individual = 1020, Fitness = 1040400
Generation 38: Best Individual = 1020, Fitness = 1040400
Generation 39: Best Individual = 1013, Fitness = 1026169
Generation 40: Best Individual = 1013, Fitness = 1026169
Generation 41: Best Individual = 1013, Fitness = 1026169
Generation 42: Best Individual = 1013, Fitness = 1026169
Generation 43: Best Individual = 988, Fitness = 976144
Generation 44: Best Individual = 1013, Fitness = 1026169
Generation 45: Best Individual = 1021, Fitness = 1042441
Generation 46: Best Individual = 1020, Fitness = 1040400
Generation 47: Best Individual = 1021, Fitness = 1042441
Generation 48: Best Individual = 1021, Fitness = 1042441
Generation 49: Best Individual = 1021, Fitness = 1042441
Best Solution: 1021, Fitness: 1042441
```

```
#APPLICATION

import random
import numpy as np

# Define the problem's fitness function (minimizing the total distance)
def fitness(route, dist_matrix):
    # Total distance of the route
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += dist_matrix[route[i]][route[i + 1]]
    # Add distance from last location back to the start if required (for TSP-style problem)
    total_distance += dist_matrix[route[-1]][route[0]]
    return total_distance

# Create an initial population (random permutations of locations)
def create_population(pop_size, num_locations):
    return [random.sample(range(num_locations), num_locations) for _ in range(pop_size)]
```

```
# Select two parents using roulette wheel selection
def select_parents(population, fitness_values):
    total_fitness = sum(fitness_values)
    probabilities = [f / total_fitness for f in fitness_values]
    parent1 = random.choices(population, probabilities)[0]
    parent2 = random.choices(population, probabilities)[0]

    # Ensure the parents are distinct
    while parent2 == parent1:
        parent2 = random.choices(population, probabilities)[0]

    return parent1, parent2

# Crossover function (order crossover for permutation-based representation)
def crossover(parent1, parent2):
    # Randomly select a crossover point
    crossover_point1 = random.randint(0, len(parent1) // 2)
    crossover_point2 = random.randint(crossover_point1, len(parent1))

    # Create offspring by combining segments of the parents
    offspring1 = [-1] * len(parent1)
    offspring2 = [-1] * len(parent2)

    # Copy crossover segments
    offspring1[crossover_point1:crossover_point2] = parent1[crossover_point1:crossover_point2]
    offspring2[crossover_point1:crossover_point2] = parent2[crossover_point1:crossover_point2]

    # Fill in the rest of the offspring
    fill_missing(offspring1, parent2)
    fill_missing(offspring2, parent1)

    return offspring1, offspring2

def fill_missing(offspring, parent):
    for i in range(len(offspring)):
        if offspring[i] == -1:
            for gene in parent:
                if gene not in offspring:
                    offspring[i] = gene
                    break

# Mutation function (swap two random locations in the route)
def mutate(offspring, mutation_rate):
    if random.random() < mutation_rate:
        idx1, idx2 = random.sample(range(len(offspring)), 2)
        offspring[idx1], offspring[idx2] = offspring[idx2], offspring[idx1]
    return offspring

# Run the genetic algorithm
def genetic_algorithm(pop_size, num_locations, generations, mutation_rate, dist_matrix):
    population = create_population(pop_size, num_locations)
    for generation in range(generations):
        # Evaluate fitness for each individual
        fitness_values = [fitness(route, dist_matrix) for route in population]

        # Select the best individuals (elite selection)
        new_population = []
        for _ in range(pop_size // 2):
            parent1, parent2 = select_parents(population, fitness_values)
            offspring1, offspring2 = crossover(parent1, parent2)
            new_population.extend([mutate(offspring1, mutation_rate), mutate(offspring2, mutation_rate)])

        # Replace the population with the new one
        population = new_population

        # Optionally print the best individual of each generation
        best_individual = min(population, key=lambda route: fitness(route, dist_matrix))
        print(f"Generation {generation}: Best Individual = {best_individual}, Fitness (Total Distance) = {fitness(best_individual, dist_matri

    # Return the best solution found
    best_solution = min(population, key=lambda route: fitness(route, dist_matrix))
    return best_solution

# Example distance matrix for locations (symmetric matrix)
num_locations = 5  # Example number of locations
dist_matrix = np.random.randint(10, 100, size=(num_locations, num_locations))
np.fill_diagonal(dist_matrix, 0)  # Diagonal should be zero, no distance to itself
```

```
# Parameters
population_size = 10
generations = 50
mutation_rate = 0.1

# Run the genetic algorithm
best_solution = genetic_algorithm(population_size, num_locations, generations, mutation_rate, dist_matrix)
print(f"Best Solution: {best_solution}, Fitness (Total Distance): {fitness(best_solution, dist_matrix)}")
```

