12/06/2018

Knapsack

Implemented using Genetic Algorithms

Karan Karan(NUID-001449291)
Ritesh Manek(NUID-001494692)
PROJECT TEAM- INFO6205_303

Table of Content

Knapsack using Genetic Algorithm

1. Introduction

A genetic algorithm is a type of search algorithm that mimics Charles Darwin’s theory of evolution. Darwin stated that through natural selection, the most fit individuals in an environment survive and pass their traits on to future generations while the least fit individuals die away along with their traits. It’s important to note that each individual’s fitness level is determined by its environment. Polar bears are more “fit” in the arctic for example because their white fur allows them to blend in with the snow and catch their prey more easily, while brown bears are spotted easily and are unable to catch any food. In terms of genetic algorithms, an “environment” is any kind of problem that the algorithm needs to solve. So if a genetic algorithm is being applied to finding a solution to $x + 5 = 10$, then $x + 5 = 10$ is the algorithm’s “environment”. Individuals in a genetic algorithm are possible solutions to the problem at hand. In the case of finding solutions to $x + 5 = 10$, all possible x values are possible solutions to the problem, although most options are not true solutions.

To begin, the genetic algorithm randomly generates some number of potential solutions to the problem we are trying to solve. In this paper we call these solutions “individuals”. A genetic algorithm loops through until a termination condition is met, hopefully resulting with one of the solutions being the solution to the problem. As the algorithm loops through, it compares the fitness function’s results with other individuals. Individuals that are seen as more favourable go through a reproduction process to create a new generation of individuals. This is done through recombination where a new potential solution is generated by selecting a combination of elements from both parent solutions or through mutating a single individual. The different techniques used for the selection process are crossover, mutation, and reproduction. These processes will be discussed more thoroughly later on in this paper. To provide an understanding of each process that a genetic algorithm undertakes, we will use a simple example of a genetic algorithm applied to a knapsack problem following the steps detailed in the image below.

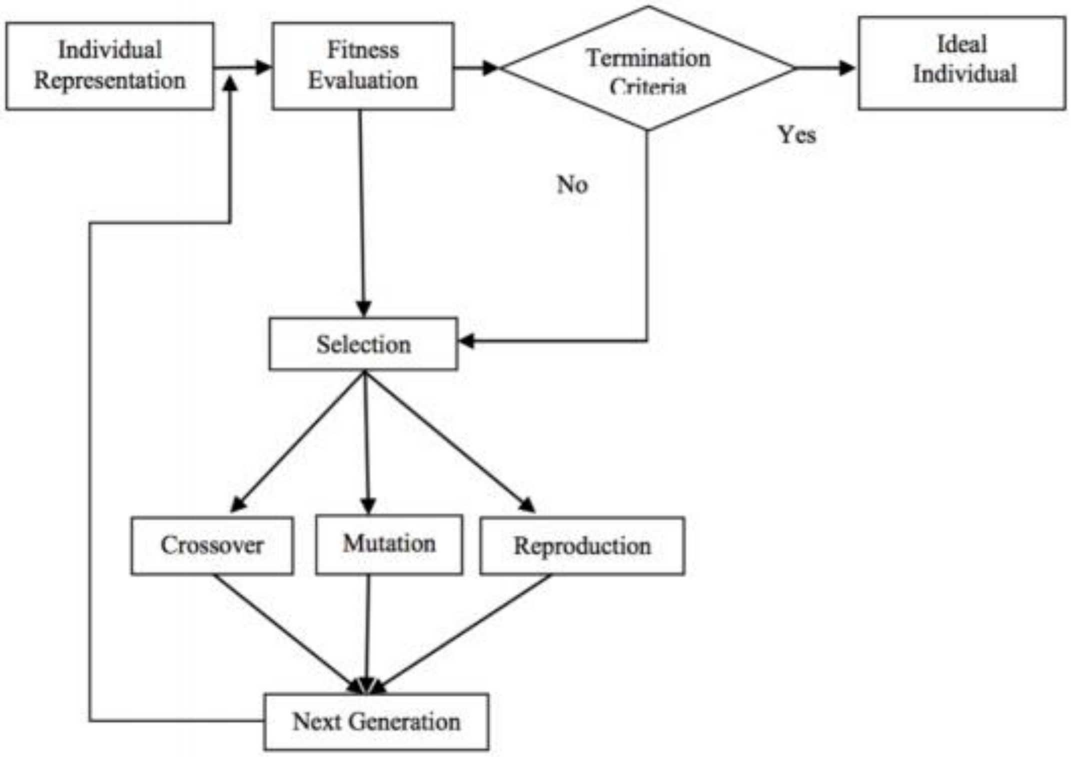
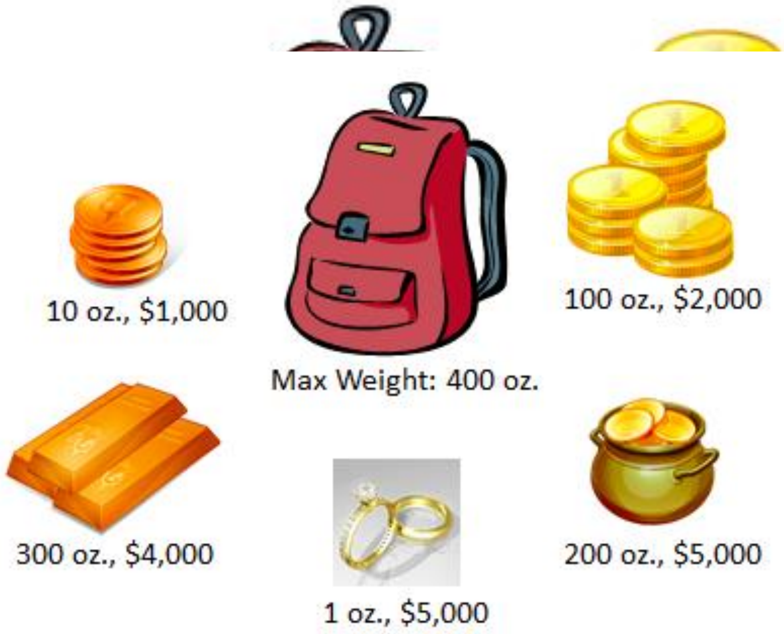


