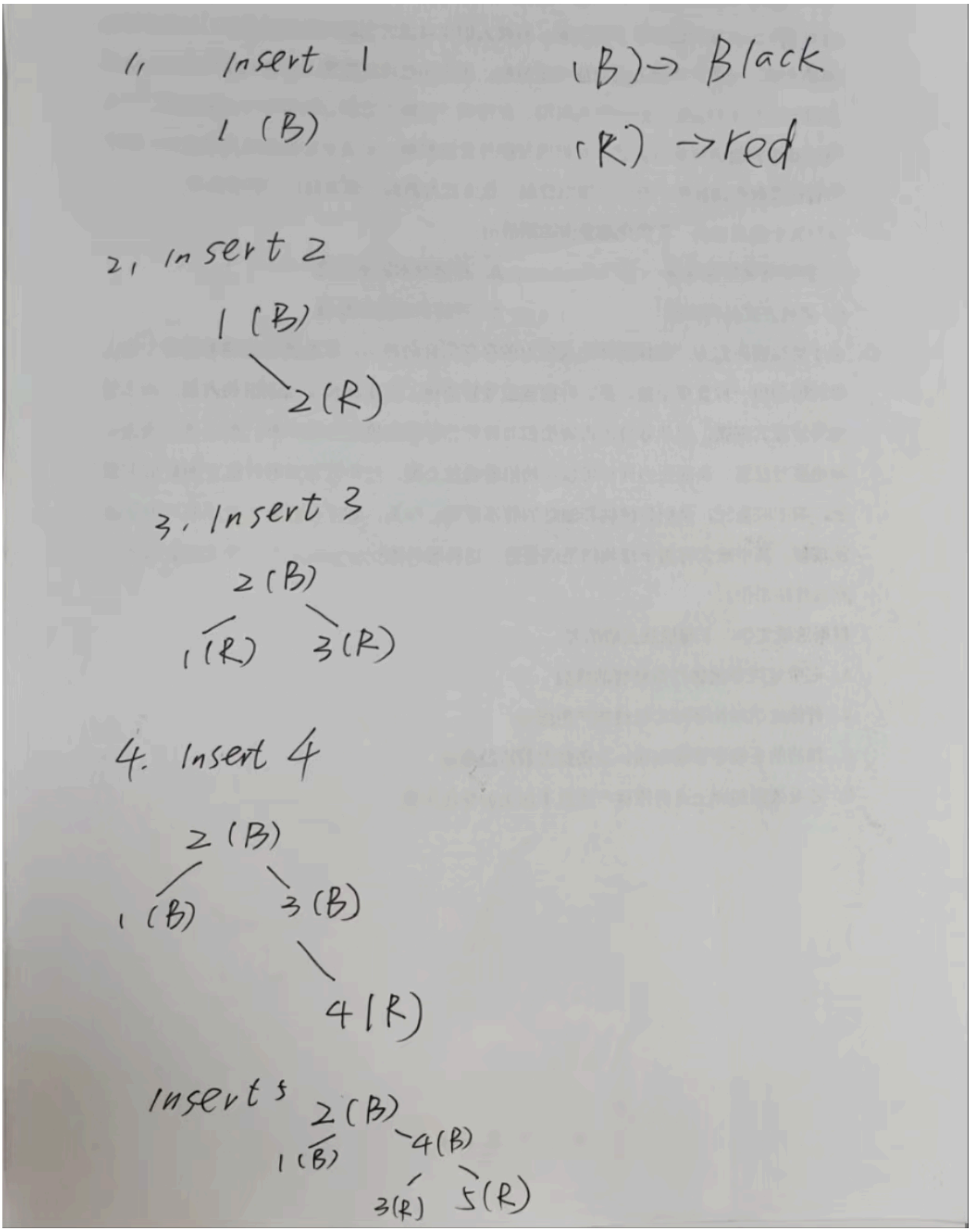
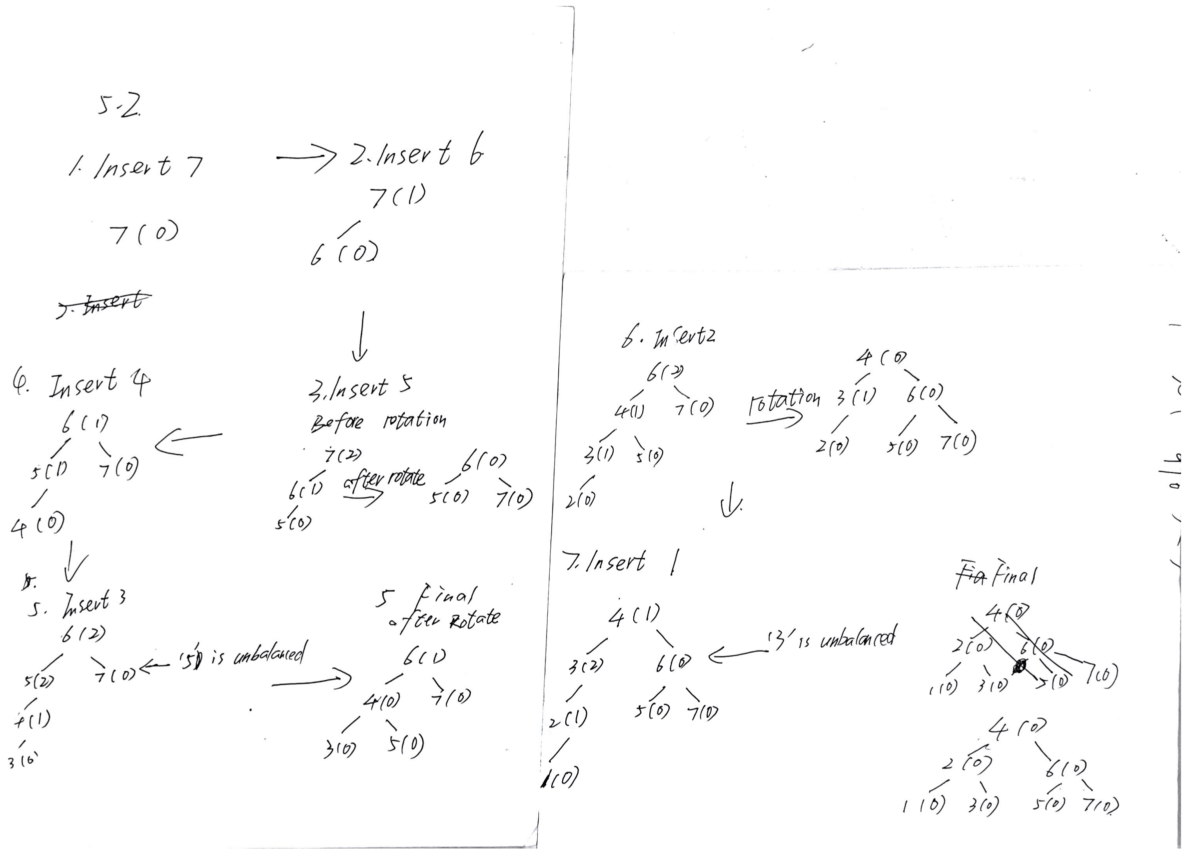
CA5