```
Generation 0: Best Individual = [1, 2, 0, 4, 3], Fitness (Total Distance) = 157
Generation 1: Best Individual = [1, 3, 2, 4, 0], Fitness (Total Distance) = 237
Generation 2: Best Individual = [1, 4, 3, 0, 2], Fitness (Total Distance) = 239
Generation 3: Best Individual = [1, 4, 2, 0, 3], Fitness (Total Distance) = 243
Generation 4: Best Individual = [1, 4, 2, 0, 3], Fitness (Total Distance) = 243
Generation 5: Best Individual = [1, 4, 2, 0, 3], Fitness (Total Distance) = 243
Generation 6: Best Individual = [1, 2, 4, 0, 3], Fitness (Total Distance) = 210
Generation 7: Best Individual = [1, 3, 2, 4, 0], Fitness (Total Distance) = 237
Generation 8: Best Individual = [1, 4, 3, 2, 0], Fitness (Total Distance) = 209
Generation 9: Best Individual = [0, 3, 1, 4, 2], Fitness (Total Distance) = 243
Generation 10: Best Individual = [4, 3, 1, 2, 0], Fitness (Total Distance) = 157
Generation 11: Best Individual = [4, 3, 1, 2, 0], Fitness (Total Distance) = 157
Generation 12: Best Individual = [4, 3, 1, 2, 0], Fitness (Total Distance) = 157
Generation 13: Best Individual = [4, 3, 1, 2, 0], Fitness (Total Distance) = 157
Generation 14: Best Individual = [4, 3, 1, 2, 0], Fitness (Total Distance) = 157
Generation 15: Best Individual = [3, 1, 4, 0, 2], Fitness (Total Distance) = 180
Generation 16: Best Individual = [3, 1, 4, 0, 2], Fitness (Total Distance) = 180
Generation 17: Best Individual = [3, 1, 4, 0, 2], Fitness (Total Distance) = 180
Generation 18: Best Individual = [2, 3, 1, 4, 0], Fitness (Total Distance) = 180
Generation 19: Best Individual = [3, 1, 4, 0, 2], Fitness (Total Distance) = 180
Generation 20: Best Individual = [3, 1, 4, 0, 2], Fitness (Total Distance) = 180
Generation 21: Best Individual = [0, 4, 1, 2, 3], Fitness (Total Distance) = 204
Generation 22: Best Individual = [3, 1, 4, 0, 2], Fitness (Total Distance) = 180
Generation 23: Best Individual = [3, 1, 4, 0, 2], Fitness (Total Distance) = 180
Generation 24: Best Individual = [3, 1, 2, 4, 0], Fitness (Total Distance) = 210
Generation 25: Best Individual = [3, 2, 0, 4, 1], Fitness (Total Distance) = 202
Generation 26: Best Individual = [2, 3, 0, 4, 1], Fitness (Total Distance) = 204
Generation 27: Best Individual = [2, 3, 0, 4, 1], Fitness (Total Distance) = 204
Generation 28: Best Individual = [3, 2, 0, 4, 1], Fitness (Total Distance) = 202
Generation 29: Best Individual = [3, 2, 0, 4, 1], Fitness (Total Distance) = 202
Generation 30: Best Individual = [3, 2, 0, 4, 1], Fitness (Total Distance) = 202
Generation 31: Best Individual = [3, 2, 0, 4, 1], Fitness (Total Distance) = 202
Generation 32: Best Individual = [1, 3, 2, 0, 4], Fitness (Total Distance) = 202
Generation 33: Best Individual = [3, 1, 4, 0, 2], Fitness (Total Distance) = 180
Generation 34: Best Individual = [1, 2, 4, 0, 3], Fitness (Total Distance) = 210
Generation 35: Best Individual = [2, 4, 0, 3, 1], Fitness (Total Distance) = 210
Generation 36: Best Individual = [3, 1, 4, 0, 2], Fitness (Total Distance) = 180
Generation 37: Best Individual = [3, 2, 1, 0, 4], Fitness (Total Distance) = 222
Generation 38: Best Individual = [3, 2, 1, 0, 4], Fitness (Total Distance) = 222
Generation 39: Best Individual = [0, 4, 1, 3, 2], Fitness (Total Distance) = 202
Generation 40: Best Individual = [3, 1, 4, 0, 2], Fitness (Total Distance) = 180
Generation 41: Best Individual = [0, 2, 3, 1, 4], Fitness (Total Distance) = 180
Generation 42: Best Individual = [1, 3, 2, 0, 4], Fitness (Total Distance) = 202
Generation 43: Best Individual = [1, 4, 0, 2, 3], Fitness (Total Distance) = 180
Generation 44: Best Individual = [1, 3, 2, 0, 4], Fitness (Total Distance) = 202
Generation 45: Best Individual = [0, 3, 1, 2, 4], Fitness (Total Distance) = 210
Generation 46: Best Individual = [0, 3, 1, 2, 4], Fitness (Total Distance) = 210
Generation 47: Best Individual = [0, 3, 1, 4, 2], Fitness (Total Distance) = 243
Generation 48: Best Individual = [3, 1, 4, 0, 2], Fitness (Total Distance) = 180
Generation 49: Best Individual = [3, 1, 2, 0, 4], Fitness (Total Distance) = 157
Best Solution: [3, 1, 2, 0, 4], Fitness (Total Distance): 157
```

Start coding or generate with AI.