UI – Zadanie 2(b)

Hredzhev Oleksandr

ID: 123560

Problem

The task of finding treasures using a genetic algorithm requires the creation of several generations and their peculiar evolution. To solve this problem, you can use many methods of selecting the next generation, crossing parents and mutations.

In my case, I used **2 selection** algorithms, **1 crossing** method and **2 mutation** methods:

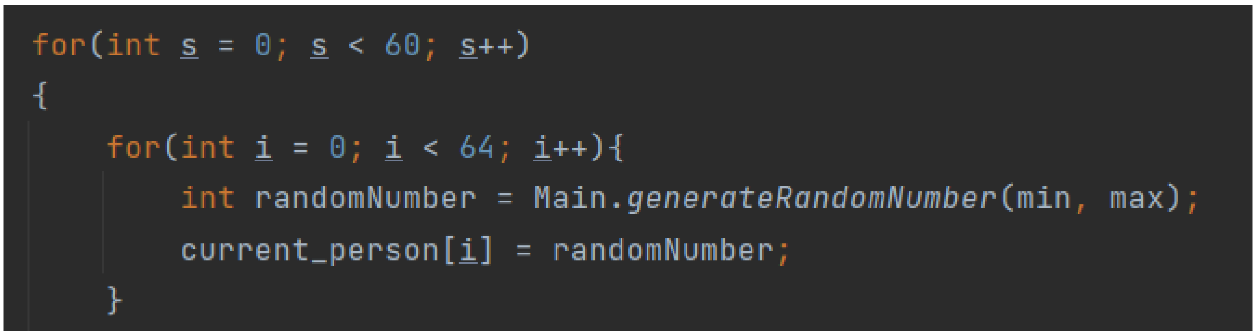
*I use 2 selection methods at once. ***Tournament*** and ***Roulette*** bet method (with a chance of 50/50, respectively)

*Crossing method - cutting the array at ***2 points*** and **replacing** these parts in places

* I also use the mutation method of replacing a random gene with a ***random number*** within an acceptable range and the gene ***inversion*** method (mutation probability is 10 percent (5/5 respectively)

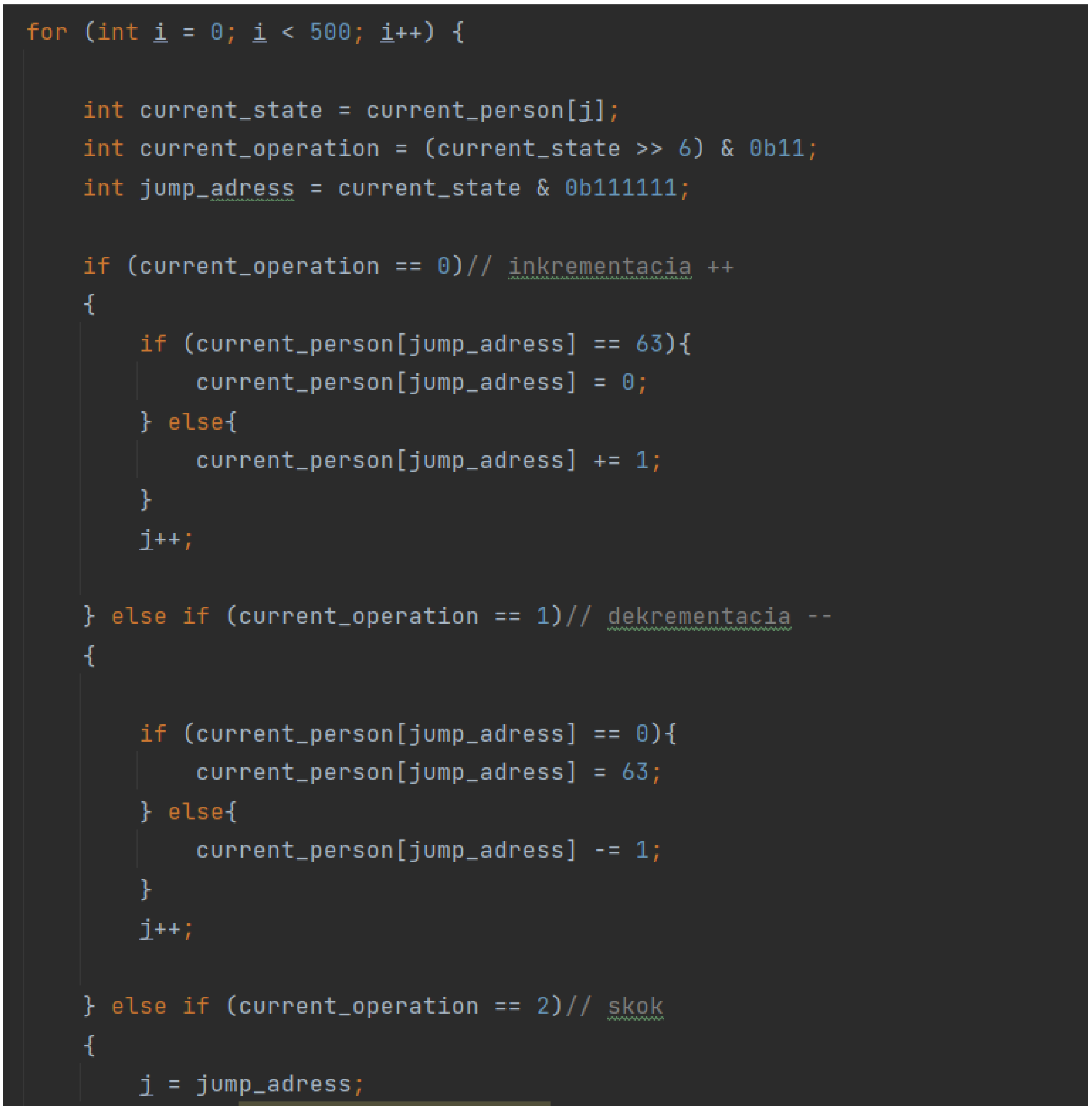
The creation of the first generation occurs by filling the array (person) with ***randomly generated values* in the acceptable range

