```
import numpy as np
import random

# Generate random cities (for the sake of this example, 10 cities)
num_cities = 10
cities = np.random.rand(num_cities, 2)  # Random positions of cities in a 2D plane

# Distance matrix (Euclidean distance between each pair of cities)
def euclidean_distance(city1, city2):
    return np.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

# Create a distance matrix for all cities
dist_matrix = np.zeros((num_cities, num_cities))
for i in range(num_cities):
    for j in range(num_cities):
        dist_matrix[i][j] = euclidean_distance(cities[i], cities[j])

# Fitness function: Calculate the total distance for a route
def fitness(route):
    total_distance = 0
    for i in range(len(route)-1):
        total_distance += dist_matrix[route[i], route[i+1]]
    total_distance += dist_matrix[route[-1], route[0]]  # Return to the start city
    return total_distance

# Generate initial population of routes
def generate_population(pop_size):
    population = []
    for _ in range(pop_size):
        route = list(np.random.permutation(num_cities))  # Random permutation of cities
        population.append(route)
    return population

# Selection: Tournament Selection (randomly pick 2 individuals, select the better one)
def select(population):
    tournament_size = 3
    selected = []
    for _ in range(2):  # Select two individuals
        tournament = random.sample(population, tournament_size)
        tournament.sort(key=lambda x: fitness(x))  # Sort by fitness (lower distance = better)
        selected.append(tournament[0])
    return selected

# Crossover: Ordered Crossover (OX)
def crossover(parent1, parent2):
    size = len(parent1)
    start, end = sorted([random.randint(0, size-1), random.randint(0, size-1)])

    # Create offspring by copying part of parent1
    offspring = [-1] * size
    offspring[start:end] = parent1[start:end]

    # Fill the remaining positions with genes from parent2
    current_pos = end
    for i in range(size):
        if parent2[i] not in offspring:
            if current_pos == size:
                current_pos = 0
            offspring[current_pos] = parent2[i]
            current_pos += 1
    return offspring

# Mutation: Swap Mutation (randomly swap two cities in the route)
def mutate(route, mutation_rate=0.1):
    if random.random() < mutation_rate:
        idx1, idx2 = random.sample(range(len(route)), 2)
        route[idx1], route[idx2] = route[idx2], route[idx1]
    return route

# Main Genetic Algorithm function
def genetic_algorithm(pop_size=100, num_generations=100, mutation_rate=0.1):
    population = generate_population(pop_size)

    best_route = None
    best_fitness = float('inf')

    for generation in range(num_generations):
        new_population = []

        # Elitism: Keep the best individual
        population.sort(key=lambda x: fitness(x))  # Sort population by fitness (lower is better)
```

```
            new_population.append(population[0])  # Keep the best route

            # Create new individuals through selection, crossover, and mutation
            while len(new_population) < pop_size:
                parent1, parent2 = select(population)  # Tournament selection
                offspring = crossover(parent1, parent2)  # Crossover to create offspring
                offspring = mutate(offspring, mutation_rate)  # Mutation
                new_population.append(offspring)

            population = new_population  # Replace population with new population

            # Track the best solution
            current_best = population[0]
            current_best_fitness = fitness(current_best)
            if current_best_fitness < best_fitness:
                best_route = current_best
                best_fitness = current_best_fitness

            # Print progress for each generation
            print(f"Generation {generation + 1}, Best fitness (total distance): {best_fitness}")

        return best_route, best_fitness

    # Run the Genetic Algorithm for 100 iterations
    best_route, best_fitness = genetic_algorithm(num_generations=100)

    # Print the best route and its fitness (total distance)
    print(f"\nBest route after 100 generations: {best_route}")
    print(f"Best fitness (total distance): {best_fitness}")
```

```
Generation 46, Best fitness (total distance): 3.280413651429527
Generation 47, Best fitness (total distance): 3.280413651429527
Generation 48, Best fitness (total distance): 3.280413651429527
Generation 49, Best fitness (total distance): 3.280413651429527
Generation 50, Best fitness (total distance): 3.280413651429527
Generation 51, Best fitness (total distance): 3.280413651429527
Generation 52, Best fitness (total distance): 3.280413651429527
Generation 53, Best fitness (total distance): 3.280413651429527
Generation 54, Best fitness (total distance): 3.280413651429527
Generation 55, Best fitness (total distance): 3.280413651429527
Generation 56, Best fitness (total distance): 3.280413651429527
```

```
Best route after 100 generations: [8, 7, 5, 6, 2, 9, 4, 3, 1, 0]
Best fitness (total distance): 3.280413651429527
```