5.1


1, Insert 1
1 (B)
(B) → Black
(R) → red
2, Insert 2
1 (B)
2 (R)
3, Insert 3
2 (B)
1 (R)
3 (R)
4. Insert 4
2 (B)
1 (B)
3 (B)
4 (R)
insert 5
2 (B)
1 (B)
4 (B)
3 (R)
5 (R)

5.2



5.3

1. Initial Population Fitness and Selection Probabilities

First, we calculate the total fitness and the probabilities for the initial population specified in the assignment1. The fitness function is Fi=100/n2.

- **Total Fitness Calculation:** Total Fitness=4.35+14.29+4.17+10.00+3.45=36.26

The completed table is as follows3:

String No.	String	n	Fi=100/n	Probability (Fi/36.26)	Cumulative Probability
1	`10111`	23	4.35	0.120	**0.120**
2	`00111`	7	14.29	0.394	**0.514**

String No.	String	n	Fi=100/n	Probability (Fi/36.26)	Cumulative Probability
3	`11000`	24	4.17	0.115	**0.629**
4	`01010`	10	10.00	0.276	**0.905**
5	`11101`	29	3.45	0.095	**1.000**

---

2. Selection: Creating the Mating Pool

We generate 5 random numbers to simulate the roulette wheel selection 4 to create a mating pool of 5.