Fig 1: Genetic Algorithm Process

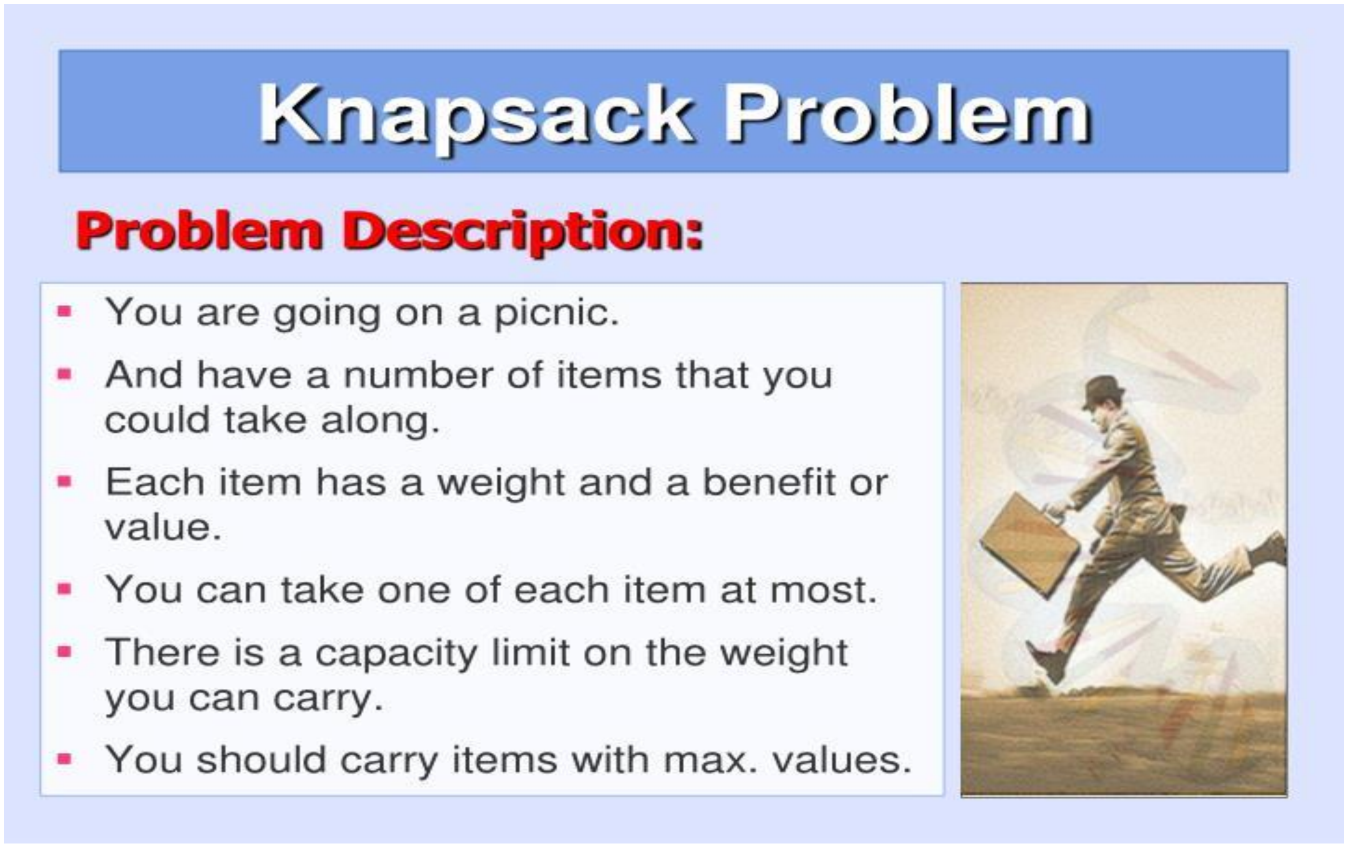
2. Problem Statement

The knapsack problem or rucksack problem is a problem in combinatorial optimization Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.



Imagine a thief with a knapsack who wants to maximize his earnings…

Interesting way to understand Knapsack problem.



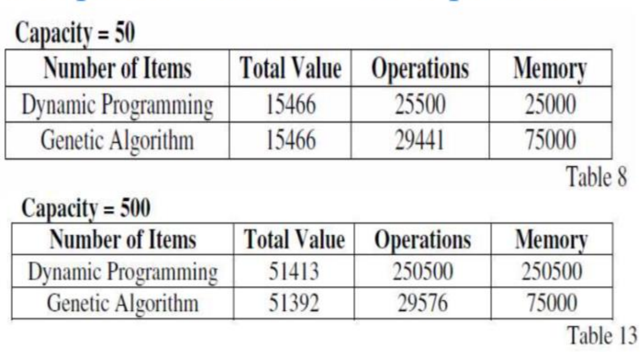
Knapsack Problem

Problem Description:

- You are going on a picnic.
- And have a number of items that you could take along.
- Each item has a weight and a benefit or value.
- You can take one of each item at most.
- There is a capacity limit on the weight you can carry.
- You should carry items with max. values.

Genetic algorithms (GAs) are stochastic search algorithms that mimic the biological process of evolution enabling thereby users to solve complex optimization problems. They operate based on a population of chromosomes, where a chromosome represents a candidate solution. Genetic operators allow the population to evolve over time and to converge to the optimal solution.
GAs have shown to be well suited for high-quality solutions to larger NP problems and currently they are the most efficient ways for finding an approximately optimal solution for optimization problems. They do not involve extensive search algorithms and do not try to find the best solution, but they simply generate a candidate for a solution, check in polynomial time whether it is a solution or not and how good a solution it is. GAs do not always give the optimal solution, but a solution that is close enough to the optimal one.

Knapsack Problem $\Rightarrow$ NP problem



Capacity = 50

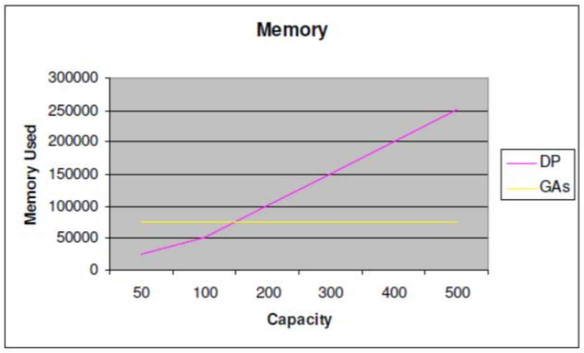
Number of Items	Total Value	Operations	Memory
Dynamic Programming	15466	25500	25000
Genetic Algorithm	15466	29441	75000

Table 8

Capacity = 500

Number of Items	Total Value	Operations	Memory
Dynamic Programming	51413	250500	250500
Genetic Algorithm	51392	29576	75000

Table 13



Dynamic Programming $\Rightarrow$ number of items and the capacity
Genetic Algorithm $\Rightarrow$ number of items and number of population

3. Implementation Details

Pseudo Code for a Basic Genetic Algorithm

The pseudo code for a basic genetic algorithm is as follows:

```
generation = 0;
population[generation] = initializePopulation(populationSize);
evaluatePopulation(population[generation]);
While isTerminationConditionMet() == false do
    parents = selectParents(population[generation]);
    population[generation+1] = crossover(parents);
    population[generation+1] = mutate(population[generation+1]);
    evaluatePopulation(population[generation]);
    generation++;
End loop;
```

A. KnapSackGA/src Classes for Knapsack implementation:

There are 5 java classes in knapsackga package.

Individual

In Individual class, we are taking an array of chromosomes as an parameter. In this class there are getters and setters of fitness and gene.

Population

In population class, there are getter and setter of fitness of the population. Also, we are comparing each individual in a population for fitness and returning the fittest one.

KnapSack

In knapsack class, we have defined five attributes i.e. chromosome length, weights, values, knapsack weight capacity, and optimum value.

KanpSackGA

In this class we have initialised the population and also added the termination condition which is the program will iterate till best solution is achieved or it will run to the number of generations defined.

GeneticAlgorithm

In Genetic Algorithm class, we are calculating fitness of an individual, mean fitness over a generation for a population. We are doing selection using tournament selection and doing crossover and mutation with elitism. Also, there are two selection methods mentioned in code i.e. Tournament selection and Roulette wheel selection. But we are using tournament selection in the code as a selection method.

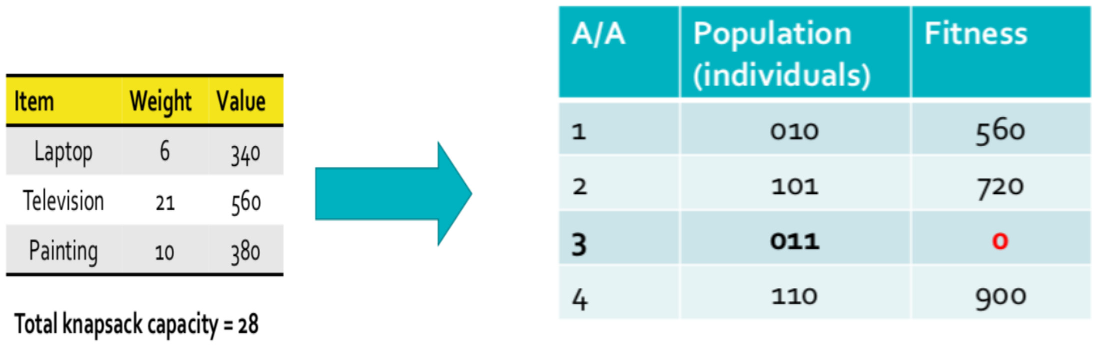
B. KnapSackGA/test Class:

KnapsackgaTest

In the test class we have tested 8 test cases of knapsack problem. All the testcases are successfully passed. Screenshot is also attached in section 6.

