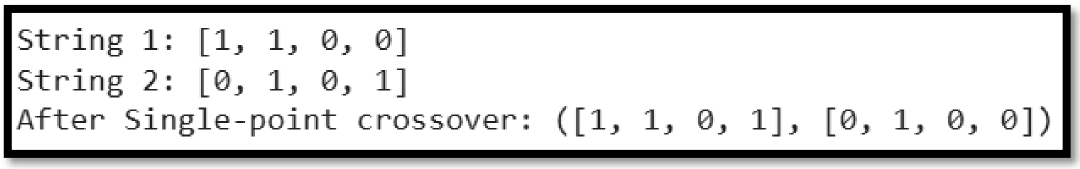
PRACTICAL: 4

Aim: Implement different crossover and Mutation Methods of GA in Python.

Code: Single Point Crossover

```
def single_point_crossover(parent1, parent2):
  point = random.randint(1, len(parent1)-1)
  return parent1[:point] + parent2[point:], parent2[:point] + parent1[point:]
p1 = [1,1,0,0]
p2 = [0,1,0,1]
print("String 1:", p1)
print("String 2:", p2)
print("After single-point crossover:",single_point_crossover(p1, p2))
```

Output:

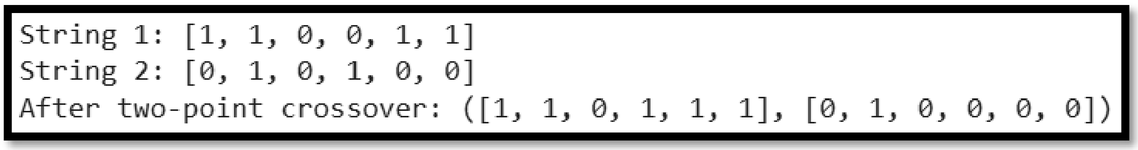


```
String 1: [1, 1, 0, 0]
String 2: [0, 1, 0, 1]
After Single-point crossover: ([1, 1, 0, 1], [0, 1, 0, 0])
```

Code: Two-Point Crossover

```
def two_point_crossover(parent1, parent2):
  p1, p2 = sorted(random.sample(range(1, len(parent1)-1), 2))
  return (parent1[:p1] + parent2[p1:p2] + parent1[p2:],
      parent2[:p1] + parent1[p1:p2] + parent2[p2:])
x1 = [1,1,0,0,1,1]
x2 = [0,1,0,1,0,0]
print("String 1:", x1)
print("String 2:", x2)
print("After two-point crossover:",two_point_crossover(x1, x2))
```

Output:

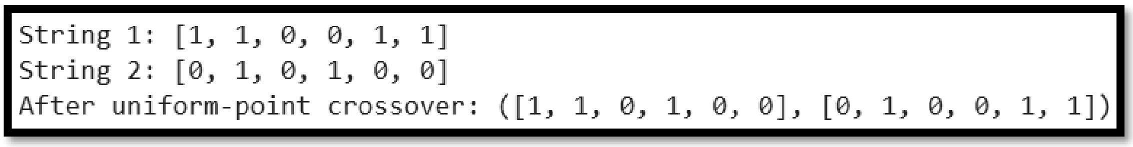


```
String 1: [1, 1, 0, 0, 1, 1]
String 2: [0, 1, 0, 1, 0, 0]
After two-point crossover: ([1, 1, 0, 1, 1, 1], [0, 1, 0, 0, 0, 0])
```

Code: Uniform Crossover

```
def uniform_crossover(parent1, parent2, prob=0.5):
    child1, child2 = [], []
    for i in range(len(parent1)):
        if random.random() < prob:
            child1.append(parent1[i])
            child2.append(parent2[i])
        else:
            child1.append(parent2[i])
            child2.append(parent1[i])
    return child1, child2
s1 = [1,1,0,0,1,1]
s2 = [0,1,0,1,0,0]
print("String 1:", s1)
print("String 2:", s2)
print("After uniform-point crossover:",uniform_crossover(s1,s2))
```

Output:

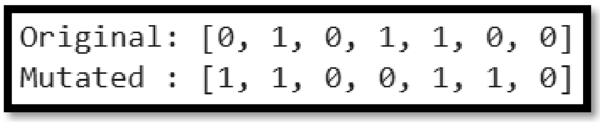


```
String 1: [1, 1, 0, 0, 1, 1]
String 2: [0, 1, 0, 1, 0, 0]
After uniform-point crossover: ([1, 1, 0, 1, 0, 0], [0, 1, 0, 0, 1, 1])
```

Code: Bit Flip Mutation

```
def bit_flip_mutation_k(individual, k=2):
    mutated = individual[:]
    # Choose k unique positions
    positions = random.sample(range(len(individual)), k)
    for i in positions:
        mutated[i] = 1 - mutated[i]  # flip bit
    return mutated
chromosome = [0, 1, 0, 1, 1, 0, 0]
print("Original:", chromosome)
print("Mutated :", bit_flip_mutation_k(chromosome, k=3))
```

Output:

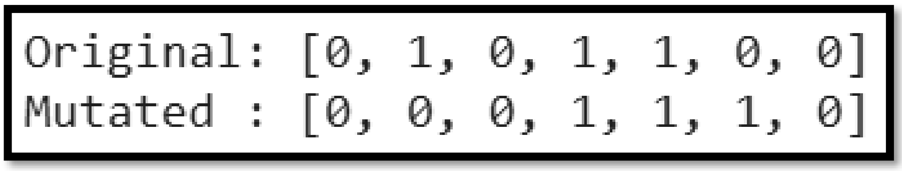


```
Original: [0, 1, 0, 1, 1, 0, 0]
Mutated : [1, 1, 0, 0, 1, 1, 0]
```

Code: Swap Mutation

```
def swap_mutation(individual):
    mutated = individual[:]
    # Select two distinct random positions
    i, j = random.sample(range(len(individual)), 2)
    # Swap the genes
    mutated[i], mutated[j] = mutated[j], mutated[i]
    return mutated
chromosome = [0, 1, 0, 1, 1, 0, 0]
print("Original:", chromosome)
print("Mutated :", swap_mutation(chromosome))
```

Output:

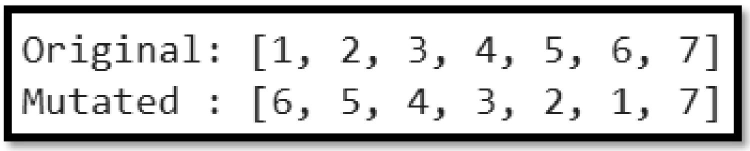


```
Original: [0, 1, 0, 1, 1, 0, 0]
Mutated : [0, 0, 0, 1, 1, 1, 0]
```

Code: Inverse Mutation

```
def inverse_mutation(individual):
    mutated = individual[:]
    i, j = sorted(random.sample(range(len(individual)), 2))
    # Reverse the subsequence
    mutated[i:j+1] = reversed(mutated[i:j+1])
    return mutated
chromosome = [1, 2, 3, 4, 5, 6, 7]
print("Original:", chromosome)
print("Mutated :", inverse_mutation(chromosome))
```

Output:



```
Original: [1, 2, 3, 4, 5, 6, 7]
Mutated : [6, 5, 4, 3, 2, 1, 7]
```