**SOFT COMPUTNG CONCEPTS**

**(ITPC-322)**



**Dr. B R Ambedkar National Institute of Technology, Jalandhar**

**Punjab- 144011**

**SUBMITTED TO : SUBMITTED BY :**

Dr. Mohit Kumar Arvinder Sngh

Assistant Professor 18124004

Dept. Of Information Technology IT/6th Semester

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| Lab No. | Practical Performed | Page No. | Remarks |
|  |  |  |  |
| 1 | To Implement Travelling Salesman problem | 3 – 9 |  |
| 2 | To Implement   * 0/1 Knapsack * partitioning into two subsets so that subset sum difference is minimum | 10 - 19 |  |
| 3 | WAP to implement the genetic algorithm | 20 - 28 |  |

**LAB : 1**

**OBJECTIVE :**

To Implement Travelling Salesman problem

**Problem Statement :**

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city.

**Implementation :**

1. To print shortest possible route : We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are **(*n* - 1)!** number of possibilities. Complexity is O((n-1)!)

**Code :**

#include<bits/stdc++.h>

using namespace std;

void INPUT()

{

#ifndef ONLINE\_JUDGE

freopen("C:/Users/arvin/Desktop/Current/input.txt", "r", stdin);

freopen("C:/Users/arvin/Desktop/Current/output.txt", "w", stdout);

#endif

}

int n ;

// Adj Matrix which defines the graphvector<vector<int>> dist;

list<int> sequence;

//If all cities have been visited

int VISITED\_ALL;

int ans\_overall = INT\_MAX;

int tsp(int mask, int pos, list<int> &temp, int cur\_cost) {

if (mask == VISITED\_ALL) {

if (ans\_overall > cur\_cost + dist[pos][0]) {

sequence = temp;

sequence.push\_back(0);

}

//all\_path[cur\_cost + dist[pos][0]] = temp;

return dist[pos][0];

}

int ans = INT\_MAX;

//visiting an unvisited city

for (int city = 0; city < n; city++) {

if ((mask & (1 << city)) == 0) {

temp.push\_back(city);

int newAns = dist[pos][city] + tsp( mask | (1 << city), city, temp , cur\_cost + dist[pos][city]);

ans = min(ans, newAns);

temp.pop\_back();

}

}

return ans;

}

int main() {

INPUT();

cin >> n;

VISITED\_ALL = (1 << n) - 1;

dist = vector< vector<int> > (n + 1, vector<int> (n + 1, INT\_MAX));

for (int i = 0; i < n; ++i)

{

for (int j = 0; j < n; ++j)

{

cin >> dist[i][j];

}

}

list<int> temp;

temp.push\_back(0);

int tsp\_cost = tsp(1, 0, temp, 0);

cout << "Min weight hamiltonian path costs : " << tsp\_cost << endl;

cout << "\n\nPath: ";

for (auto i : sequence) {

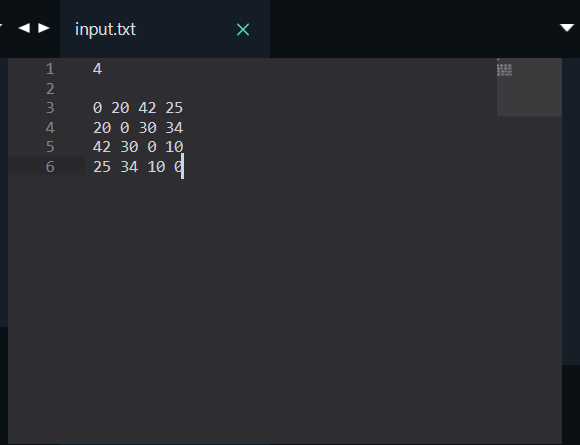
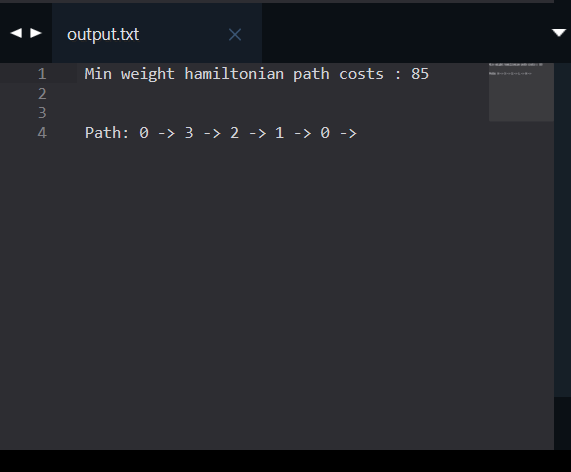
cout << i << " -> ";

}

return 0;

}

**Output :**

1. Optimised method to find minimum travelling cost using Dynamic Programming : Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

**Algorithm: Traveling-Salesman-Problem**

C ({1}, 1) = 0

for s = 2 to n do

for all subsets S Є {1, 2, 3, … , n} of size s and containing 1

C (S, 1) = ∞

for all j Є S and j ≠ 1

C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}

Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

**Analysis :**

There are at the most 2*n*.*n* sub-problems and each one takes linear time to solve. Therefore, the total running time is *O*(2*n*.*n*2).