C. Fitness Evaluation:

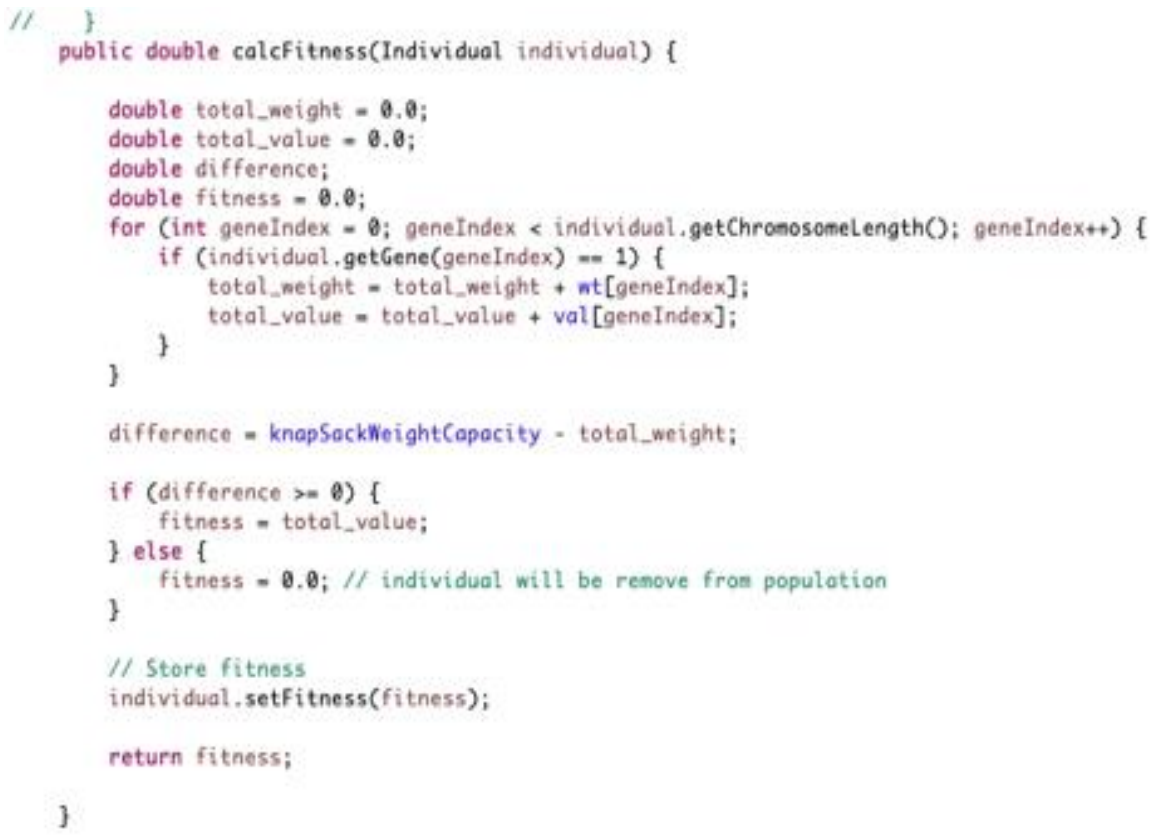
The fitness function sums the corresponding weights and values for each population member one by one. It then compares the population member's total weight to the knapsack capacity.
if(knapsack capacity >= population's member total weight)
fitness value = population's member total value
else
fitness value = 0



Item	Weight	Value
Laptop	6	340
Television	21	560
Painting	10	380

Total knapsack capacity = 28

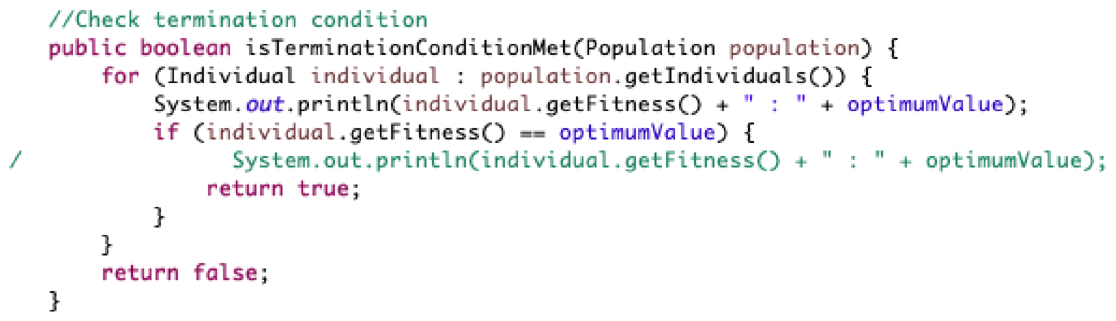
A/A	Population (individuals)	Fitness
1	010	560
2	101	720
3	**011**	**0**
4	110	900



```
//    }
    public double calcFitness(Individual individual) {

        double total_weight = 0.0;
        double total_value = 0.0;
        double difference;
        double fitness = 0.0;
        for (int geneIndex = 0; geneIndex < individual.getChromosomeLength(); geneIndex++) {
            if (individual.getGene(geneIndex) == 1) {
                total_weight = total_weight + wt[geneIndex];
                total_value = total_value + val[geneIndex];
            }
        }

        difference = knapSackWeightCapacity - total_weight;

        if (difference >= 0) {
            fitness = total_value;
        } else {
            fitness = 0.0; // individual will be remove from population
        }

        // Store fitness
        individual.setFitness(fitness);

        return fitness;

    }
```

D. Terminating Condition:

The population converges when either 90% of the chromosomes in the population have the same fitness value or the number of generations is greater than a fixed number.



```
   //Check termination condition
   public boolean isTerminationConditionMet(Population population) {
       for (Individual individual : population.getIndividuals()) {
           System.out.println(individual.getFitness() + " : " + optimumValue);
           if (individual.getFitness() == optimumValue) {
/                 System.out.println(individual.getFitness() + " : " + optimumValue);
               return true;
           }
       }
       return false;
   }
```

E. Parent Selection:

Fitness Proportionate Selection is one of the most popular ways of parent selection. In this every individual can become a parent with a probability which is proportional to its fitness. Therefore, fitter individuals have a higher chance of mating and propagating their features to the next generation.
Therefore, such a selection strategy applies a selection pressure to the more fit individuals in the population, evolving better individuals over time.
we tried two selection methods: roulette-wheel and group selection, combined with elitism, where two of the fittest chromosomes are copied without changes to the new population, so the best solutions found will not be lost.

Roulette Wheel Selection

Roulette wheel selection is widely used for maximization problems but it has two main drawbacks. First drawback is that it can handle only maximization problem so minimization problem must be converted into an equivalent maximization problem. If fitness values may be negative then transformation from minimization to maximization becomes more difficult. Second problem is that if a population contains a chromosome having exceptionally better fitness compared to the rest of chromosomes in the population then this chromosome occupies most of the roulette wheel area. Thus, almost all the spinning of the roulette wheel is likely to choose the same chromosome, this may result in the lost of genes diversity and population may converge to local optima. Second problem can be avoided by the scaling of fitness, where fitness of each chromosome is linearly mapped between a lower and an upper bound before marking the roulette wheel.

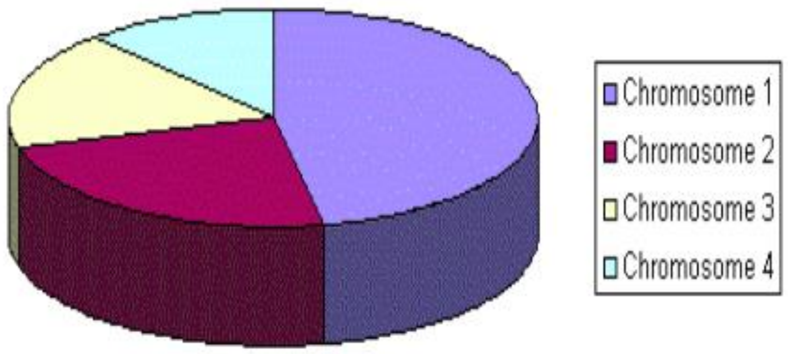
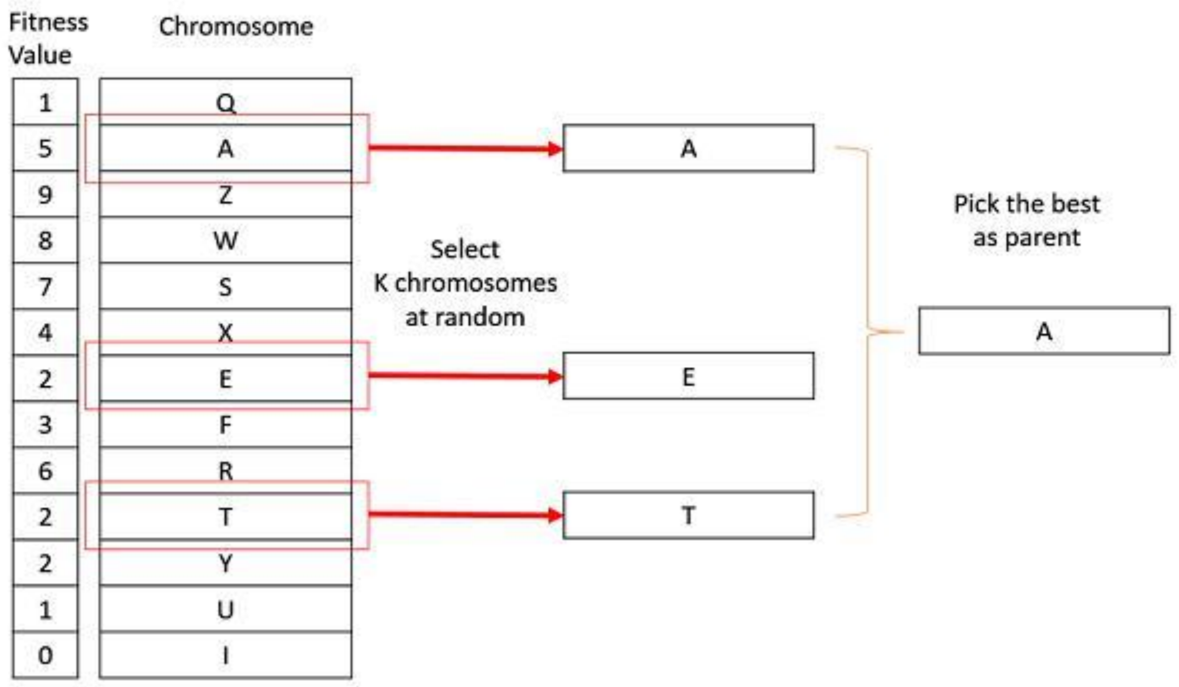