Spin	Random Number (Generated)	Selected Individual (based on Cumulative Probability)
1	0.813	**String 4 (** `01010` **)**
2	0.134	**String 2 (** `00111` **)**
3	0.495	**String 2 (** `00111` **)**
4	0.952	**String 5 (** `11101` **)**
5	0.076	**String 1 (** `10111` **)**

- The Mating Pool is
  1. `01010` (from String 4)
  2. `00111` (from String 2)
  3. `00111` (from String 2)
  4. `11101` (from String 5)
  5. `10111` (from String 1)

---

3. Crossover Operations

We pair individuals (1,2), (3,4), and (5,1) from the mating pool 6 and perform single-point crossover with a probability of 0.57.

- **Pair 1: (Individual 1: `01010`, Individual 2: `00111`)**
  - Random number for crossover check: 0.31 (≤ 0.5), so **Crossover Occurs**.
  - Random crossover point (1-4): 3.
  - Parents: `010|10` and `001|11`
  - **Offspring 1 & 2**: `01011` and `00110`
- **Pair 2: (Individual 3: `00111`, Individual 4: `11101`)**
  - Random number for crossover check: 0.74 (> 0.5), so **No Crossover**.

- Offspring 3 & 4

  : 00111

   and 11101

   (direct copies) 8
- **Pair 3: (Individual 5: `10111`, Individual 1: `01010`)**
  - Random number for crossover check: 0.45 (≤ 0.5), so **Crossover Occurs**.
  - Random crossover point (1-4): 2.
  - Parents: `10|111` and `01|010`
  - Offspring 5

    : 10010

     (The problem asks for a new population of 5, so we only need one offspring from the last pair)9

    .
- **New Population (Before Mutation)**10:

  1. `01011`
  2. `00110`
  3. `00111`
  4. `11101`
  5. `10010`

---

4. Mutation

Each of the 25 bits has a 0.02 probability of flipping11. We simulate this process and assume **one mutation occurs**.

- **Simulation**: The 8th bit (Individual 2, 3rd bit) is chosen for mutation.
- Individual 2 before mutation: `00110`
- Individual 2 **after mutation**: `00010`
- **Population after Mutation**
  1. `01011`
  2. `00010`
  3. `00111`
  4. `11101`
  5. `10010`

---

5. Final Evaluation and Comparison

We calculate the fitness for each individual in the new, mutated population to evaluate its performance. 1

New Individual String	n (decimal)	Fitness (Fi=100/n)
`01011`	11	9.09
`00010`	2	50.00
`00111`	7	14.29
`11101`	29	3.45
`10010`	18	5.56

Now, we compare the total and average fitness of the new population against the initial one. 2

- **New Population Stats**:
  - **Total Fitness**: 9.09+50.00+14.29+3.45+5.56=82.39
  - **Average Fitness**: 82.39÷5=16.48
- **Initial Population Stats**:
  - **Total Fitness**: 36.26
  - **Average Fitness**: 36.26÷5=7.25

Conclusion: Yes, the average fitness has improved significantly, increasing from **7.25** to **16.48**. The genetic algorithm successfully evolved the population towards better solutions.

- Fitness Comparison
  - **Initial Population:** Total Fitness = 36.26, Average Fitness = 36.26 / 5 = **7.25**
  - **New Population:** Total Fitness = 82.39, Average Fitness = 82.39 / 5 = **16.48**

Conclusion: Yes, the average fitness has significantly improved from 7.25 to 16.48. This is mainly due to the discovery of the highly fit individual `00010` (with n=2) through mutation.

5.4

1. Algorithmic Approach

To solve this, we use **dynamic programming**. The core of this method is to build a 2D table, `C`, where `C[i][j]` stores the length of the longest common subsequence between the first `i` elements of sequence `X` and the first `j` elements of sequence `Y`. 1 We also use arrows (↖, ↑, ←) to trace the path back to reconstruct the actual subsequence.

The rules for filling the table are:

- If `X[i]` and `Y[j]` match, the length increases, and we follow the diagonal arrow `↖`.
- If they do not match, we take the maximum length from the cell above or the cell to the left, and follow the corresponding arrow `↑` or `←`.