Below I will show how these algorithms are implemented in code:

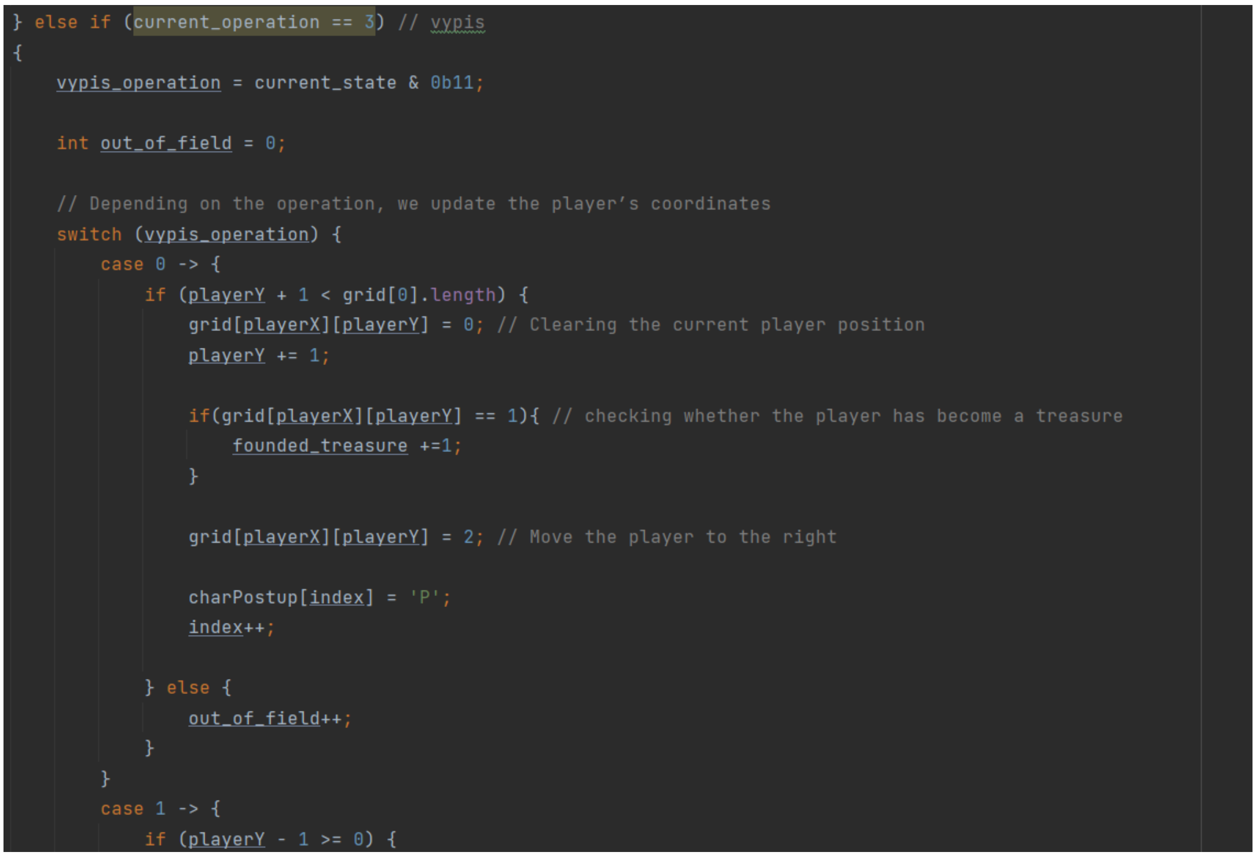


```
for(int s = 0; s < 60; s++)
{
    for(int i = 0; i < 64; i++){
        int randomNumber = Main.generateRandomNumber(min, max);
        current_person[i] = randomNumber;
    }
```