Figure 3.1 Roulette Wheel Selection Method

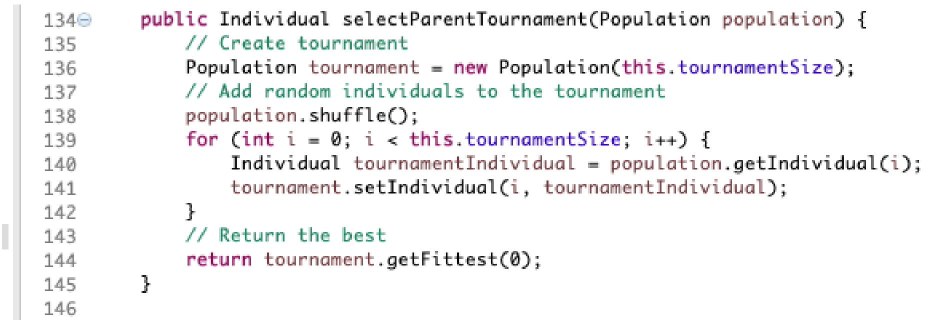
Tournament Selection

Tournament selection is one of the most common selection methods used in genetic algorithms. Its strengths being that it's a relatively simple algorithm to implement and allows for variable selection pressure by updating the tournament size.

In K-Way tournament selection, we select K individuals from the population at random and select the best out of these to become a parent. The same process is repeated for selecting the next parent. Tournament Selection is also extremely popular in literature as it can even work with negative fitness values.



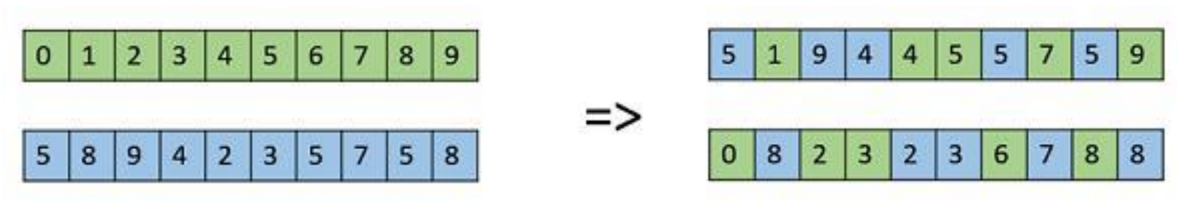
In the code, Tournament selection for parent selection is used.



```java
    public Individual selectParentTournament(Population population) {
        // Create tournament
        Population tournament = new Population(this.tournamentSize);
        // Add random individuals to the tournament
        population.shuffle();
        for (int i = 0; i < this.tournamentSize; i++) {
            Individual tournamentIndividual = population.getIndividual(i);
            tournament.setIndividual(i, tournamentIndividual);
        }
        // Return the best
        return tournament.getFittest(0);
    }
```

F. Crossover Function:

Crossover or recombination works as per the principle of sexual recombination. In biological systems, recombination is a complex process that occurs between male and female of same species. Two chromosomes are physically aligned, breakage occurs at one or more location on each chromosome and homologous chromosome fragments are exchanged before the breaks are repaired. Same concept is also applied in the genetic algorithm. Two parent chromosomes are selected randomly from the mating pool, few genes of the chromosomes are exchanged between these two parents and offspring are produced. In general, crossover operator recombines two chromosomes so it is also known as recombination. Crossover is intelligent search operator that exploits the information acquired by the parent chromosomes to generate new offspring. If both the parents have same genetic structure then offspring are just copies of the parent irrespective of cutting point but if parents have different genetic structure then offspring are different then parents. Thus, crossover is sampling process, which samples new points in the search space. Better the sampling rate, more chances of getting solution. In biological system, new child is produced by sexual crossing-over only. Thus, crossover should be performed every time, so generally probability of crossover is very high like 1.00, 0.95, 0.90 etc...
Crossover can also be done by various methods but the method that I have used is Uniform Crossover. In uniform crossover each gene has 50% chance of either coming from 1st parent or second parent.



Crossover Pseudo Code

Now that we have a selection and a crossover method, let’s look at some pseudo code which outlines the crossover process to be implemented.

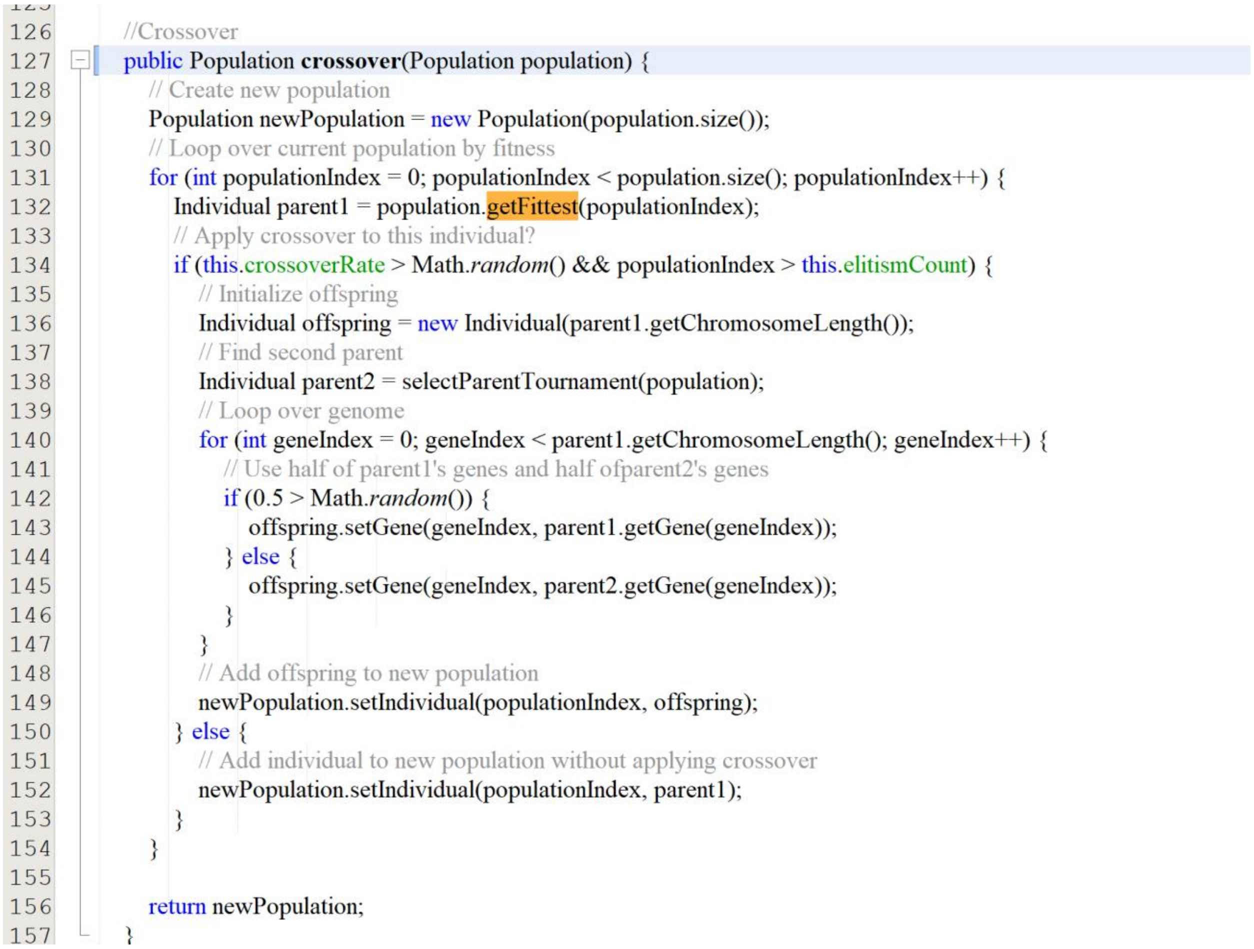
```
1: For each individual in population:
2:     newPopulation = new array;
2:     If crossoverRate > random():
3:         secondParent = selectParent();
4:         offspring = crossover(individual, secondParent);
5:         newPopulation.push(offspring);
6:     Else:
7:         newPopulation.push(individual);
8:     End if
9: End loop;
```

G. Elitism:

A basic genetic algorithm will often lose the best individuals in a population between generations because of the crossover and mutation operators. However, we need these operators to find better solutions.
One simple optimization technique used to tackle this problem is to always allow the fittest individual, or individuals, to be added unaltered to the next generation's population. This way the best individuals are no longer lost from generation to generation.
This process of retaining the best for the next generation is called elitism.
Typically, the optimal number of 'elite' individuals in a population will be a very small proportion of the total population size. This is because if the value is too high, it will slow down the genetic algorithm's search process due to a lack of genetic diversity caused by preserving too many individuals.