2. Dynamic Programming Table

The two sequences are:

- **X:** `(1, 0, 0, 1, 0, 1, 0, 1)`
- **Y:** `(0, 1, 0, 1, 1, 0, 1, 1, 0)`

The resulting `(8+1) x (9+1)` table is shown below. The value in the bottom-right cell, **6**, is the length of the LCS.

	j=0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
		Y: `0`	`1`	`0`	`1`	`1`	`0`	`1`	`1`	`0`
`i` **=0**	0	0	0	0	0	0	0	0	0	0
1 `X`:`1`	0	↑ 0	↖ **1**	← 1	↖ **1**	← 1	← 1	↖ **1**	← 1	← 1
2 `X`:`0`	0	↖ **1**	↑ 1	↖ **2**	← 2	← 2	↖ **2**	↑ 2	↑ 2	↖ **2**
3 `X`:`0`	0	↖ **1**	↑ 1	↖ **2**	↑ 2	↑ 2	↖ **3**	← 3	← 3	↖ **3**
4 `X`:`1`	0	↑ 1	↖ **2**	↑ 2	↖ **3**	← 3	↑ 3	↖ **4**	← 4	↑ 4
5 `X`:`0`	0	↖ **1**	↑ 2	↖ **3**	↑ 3	↑ 3	↖ **4**	↑ 4	↑ 4	↖ **5**
6 `X`:`1`	0	↑ 1	↖ **2**	↑ 3	↖ **4**	↑ 4	↑ 4	↖ **5**	← 5	↑ 5
7 `X`:`0`	0	↖ **1**	↑ 2	↖ **3**	↑ 4	↑ 4	↖ **5**	↑ 5	↑ 5	↖ **6**
8 `X`:`1`	0	↑ 1	↖ **2**	↑ 3	↖ **4**	↖ **5**	↑ 5	↖ **6**	← 6	↑ 6

---

3. Backtracking to Find the LCS

We trace the arrows back from the bottom-right cell `c[8,9]` to find the subsequence.

1. Start at `c[8,9]`: The arrow is `↑`. Move up to `c[7,9]`.
2. `c[7,9]`: The arrow is `↖`. This means `x[7]` (`0`) is part of the LCS. Move to `c[6,8]`.
3. `c[6,8]`: The arrow is `←`. Move left to `c[6,7]`.
4. `c[6,7]`: The arrow is `↖`. This means `x[6]` (`1`) is part of the LCS. Move to `c[5,6]`.
5. `c[5,6]`: The arrow is `↖`. This means `x[5]` (`0`) is part of the LCS. Move to `c[4,5]`.
6. `c[4,5]`: The arrow is `←`. Move left to `c[4,4]`.
7. `c[4,4]`: The arrow is `↖`. This means `x[4]` (`1`) is part of the LCS. Move to `c[3,3]`.
8. `c[3,3]`: The arrow is `↖`. This means `x[3]` (`0`) is part of the LCS. Move to `c[2,2]`.
9. `c[2,2]`: The arrow is `↑`. Move up to `c[1,2]`.
10. `c[1,2]`: The arrow is `↖`. This means `x[1]` (`1`) is part of the LCS. Move to `c[0,1]`.
11. We have reached the first row, so the process ends.

Reversing the elements we collected gives us the final result.

- **Longest Common Subsequence (LCS)**: `(1, 0, 1, 0, 1, 0)`

5.5

---

1. Core Idea

The dynamic programming approach solves the 0-1 knapsack problem by breaking it down into smaller, overlapping subproblems. A table is constructed bottom-up to store the optimal solutions for all subproblems, where a subproblem is defined by considering a subset of items and a smaller knapsack capacity. The solution to a larger problem is then derived from the already computed solutions of its subproblems.

---

2. State Definition

We define a two-dimensional array, `dp[i][w]`, to represent the state of a subproblem:

- **`dp[i][w]`**: The maximum value that can be achieved using the first `i` items (from item 1 to item `i`) with a maximum knapsack capacity of `w`.

Let `n` be the total number of items and `W` be the total capacity of the knapsack. Our goal is to find `dp[n][W]`.

---

3. State Transition Equation

When considering the `i`-th item (with value vi and weight wi), we have two choices for each capacity `w`:

1. **Do not include item `i`**: If we do not take the `i`-th item, the maximum value is simply the best we could achieve with the first `i-1` items and the same capacity `w`. The value would be `dp[i-1][w]`.
2. **Include item `i`**: This is only possible if the weight of the item, wi, does not exceed the current capacity `w` (i.e., wi≤w). If we take the item, its value vi is added to the maximum value that could be achieved with the remaining capacity (w−wi) using the first `i-1` items. The value would be vi+dp[i−1][w−wi].