This is how the first generation is created

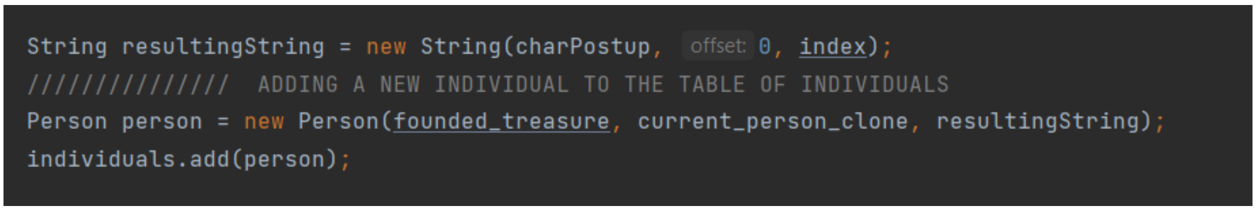


```
for (int i = 0; i < 500; i++) {

    int current_state = current_person[j];
    int current_operation = (current_state >> 6) & 0b11;
    int jump_adress = current_state & 0b111111;

    if (current_operation == 0)// inkrementacia ++
    {
        if (current_person[jump_adress] == 63){
            current_person[jump_adress] = 0;
        } else{
            current_person[jump_adress] += 1;
        }
        j++;

    } else if (current_operation == 1)// dekrementacia --
    {

        if (current_person[jump_adress] == 0){
            current_person[jump_adress] = 63;
        } else{
            current_person[jump_adress] -= 1;
        }
        j++;

    } else if (current_operation == 2)// skok
    {
        j = jump_adress;
```



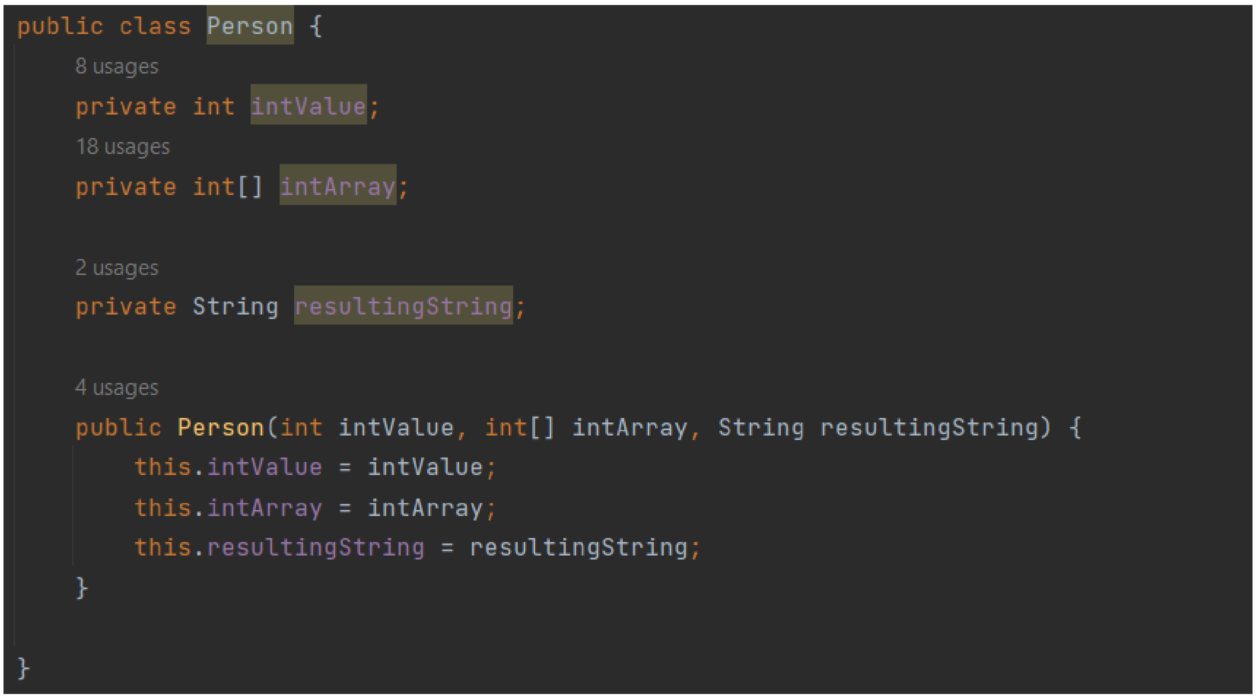
```
} else if (current_operation == 3) // vypis
{
    vypis_operation = current_state & 0b11;

    int out_of_field = 0;

    // Depending on the operation, we update the player's coordinates
    switch (vypis_operation) {
        case 0 -> {
            if (playerY + 1 < grid[0].length) {
                grid[playerX][playerY] = 0; // Clearing the current player position
                playerY += 1;

                if(grid[playerX][playerY] == 1){ // checking whether the player has become a treasure
                    founded_treasure +=1;
                }

                grid[playerX][playerY] = 2; // Move the player to the right

                charPostup[index] = 'P';
                index++;

            } else {
                out_of_field++;
            }
        }
        case 1 -> {
            if (playerY - 1 >= 0) {
```