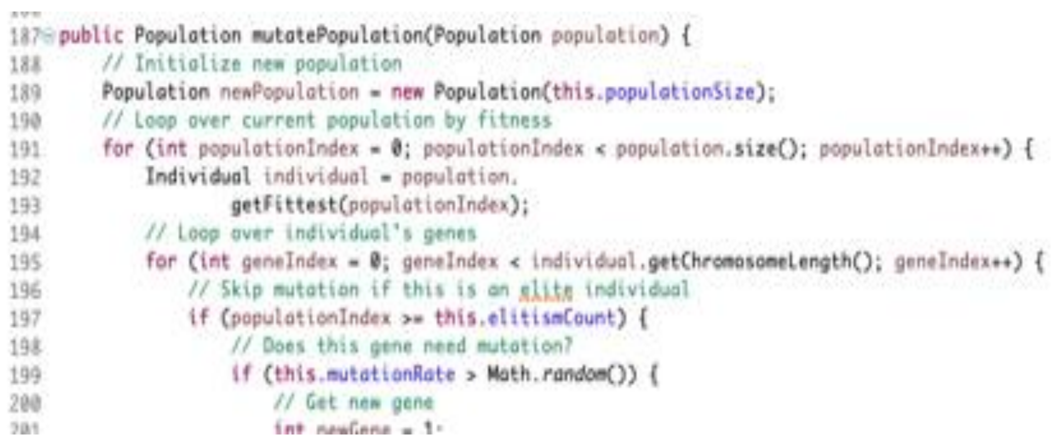
Implementing elitism is simple in both crossover and mutation contexts.

For Crossover:



```
    //Crossover
    public Population crossover(Population population) {
        // Create new population
        Population newPopulation = new Population(population.size());
        // Loop over current population by fitness
        for (int populationIndex = 0; populationIndex < population.size(); populationIndex++) {
            Individual parent1 = population.getFittest(populationIndex);
            // Apply crossover to this individual?
            if (this.crossoverRate > Math.random() && populationIndex > this.elitismCount) {
                // Initialize offspring
                Individual offspring = new Individual(parent1.getChromosomeLength());
                // Find second parent
                Individual parent2 = selectParentTournament(population);
                // Loop over genome
                for (int geneIndex = 0; geneIndex < parent1.getChromosomeLength(); geneIndex++) {
                    // Use half of parent1's genes and half ofparent2's genes
                    if (0.5 > Math.random()) {
                        offspring.setGene(geneIndex, parent1.getGene(geneIndex));
                    } else {
                        offspring.setGene(geneIndex, parent2.getGene(geneIndex));
                    }
                }
                // Add offspring to new population
                newPopulation.setIndividual(populationIndex, offspring);
            } else {
                // Add individual to new population without applying crossover
                newPopulation.setIndividual(populationIndex, parent1);
            }
        }

        return newPopulation;
    }
```

For Mutation:



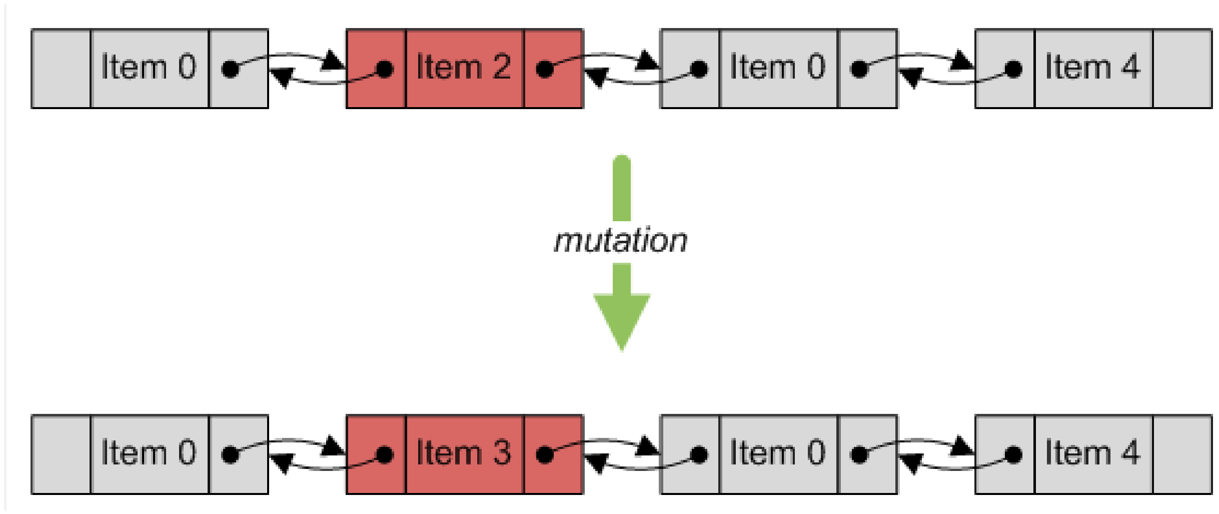
```
public Population mutatePopulation(Population population) {
    // Initialize new population
    Population newPopulation = new Population(this.populationSize);
    // Loop over current population by fitness
    for (int populationIndex = 0; populationIndex < population.size(); populationIndex++) {
        Individual individual = population.
                getFittest(populationIndex);
        // Loop over individual's genes
        for (int geneIndex = 0; geneIndex < individual.getChromosomeLength(); geneIndex++) {
            // Skip mutation if this is an elite individual
            if (populationIndex >= this.elitismCount) {
                // Does this gene need mutation?
                if (this.mutationRate > Math.random()) {
                    // Get new gene
                    int newGene = 1;
```

Crossover is only applied if *both* the crossover conditional is met *and* the individual is not considered elite.
At this point, the individuals in the population have already been sorted by their fitness, so the strongest individuals have the lowest indices. Therefore, if we want three elite individuals, we should skip indices 0-2 from consideration.

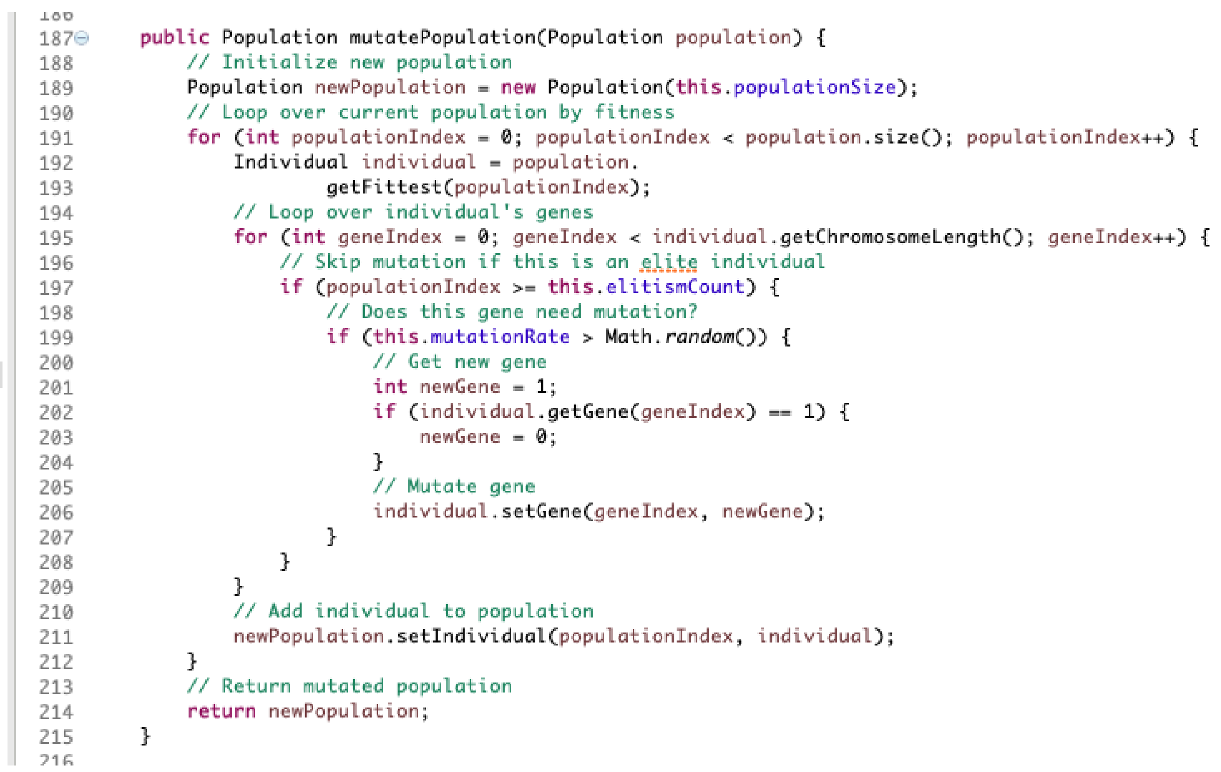
H. Mutation Function:

Mutation is secondary operator used in genetic algorithm to explore new points in the search space. In the latter stages of a run, the population may converge in wrong direction and stuck to the local optima. The effect of mutation is to reintroduce divergence into a converging population. Mutation operator selects one chromosome randomly from the population, then selects some genes using mutation probability and flips that bit. So mutation is a random operator that randomly alters some value. Mutation either explores some new points in the search space and leads population to the global optima direction or alters value of the best chromosome and losses knowledge acquired, till now. Generally per gene probability of mutation is 0.001, 0.01, 0.02 etc.