The state transition equation combines these two choices by taking the maximum:

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{if } w_i > w \\ \max(dp[i-1][w], v_i + dp[i-1][w-w_i]) & \text{if } w_i \leq w \end{cases}$$

---

4. Table Construction

The dynamic programming table, `dp`, of size `(n+1) x (W+1)` is constructed as follows:

1. **Initialization**: The first row and the first column are initialized to 0. `dp[0][w] = 0` for all `w` from 0 to `W` (no value if there are no items), and `dp[i][0] = 0` for all `i` from 0 to `n` (no value if knapsack capacity is 0).
2. **Iteration**: The table is filled iteratively. We loop from `i = 1` to `n` (for each item) and, within that, we loop from `w = 1` to `W` (for each capacity). In each step, `dp[i][w]` is computed using the state transition equation.
3. **Result**: The final answer, which is the maximum possible value, is located at `dp[n][W]`.

This algorithm has a time complexity of **O(n·W)** because it involves filling an `n x W` table, and each entry's computation takes constant time.

5.6

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ThreadLocalRandom;

/**
 * Main class to execute the Genetic Algorithm as described in Question 5.3.
 * It handles selection, crossover, mutation, and evaluation.
 */
public class GeneticAlgorithm {

    private static final double CROSSOVER_PROBABILITY = 0.5; //
```

```
    private static final double MUTATION_PROBABILITY = 0.02; //
    private static final int POPULATION_SIZE = 5;

    public static void main(String[] args) {
        // 1. INITIAL POPULATION
        List<Individual> initialPopulation = new ArrayList<>();
        initialPopulation.add(new Individual("10111"));
        initialPopulation.add(new Individual("00111"));
        initialPopulation.add(new Individual("11000"));
        initialPopulation.add(new Individual("01010"));
        initialPopulation.add(new Individual("11101"));

        System.out.println("--- Initial Population ---");
        printPopulationDetails(initialPopulation);

        // 2. SELECTION (Roulette Wheel)
        List<Individual> matingPool = performSelection(initialPopulation);
        System.out.println("\n--- Mating Pool (from Roulette Wheel Selection) ---
"); //
        for (int i = 0; i < matingPool.size(); i++) {
            System.out.printf("Mating Pool Slot %d: %s\n", i + 1,
matingPool.get(i));
        }

        // 3. CROSSOVER
        List<Individual> newPopulation = performCrossover(matingPool);
        System.out.println("\n--- New Population (After Crossover) ---"); //
        printPopulationDetails(newPopulation);

        // 4. MUTATION
        for (Individual individual : newPopulation) {
            individual.mutate(MUTATION_PROBABILITY);
        }
        System.out.println("\n--- Final Population (After Mutation) ---"); //
        printPopulationDetails(newPopulation);

        // 5. FINAL EVALUATION & COMPARISON
        System.out.println("\n--- Fitness Comparison ---");
        double initialAvgFitness = calculateAverageFitness(initialPopulation);
        double finalAvgFitness = calculateAverageFitness(newPopulation);
        System.out.printf("Initial Average Fitness: %.2f\n", initialAvgFitness);
        System.out.printf("Final Average Fitness:   %.2f\n", finalAvgFitness);

        if (finalAvgFitness > initialAvgFitness) {
            System.out.println("Conclusion: The average fitness has improved.");
        } else if (finalAvgFitness < initialAvgFitness) {
            System.out.println("Conclusion: The average fitness has not
improved.");
        } else {
            System.out.println("Conclusion: The average fitness remained the
same.");
        }
    }

    /**
     * Performs Roulette Wheel Selection to create a mating pool.
```

```
     */
    public static List<Individual> performSelection(List<Individual> population)
{
        double totalFitness = 0;
        for (Individual individual : population) {
            totalFitness += individual.getFitness();
        }