This is how a sequence of numbers is processed (man). Most of the code is shown, the rest works similarly



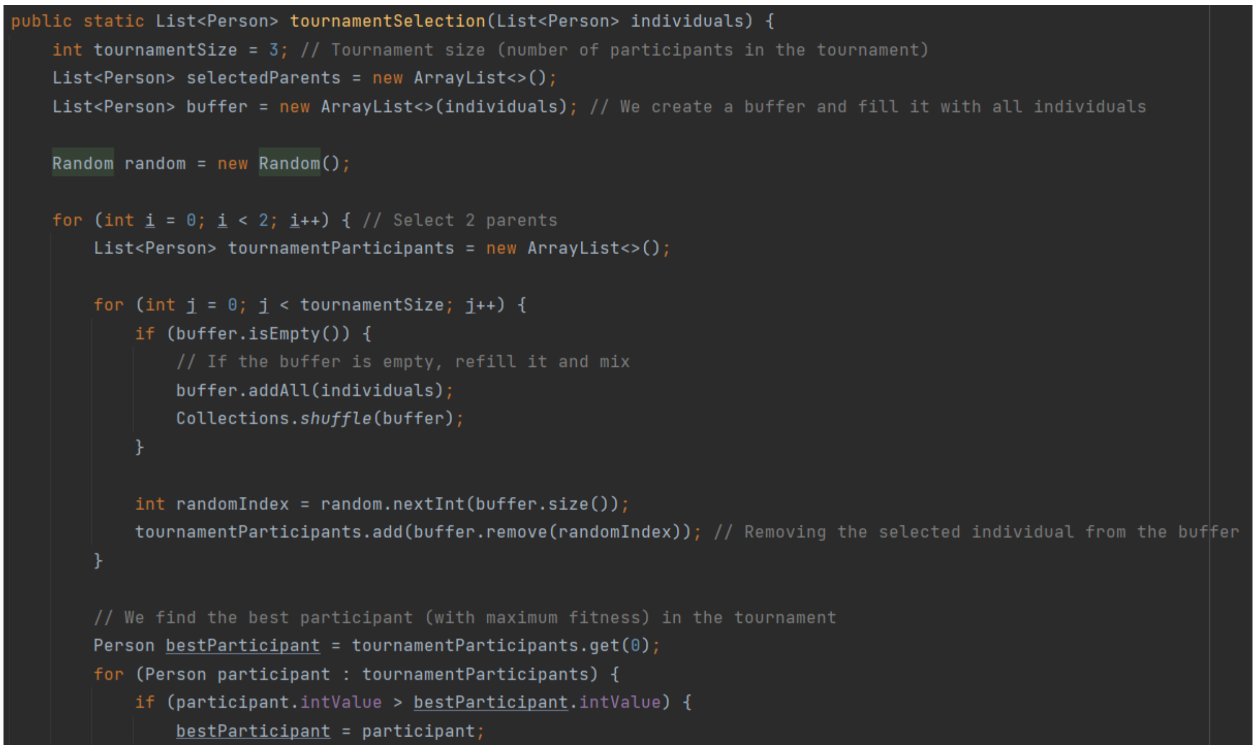
```
String resultingString = new String(charPostup,  offset: 0, index);
//////////////  ADDING A NEW INDIVIDUAL TO THE TABLE OF INDIVIDUALS
Person person = new Person(founded_treasure, current_person_clone, resultingString);
individuals.add(person);
```