Mutation operation is responsible for retaining genetic diversity of the chromosomes in the population. It is done by randomly changing genes in newly produced chromosomes:



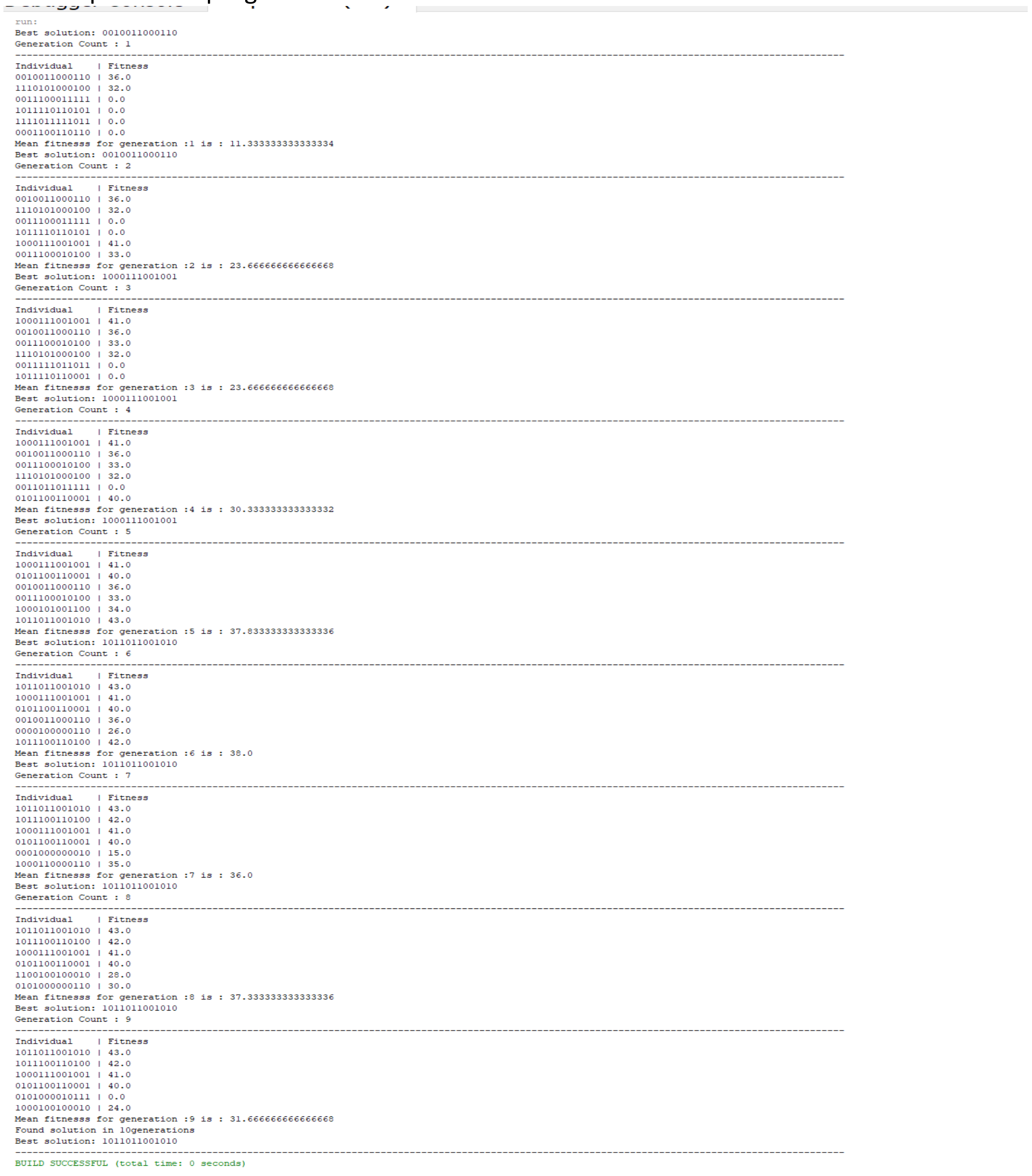
Implementation of mutation in code:



```java
    public Population mutatePopulation(Population population) {
        // Initialize new population
        Population newPopulation = new Population(this.populationSize);
        // Loop over current population by fitness
        for (int populationIndex = 0; populationIndex < population.size(); populationIndex++) {
            Individual individual = population.
                    getFittest(populationIndex);
            // Loop over individual's genes
            for (int geneIndex = 0; geneIndex < individual.getChromosomeLength(); geneIndex++) {
                // Skip mutation if this is an elite individual
                if (populationIndex >= this.elitismCount) {
                    // Does this gene need mutation?
                    if (this.mutationRate > Math.random()) {
                        // Get new gene
                        int newGene = 1;
                        if (individual.getGene(geneIndex) == 1) {
                            newGene = 0;
                        }
                        // Mutate gene
                        individual.setGene(geneIndex, newGene);
                    }
                }
            }
            // Add individual to population
            newPopulation.setIndividual(populationIndex, individual);
        }
        // Return mutated population
        return newPopulation;
    }
```