        System.out.printf("\nTotal Fitness of Initial Population: %.2f\n\n",
totalFitness); //
        System.out.println("--- Probabilities for Selection ---"); //
        System.out.println("String No. | Fi    | Probability | Cumulative
Probability");
        System.out.println("-----------|-------|-------------|-------------------
----");

        double cumulativeProb = 0;
        for (int i = 0; i < population.size(); i++) {
            Individual ind = population.get(i);
            ind.selectionProbability = ind.getFitness() / totalFitness;
            cumulativeProb += ind.selectionProbability;
            ind.cumulativeProbability = cumulativeProb;
            System.out.printf("%-10d | %-5.2f | %-11.2f | %.2f\n",
                    i + 1, ind.getFitness(), ind.selectionProbability,
ind.cumulativeProbability);
        }

        // Generate mating pool with replacement
        List<Individual> matingPool = new ArrayList<>();
        for (int i = 0; i < POPULATION_SIZE; i++) {
            double random = ThreadLocalRandom.current().nextDouble();
            for (Individual individual : population) {
                if (random <= individual.cumulativeProbability) {
                    // Add a copy to avoid modifying original population
individuals
                    matingPool.add(new Individual(individual));
                    break;
                }
            }
        }
        return matingPool;
    }

    /**
     * Performs single-point crossover on the mating pool.
     */
    public static List<Individual> performCrossover(List<Individual> matingPool)
{
        List<Individual> newPopulation = new ArrayList<>();

        // Pair individuals: (1,2), (3,4), (5,1) as per the example
        int[][] pairs = {{0, 1}, {2, 3}, {4, 0}};

        for (int[] pair : pairs) {
            Individual parent1 = matingPool.get(pair[0]);
            Individual parent2 = matingPool.get(pair[1]);
```

```
            Individual offspring1 = new Individual(parent1);
            Individual offspring2 = new Individual(parent2);

            // Check if crossover occurs with a 0.5 probability
            if (ThreadLocalRandom.current().nextDouble() <=
CROSSOVER_PROBABILITY) {
                // Choose a random crossover point between 1 and 4
                int crossoverPoint = ThreadLocalRandom.current().nextInt(1, 5);

                String p1_part1 = parent1.getBinaryString().substring(0,
crossoverPoint);
                String p1_part2 =
parent1.getBinaryString().substring(crossoverPoint);
                String p2_part1 = parent2.getBinaryString().substring(0,
crossoverPoint);
                String p2_part2 =
parent2.getBinaryString().substring(crossoverPoint);

                offspring1 = new Individual(p1_part1 + p2_part2);
                offspring2 = new Individual(p2_part1 + p1_part2);
            }
            // If crossover does not occur, offspring are copies of parents

            newPopulation.add(offspring1);
            newPopulation.add(offspring2);
        }

        // The pairing produces 6 offspring. We truncate to the required
population size of 5.
        return new ArrayList<>(newPopulation.subList(0, POPULATION_SIZE));
    }

    /**
     * Helper function to print details of a population.
     */
    public static void printPopulationDetails(List<Individual> population) {
        for (int i = 0; i < population.size(); i++) {
            System.out.printf("Individual %d: %s\n", i + 1, population.get(i));
        }
    }

    /**
     * Helper function to calculate the average fitness of a population.
     */
    public static double calculateAverageFitness(List<Individual> population) {
        if (population.isEmpty()) {
            return 0;
        }
        double totalFitness = 0;
        for (Individual individual : population) {
            totalFitness += individual.getFitness();
        }
        return totalFitness / population.size();
    }
}
```

```java
import java.util.concurrent.ThreadLocalRandom;

/**
 * Represents a single individual in the population for the Genetic Algorithm.
 * It holds the binary string representation, its decimal equivalent, and its
fitness score.
 */
public class Individual {
    private String binaryString;
    private int decimalValue;
    private double fitness;
    public double selectionProbability;
    public double cumulativeProbability;

    /**
     * Constructor for creating an Individual.
     * @param binaryString The 5-bit binary string for this individual.
     */
    public Individual(String binaryString) {
        this.binaryString = binaryString;
        this.calculateDecimalValue();
        this.calculateFitness(); // Fitness is calculated as Fi = 100/n
    }

    /**
     * Copy constructor to create a new Individual instance from another.
     */
    public Individual(Individual source) {
        this.binaryString = source.binaryString;
        this.decimalValue = source.decimalValue;
        this.fitness = source.fitness;
        this.selectionProbability = source.selectionProbability;
        this.cumulativeProbability = source.cumulativeProbability;
    }