Adding a processed person to the list for further selection



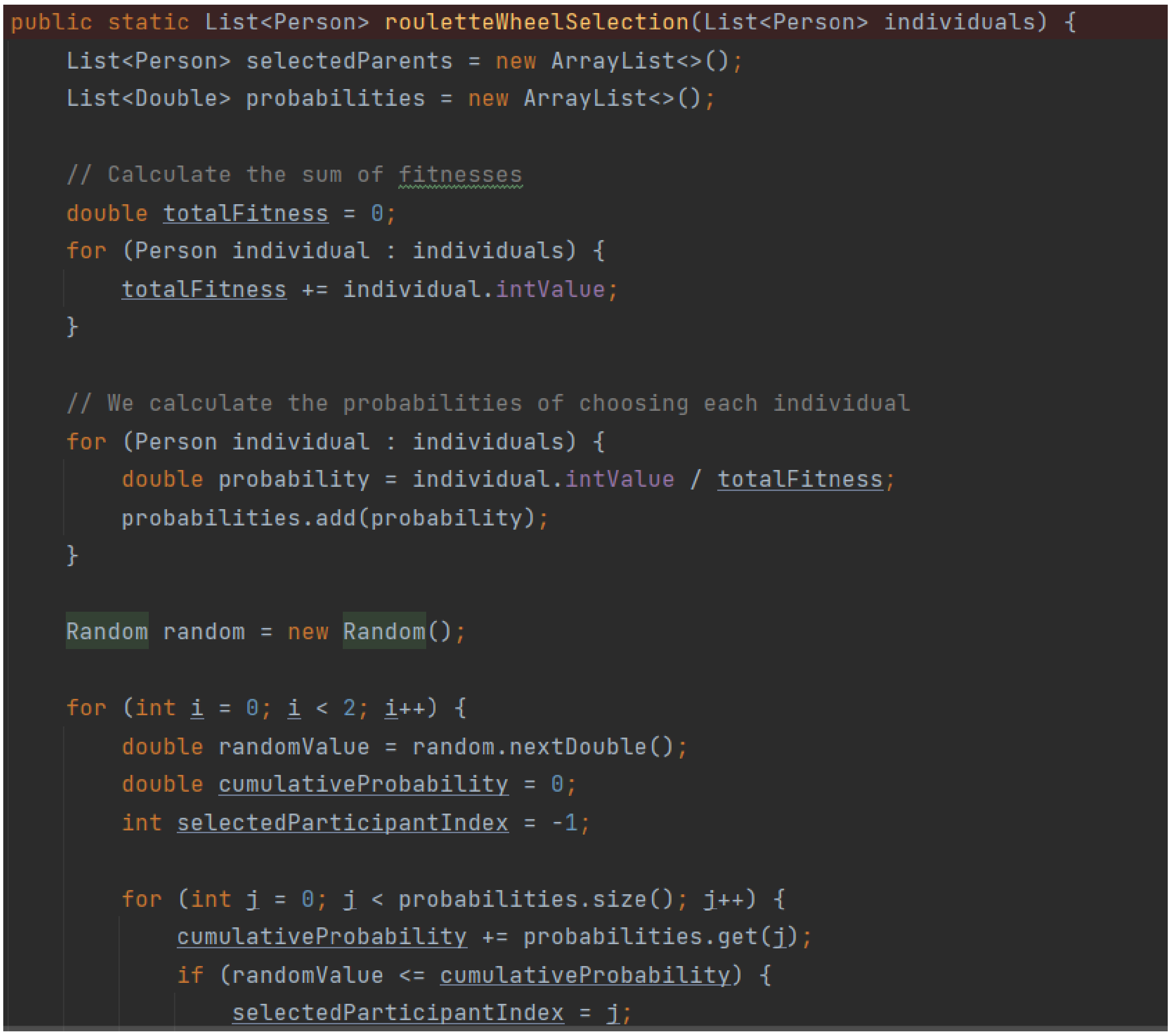
```java
public class Person {
    8 usages
    private int intValue;
    18 usages
    private int[] intArray;

    2 usages
    private String resultingString;

    4 usages
    public Person(int intValue, int[] intArray, String resultingString) {
        this.intValue = intValue;
        this.intArray = intArray;
        this.resultingString = resultingString;
    }

}
```