4. Program Output:

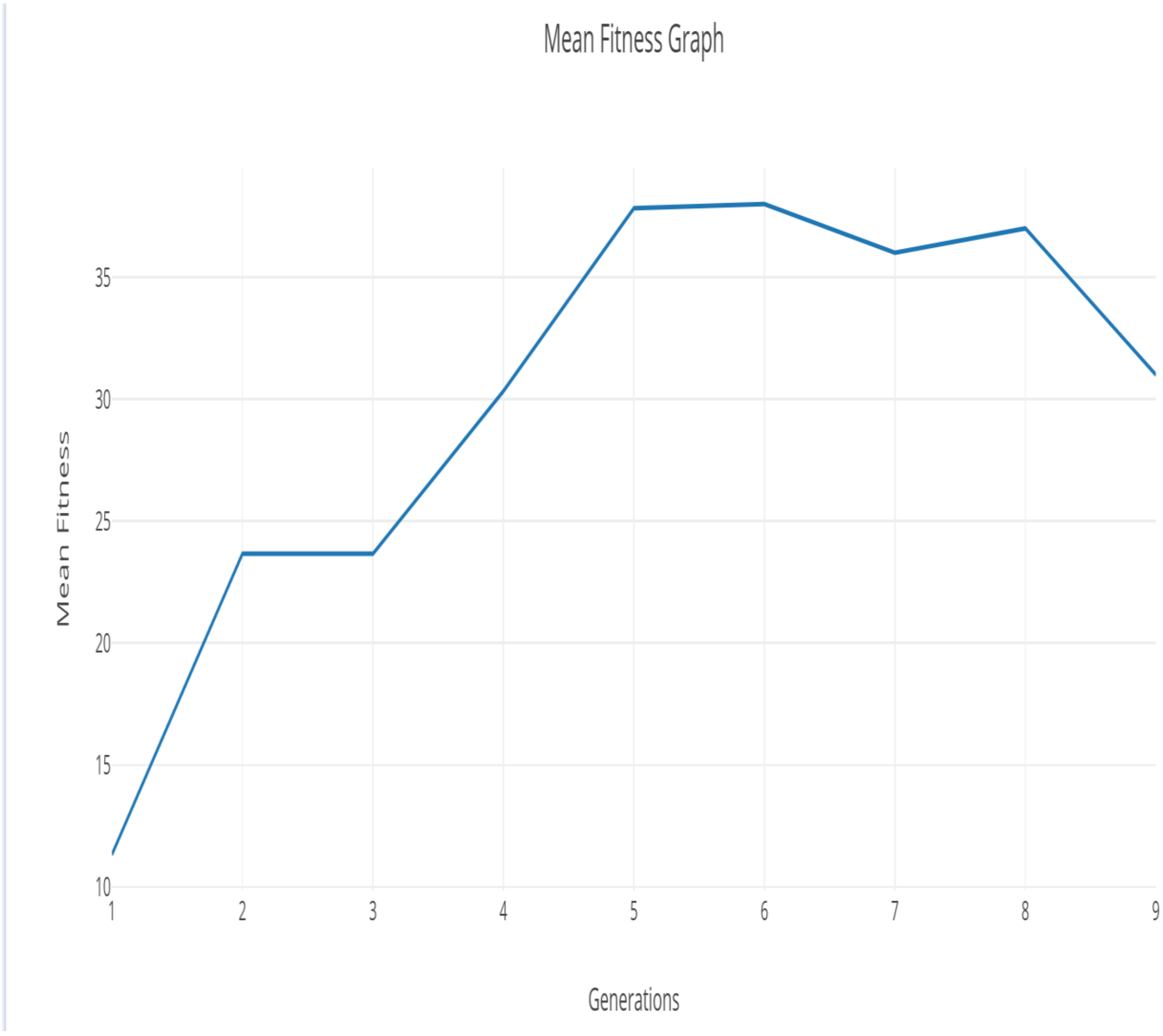
The output of the program is shown below:



```
run:
Best solution: 0010011000110
Generation Count : 1
-----------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
0010011000110 | 36.0
1110101000100 | 32.0
0011100011111 | 0.0
1011110110101 | 0.0
1111011111011 | 0.0
0001100110110 | 0.0
Mean fitnesss for generation :1 is : 11.333333333333334
Best solution: 0010011000110
Generation Count : 2
-----------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
0010011000110 | 36.0
1110101000100 | 32.0
0011100011111 | 0.0
1011110110101 | 0.0
1000111001001 | 41.0
0011100010100 | 33.0
Mean fitnesss for generation :2 is : 23.666666666666668
Best solution: 1000111001001
Generation Count : 3
-----------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
1000111001001 | 41.0
0010011000110 | 36.0
0011100010100 | 33.0
1110101000100 | 32.0
0011111011011 | 0.0
1011110110001 | 0.0
Mean fitnesss for generation :3 is : 23.666666666666668
Best solution: 1000111001001
Generation Count : 4
-----------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
1000111001001 | 41.0
0010011000110 | 36.0
0011100010100 | 33.0
1110101000100 | 32.0
0011011011111 | 0.0
0101100110001 | 40.0
Mean fitnesss for generation :4 is : 30.333333333333332
Best solution: 1000111001001
Generation Count : 5
-----------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
1000111001001 | 41.0
0101100110001 | 40.0
0010011000110 | 36.0
0011100010100 | 33.0
1000101001100 | 34.0
1011011001010 | 43.0
Mean fitnesss for generation :5 is : 37.833333333333336
Best solution: 1011011001010
Generation Count : 6
-----------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
1011011001010 | 43.0
1000111001001 | 41.0
0101100110001 | 40.0
0010011000110 | 36.0
0000100000110 | 26.0
1011100110100 | 42.0
Mean fitnesss for generation :6 is : 38.0
Best solution: 1011011001010
Generation Count : 7
-----------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
1011011001010 | 43.0
1011100110100 | 42.0
1000111001001 | 41.0
0101100110001 | 40.0
0001000000010 | 15.0
1000110000110 | 35.0
Mean fitnesss for generation :7 is : 36.0
Best solution: 1011011001010
Generation Count : 8
-----------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
1011011001010 | 43.0
1011100110100 | 42.0
1000111001001 | 41.0
0101100110001 | 40.0
1100100100010 | 28.0
0101000000110 | 30.0
Mean fitnesss for generation :8 is : 37.333333333333336
Best solution: 1011011001010
Generation Count : 9
-----------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
1011011001010 | 43.0
1011100110100 | 42.0
1000111001001 | 41.0
0101100110001 | 40.0
0101000010111 | 0.0
1000100100010 | 24.0
Mean fitnesss for generation :9 is : 31.666666666666668
Found solution in 10generations
Best solution: 1011011001010
-----------------------------------------------------------------------------------------------------------------------------------------------
BUILD SUCCESSFUL (total time: 0 seconds)
```

In the above output, desired output or we can say the best solution is 24.00 which is achieved after 9 Generations. Although the number of generations is 1000. Also, in output we have calculated the mean fitness of all the individuals in the population over a generation.

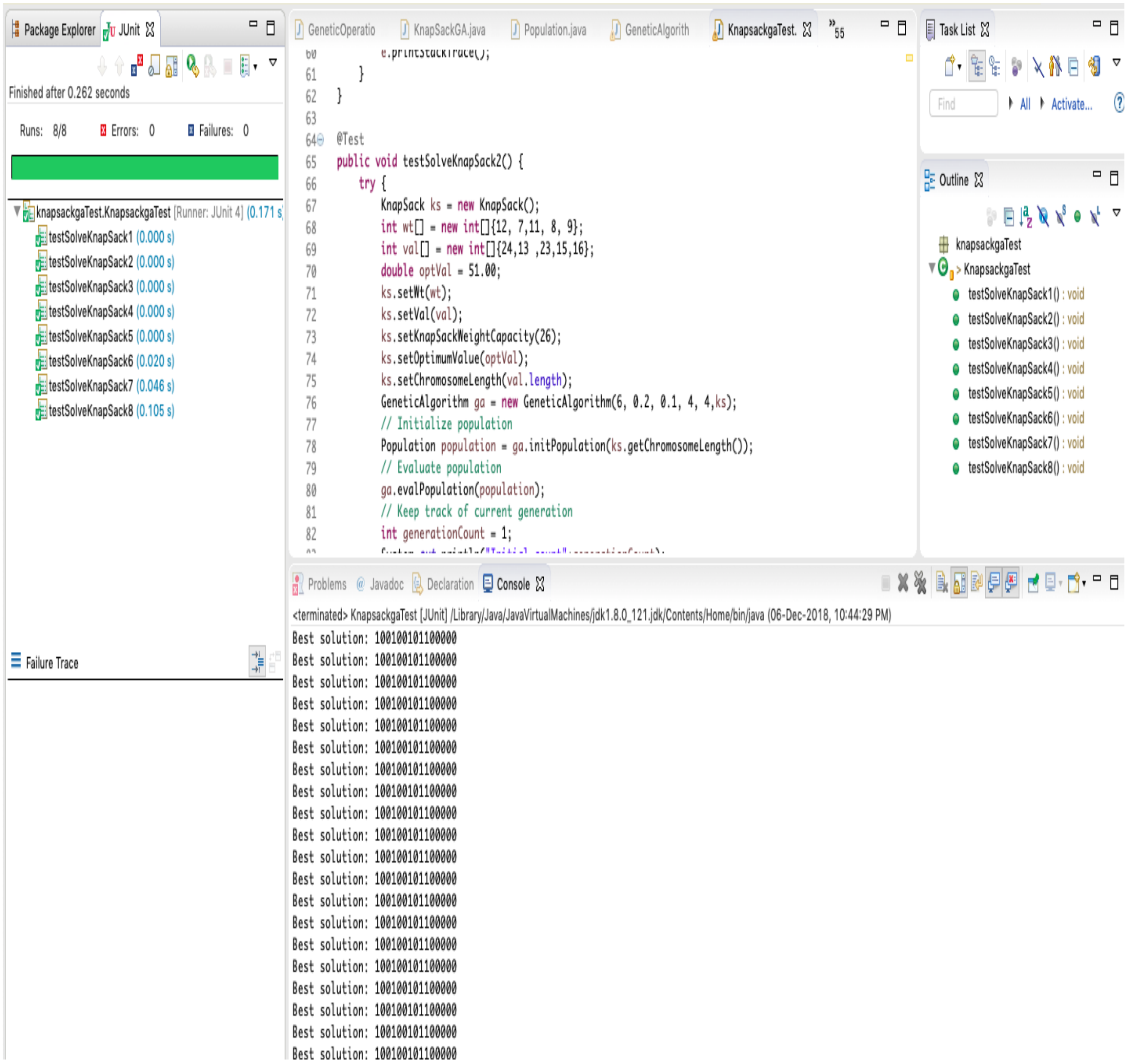
Below is the mean fitness graph of the output generated after 9 generations.



5. Test Case Passed:

There are 8 Test cases used in my project.

KnapsackgaTest class is used to initialize weight, value, optimum value, population size, tournament count, elitism count, Crossover rate, Mutation rate and weight capacity. Also, termination condition is mentioned to run the program till the best solution is achieved or to the number of generations mentioned.