    /**
     * Calculates the decimal value 'n' from the binary string.
     */
    public void calculateDecimalValue() {
        // Handles cases where the binary string might be invalid after mutation
        try {
            this.decimalValue = Integer.parseInt(this.binaryString, 2);
        } catch (NumberFormatException e) {
            this.decimalValue = 0; // Assign a default/error value
        }
    }

    /**
     * Calculates the fitness Fi = 100/n for the individual.
     * If the decimal value is 0, fitness is set to 0 to avoid division by zero.
     */
    public void calculateFitness() {
        if (this.decimalValue == 0) {
            this.fitness = 0;
```

```java
        } else {
            this.fitness = 100.0 / this.decimalValue;
        }
    }

    /**
     * Mutates the individual's binary string based on a given probability.
     * For each bit, it flips the value (0 to 1 or 1 to 0) if a random
     * number is less than or equal to the mutation rate.
     * @param mutationRate The probability of a single bit mutating.
     */
    public void mutate(double mutationRate) {
        StringBuilder mutatedString = new StringBuilder(this.binaryString);
        for (int i = 0; i < mutatedString.length(); i++) {
            if (ThreadLocalRandom.current().nextDouble() <= mutationRate) {
                mutatedString.setCharAt(i, mutatedString.charAt(i) == '0' ? '1' :
'0');
            }
        }
        this.binaryString = mutatedString.toString();
        // Recalculate values after mutation
        this.calculateDecimalValue();
        this.calculateFitness();
    }

    // Standard getters
    public String getBinaryString() {
        return binaryString;
    }

    public int getDecimalValue() {
        return decimalValue;
    }

    public double getFitness() {
        return fitness;
    }

    @Override
    public String toString() {
        return String.format("String: %s, n: %2d, Fitness (Fi): %.2f",
                binaryString, decimalValue, fitness);
    }
}
```

```
C:\Users\LENOVO\.jdks\corretto-17.0.9\bin\java.exe "-javaagent:D:\Program Files\JetBrains\IntelliJ IDEA 2025.1.2\lib\idea_rt.
--- Initial Population ---
Individual 1: String: 10111, n: 23, Fitness (Fi): 4.35
Individual 2: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 3: String: 11000, n: 24, Fitness (Fi): 4.17
Individual 4: String: 01010, n: 10, Fitness (Fi): 10.00
Individual 5: String: 11101, n: 29, Fitness (Fi): 3.45

Total Fitness of Initial Population: 36.25

--- Probabilities for Selection ---
String No. | Fi    | Probability | Cumulative Probability
-----------|-------|-------------|----------------------
1          | 4.35  | 0.12        | 0.12
2          | 14.29 | 0.39        | 0.51
3          | 4.17  | 0.11        | 0.63
4          | 10.00 | 0.28        | 0.90
5          | 3.45  | 0.10        | 1.00

--- Mating Pool (from Roulette Wheel Selection) ---
Mating Pool Slot 1: String: 00111, n:  7, Fitness (Fi): 14.29
Mating Pool Slot 2: String: 00111, n:  7, Fitness (Fi): 14.29
Mating Pool Slot 3: String: 01010, n: 10, Fitness (Fi): 10.00
Mating Pool Slot 4: String: 00111, n:  7, Fitness (Fi): 14.29
Mating Pool Slot 5: String: 01010, n: 10, Fitness (Fi): 10.00

--- New Population (After Crossover) ---
Individual 1: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 2: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 3: String: 01010, n: 10, Fitness (Fi): 10.00
Individual 4: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 5: String: 01010, n: 10, Fitness (Fi): 10.00

--- Final Population (After Mutation) ---
Individual 1: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 2: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 3: String: 01010, n: 10, Fitness (Fi): 10.00
```


```
--- New Population (After Crossover) ---
Individual 1: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 2: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 3: String: 01010, n: 10, Fitness (Fi): 10.00
Individual 4: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 5: String: 01010, n: 10, Fitness (Fi): 10.00

--- Final Population (After Mutation) ---
Individual 1: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 2: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 3: String: 01010, n: 10, Fitness (Fi): 10.00
```


```
Individual 4: String: 00111, n:  7, Fitness (Fi): 14.29
Individual 5: String: 01010, n: 10, Fitness (Fi): 10.00

--- Fitness Comparison ---
Initial Average Fitness: 7.25
Final Average Fitness:   12.57
Conclusion: The average fitness has improved.

Process finished with exit code 0
```