Human structure in my code



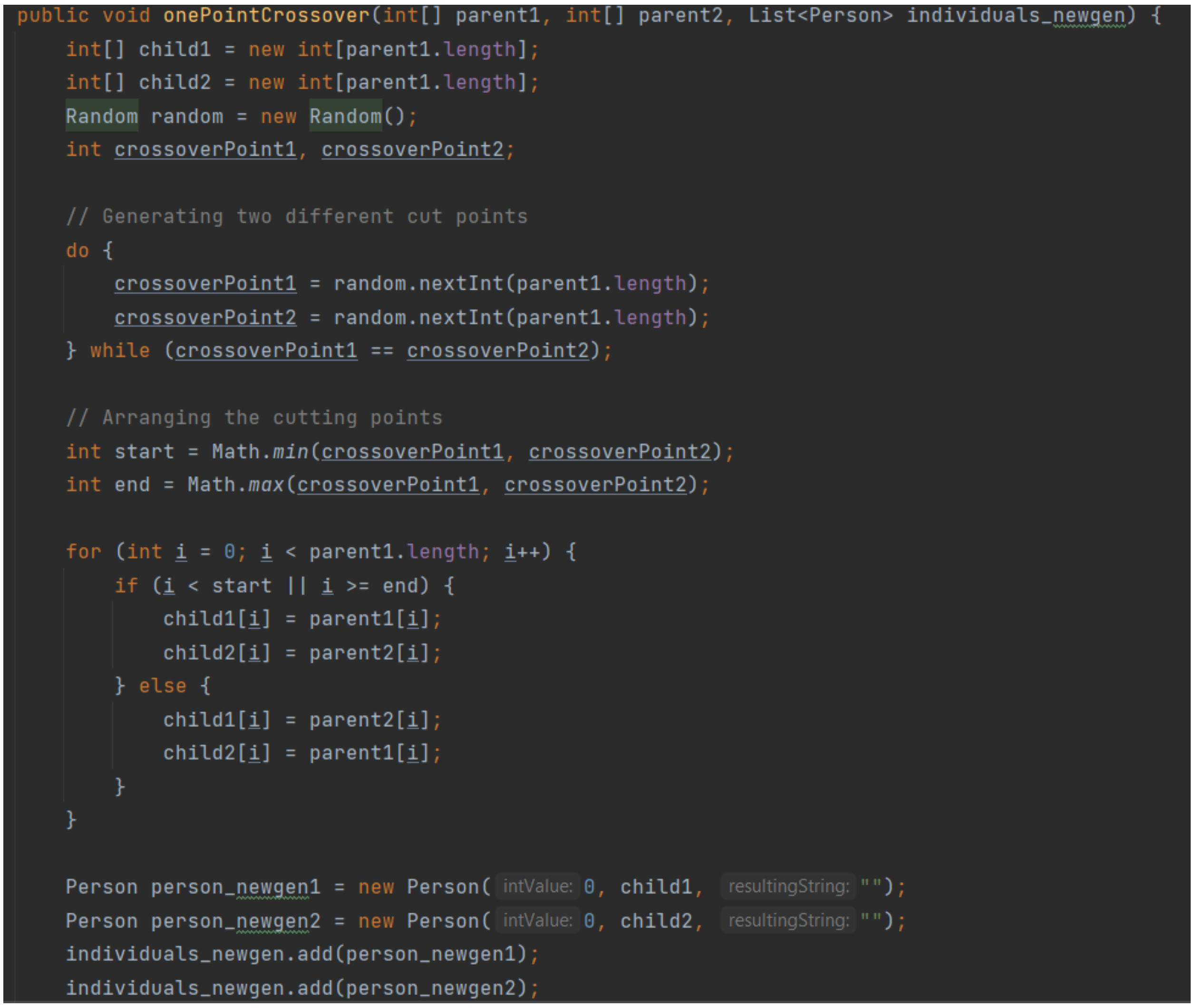
```java
public static List<Person> tournamentSelection(List<Person> individuals) {
    int tournamentSize = 3; // Tournament size (number of participants in the tournament)
    List<Person> selectedParents = new ArrayList<>();
    List<Person> buffer = new ArrayList<>(individuals); // We create a buffer and fill it with all individuals

    Random random = new Random();

    for (int i = 0; i < 2; i++) { // Select 2 parents
        List<Person> tournamentParticipants = new ArrayList<>();

        for (int j = 0; j < tournamentSize; j++) {
            if (buffer.isEmpty()) {
                // If the buffer is empty, refill it and mix
                buffer.addAll(individuals);
                Collections.shuffle(buffer);
            }

            int randomIndex = random.nextInt(buffer.size());
            tournamentParticipants.add(buffer.remove(randomIndex)); // Removing the selected individual from the buffer
        }

        // We find the best participant (with maximum fitness) in the tournament
        Person bestParticipant = tournamentParticipants.get(0);
        for (Person participant : tournamentParticipants) {
            if (participant.intValue > bestParticipant.intValue) {
                bestParticipant = participant;
```