Finished after 0.262 seconds

Runs: 8/8 Errors: 0 Failures: 0

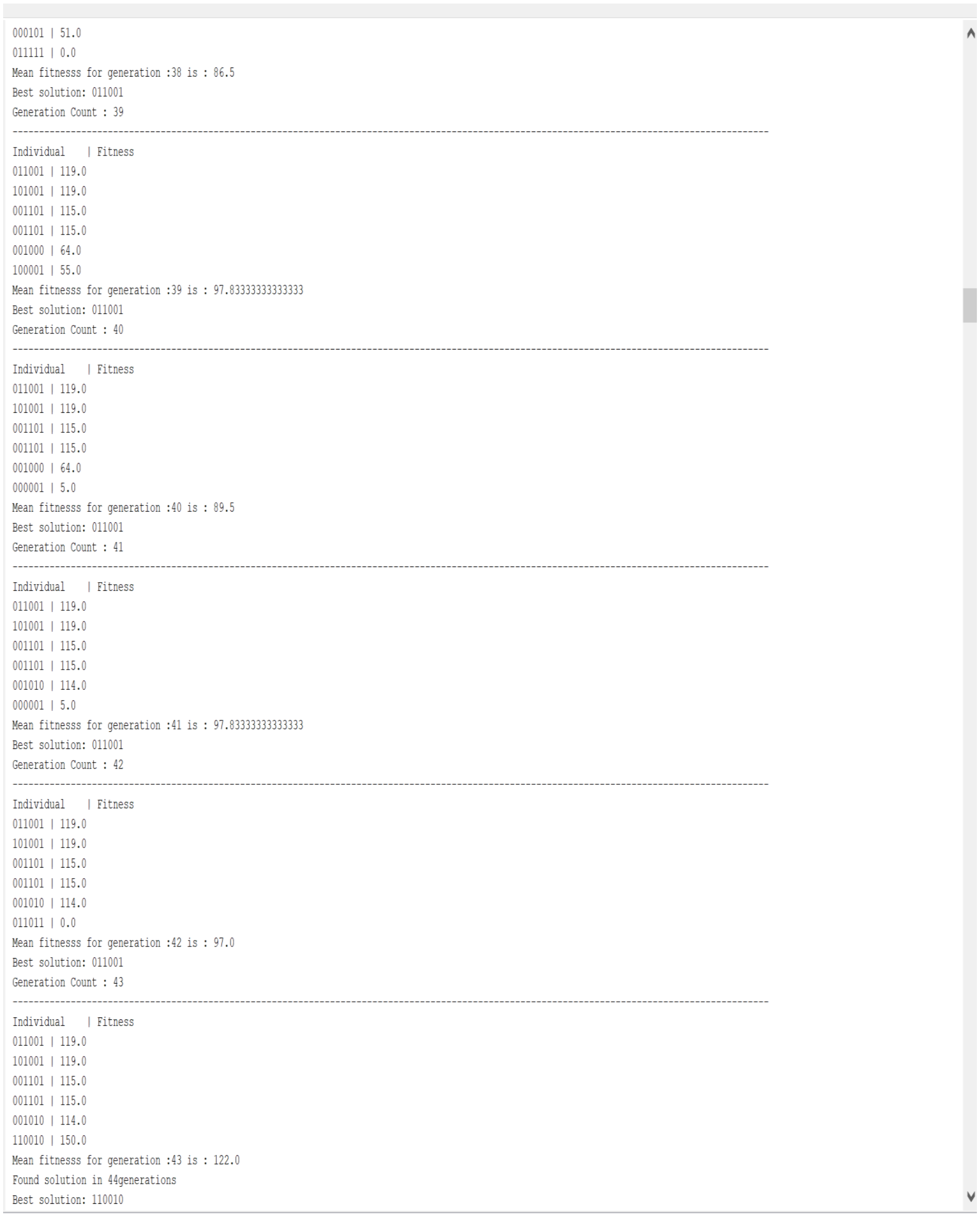
```
knapsackgaTest.KnapsackgaTest [Runner: JUnit 4] (0.171 s)
    testSolveKnapSack1 (0.000 s)
    testSolveKnapSack2 (0.000 s)
    testSolveKnapSack3 (0.000 s)
    testSolveKnapSack4 (0.000 s)
    testSolveKnapSack5 (0.000 s)
    testSolveKnapSack6 (0.020 s)
    testSolveKnapSack7 (0.046 s)
    testSolveKnapSack8 (0.105 s)
```

```java
            e.printStackTrace();
        }
    }

    @Test
    public void testSolveKnapSack2() {
        try {
            KnapSack ks = new KnapSack();
            int wt[] = new int[]{12, 7,11, 8, 9};
            int val[] = new int[]{24,13 ,23,15,16};
            double optVal = 51.00;
            ks.setWt(wt);
            ks.setVal(val);
            ks.setKnapSackWeightCapacity(26);
            ks.setOptimumValue(optVal);
            ks.setChromosomeLength(val.length);
            GeneticAlgorithm ga = new GeneticAlgorithm(6, 0.2, 0.1, 4, 4,ks);
            // Initialize population
            Population population = ga.initPopulation(ks.getChromosomeLength());
            // Evaluate population
            ga.evalPopulation(population);
            // Keep track of current generation
            int generationCount = 1;
```

```
knapsackgaTest
KnapsackgaTest
    testSolveKnapSack1() : void
    testSolveKnapSack2() : void
    testSolveKnapSack3() : void
    testSolveKnapSack4() : void
    testSolveKnapSack5() : void
    testSolveKnapSack6() : void
    testSolveKnapSack7() : void
    testSolveKnapSack8() : void
```

```
<terminated> KnapsackgaTest [JUnit] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java (06-Dec-2018, 10:44:29 PM)
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
Best solution: 100100101100000
```

Output after running all the test cases:



```
000101 | 51.0
011111 | 0.0
Mean fitnesss for generation :38 is : 86.5
Best solution: 011001
Generation Count : 39
------------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
011001 | 119.0
101001 | 119.0
001101 | 115.0
001101 | 115.0
001000 | 64.0
100001 | 55.0
Mean fitnesss for generation :39 is : 97.83333333333333
Best solution: 011001
Generation Count : 40
------------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
011001 | 119.0
101001 | 119.0
001101 | 115.0
001101 | 115.0
001000 | 64.0
000001 | 5.0
Mean fitnesss for generation :40 is : 89.5
Best solution: 011001
Generation Count : 41
------------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
011001 | 119.0
101001 | 119.0
001101 | 115.0
001101 | 115.0
001010 | 114.0
000001 | 5.0
Mean fitnesss for generation :41 is : 97.83333333333333
Best solution: 011001
Generation Count : 42
------------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
011001 | 119.0
101001 | 119.0
001101 | 115.0
001101 | 115.0
001010 | 114.0
011011 | 0.0
Mean fitnesss for generation :42 is : 97.0
Best solution: 011001
Generation Count : 43
------------------------------------------------------------------------------------------------------------------------------------------------
Individual    | Fitness
011001 | 119.0
101001 | 119.0
001101 | 115.0
001101 | 115.0
001010 | 114.0
110010 | 150.0
Mean fitnesss for generation :43 is : 122.0
Found solution in 44generations
Best solution: 110010
```

6. Complexity of the program

Since the number of chromosomes in each generation (Size) and the number of generations are fixed, the complexity of the program depends only on the number of items that may potentially be placed in the knapsack. Let us suppose the following abbreviations, N for the number of items, S for the size of the population, and G for the number of possible generations. The function that initializes the array chromosomes has a complexity of O(N). The fitness, crossover function, and mutation functions have also complexities of O(N). The complexities of the two selection functions and the function that checks for the terminating condition do not depend on N (but on the size of the population) and they have constant times of running O(1). The selection, crossover, and mutation operations are performed in a for loop, which runs S times. Since, S is a constant, the complexity of the whole loop is O(N). Finally, all these genetic operations are performed in a do while loop, which runs at most G times. Since G is a constant, it will not affect the overall asymptotic complexity of the program. Thus, the total complexity of the program is O(N).

<u>7. Conclusion:</u>

We have shown how Genetic Algorithms can be used to find good solutions for the 0-1 Knapsack Problem. GAs reduce the complexity of the KP from exponential to linear, which makes it possible to find approximately optimal solutions for an NP problem. The results from the program show that the implementation of a good selection method and elitism are very important for the good performance of a genetic algorithm.

8. References

1. http://math.stmarys-ca.edu/wp-content/uploads/2017/07/Christopher-Queen.pdf
2. https://www.codeproject.com/Articles/607791/ga-knapsack
3. Genetic Algorithms in Java Basics by Lee Jacobson and Burak Kanber
4. https://www.dataminingapps.com/2017/03/solving-the-knapsack-problem-with-a-simple-genetic-algorithm/
5. http://www.sc.ehu.es/ccwbayes/docencia/kzmm/files/AG-knapsack.pdf
6. http://shodhganga.inflibnet.ac.in/bitstream/10603/78747/10/10_chapter%203.pdf