Part of the tournament selection code



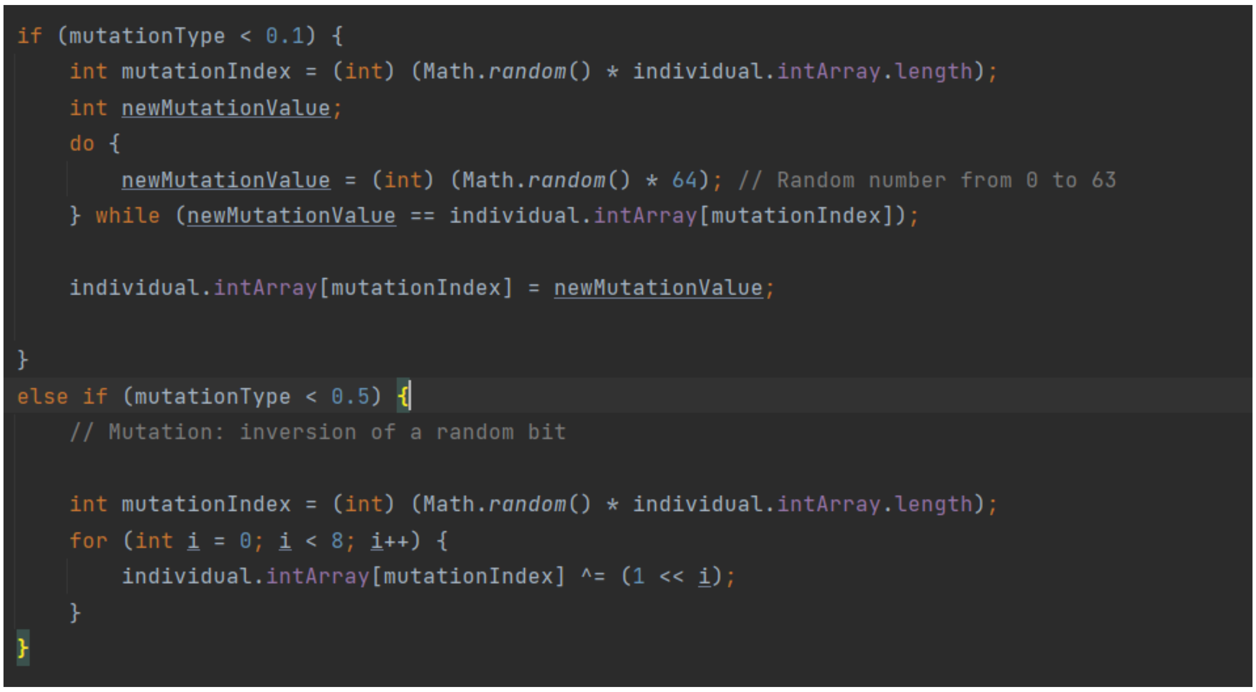
```java
public static List<Person> rouletteWheelSelection(List<Person> individuals) {
    List<Person> selectedParents = new ArrayList<>();
    List<Double> probabilities = new ArrayList<>();

    // Calculate the sum of fitnesses
    double totalFitness = 0;
    for (Person individual : individuals) {
        totalFitness += individual.intValue;
    }

    // We calculate the probabilities of choosing each individual
    for (Person individual : individuals) {
        double probability = individual.intValue / totalFitness;
        probabilities.add(probability);
    }

    Random random = new Random();

    for (int i = 0; i < 2; i++) {
        double randomValue = random.nextDouble();
        double cumulativeProbability = 0;
        int selectedParticipantIndex = -1;

        for (int j = 0; j < probabilities.size(); j++) {
            cumulativeProbability += probabilities.get(j);
            if (randomValue <= cumulativeProbability) {
                selectedParticipantIndex = j;
```

Part of the roulette selection code



```java
public void onePointCrossover(int[] parent1, int[] parent2, List<Person> individuals_newgen) {
    int[] child1 = new int[parent1.length];
    int[] child2 = new int[parent1.length];
    Random random = new Random();
    int crossoverPoint1, crossoverPoint2;

    // Generating two different cut points
    do {
        crossoverPoint1 = random.nextInt(parent1.length);
        crossoverPoint2 = random.nextInt(parent1.length);
    } while (crossoverPoint1 == crossoverPoint2);

    // Arranging the cutting points
    int start = Math.min(crossoverPoint1, crossoverPoint2);
    int end = Math.max(crossoverPoint1, crossoverPoint2);

    for (int i = 0; i < parent1.length; i++) {
        if (i < start || i >= end) {
            child1[i] = parent1[i];
            child2[i] = parent2[i];
        } else {
            child1[i] = parent2[i];
            child2[i] = parent1[i];
        }
    }

    Person person_newgen1 = new Person( intValue: 0, child1,  resultingString: "");
    Person person_newgen2 = new Person( intValue: 0, child2,  resultingString: "");
    individuals_newgen.add(person_newgen1);
    individuals_newgen.add(person_newgen2);
```

crossing mechanism



```
if (mutationType < 0.1) {
    int mutationIndex = (int) (Math.random() * individual.intArray.length);
    int newMutationValue;
    do {
        newMutationValue = (int) (Math.random() * 64); // Random number from 0 to 63
    } while (newMutationValue == individual.intArray[mutationIndex]);

    individual.intArray[mutationIndex] = newMutationValue;

}
else if (mutationType < 0.5) {
    // Mutation: inversion of a random bit

    int mutationIndex = (int) (Math.random() * individual.intArray.length);
    for (int i = 0; i < 8; i++) {
        individual.intArray[mutationIndex] ^= (1 << i);
    }
}
```

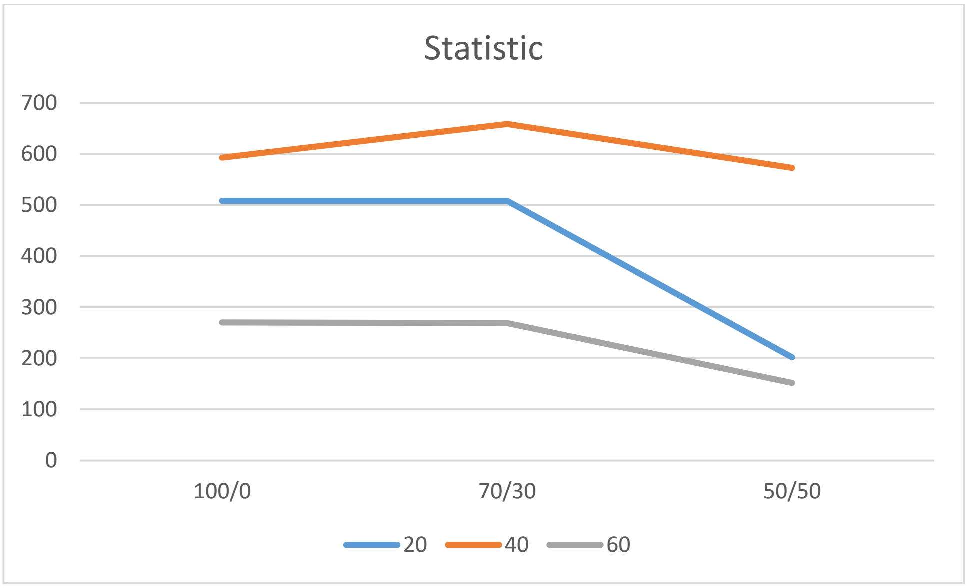
Mutations

Results and statistics

**When testing the roulette method, the results turned out to be quite bad, so there is no point in presenting its statistics. I decided to use the tournament method, which worked well alone, and I added the roulette method as a more diverse selection of people.

**The mutation has noticeably improved treasure hunting compared to its absence, but since this is a prerequisite for evolution, there is no point in citing such statistics. There was no significant difference between the mutation methods that I used in terms of creating more than 30 generations, so their separate comparison does not make sense.

↓population///method(tournament/roulette)→	100/0	70/30	50/50
20	508	508	202
40	593	659	573
60	270	269	152



Speculation about my algorithm

***Unfortunately, my mechanism of evolution is quite dependent on chance, since this is how the first generation is created, on which all subsequent ones depend. To improve my program, I believe that it might be rational to add selection not only by the number of treasures found, but also by the shortest path, although this method would slow down the program and could not always lead to a positive result.