**Code :**

#include <iostream>

#include<vector>

#include<climits>

#include<cstring>

using namespace std;

void INPUT()

{

#ifndef ONLINE\_JUDGE

freopen("C:/Users/arvin/Desktop/Current/input.txt", "r", stdin);

freopen("C:/Users/arvin/Desktop/Current/output.txt", "w", stdout);

#endif

}

int n = 4;

vector<vector<int>> dp;

// Adj Matrix which defines the graph

vector<vector<int>> dist;

//If all cities have been visited

int VISITED\_ALL = (1 << n) - 1;

int tsp(int mask, int pos) {

if (mask == VISITED\_ALL) {

return dist[pos][0];

}

//Lookup

if (dp[mask][pos] != -1) {

return dp[mask][pos];

}

int ans = INT\_MAX;

//visiting an unvisited city

for (int city = 0; city < n; city++) {

if ((mask & (1 << city)) == 0) {

int newAns = dist[pos][city] + tsp( mask | (1 << city), city);

ans = min(ans, newAns);

}

}

return dp[mask][pos] = ans;

}

int main() {

INPUT();

cin >> n;

VISITED\_ALL = (1 << n) - 1;

dist = vector< vector<int> > (n + 1, vector<int> (n + 1, INT\_MAX));

dp = vector< vector<int> > (1 << n, vector<int> (n, -1));

for (int i = 0; i < n; ++i)

{

for (int j = 0; j < n; ++j)

{

cin >> dist[i][j];

}

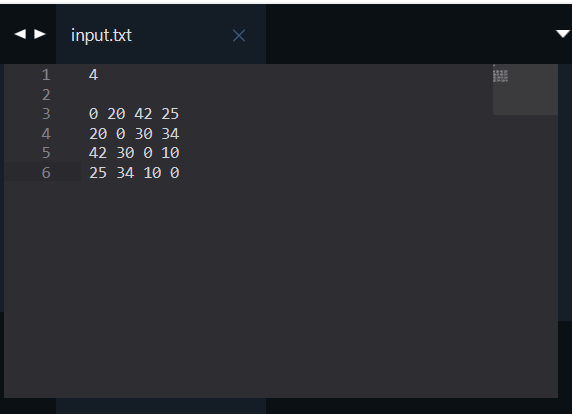
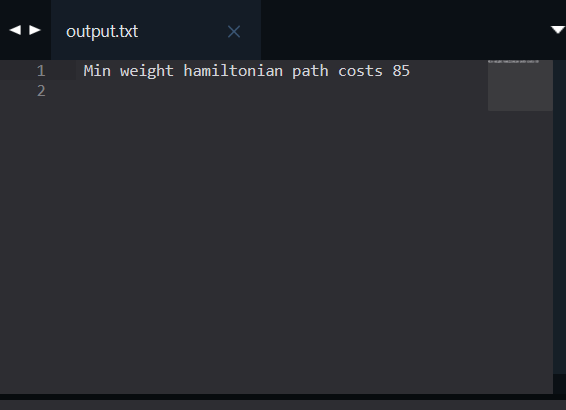
}

cout << "Min weight hamiltonian path costs " << tsp(1, 0) << endl;

return 0;

}

**Output :**

**LAB : 2**

**OBJECTIVE :**

1. To Implement 0/1 Knapsack and analyse complexity of algorithm.
2. To Implement solution of partitioning into two subsets so that subset sum difference is minimum (such that each element of given set is included in one of two subsets).
3. **To Implement 0/1 Knapsack and analyse complexity of algorithm.**

**Problem Statement :**

Given the weights and profits of ‘N’ items, we are asked to put these items in a knapsack that has a capacity ‘C’. The goal is to get the maximum profit from the items in the knapsack. Each item can only be selected once, as we don’t have multiple quantities of any item.

**Implementation : Bottom-Up DP**

dp[i][c] will represent the maximum knapsack profit for capacity ‘c’ calculated from the first ‘i’ items.

So, for each item at index ‘i’ (0 <= i < items.length) and capacity ‘c’ (0 <= c <= capacity), we have two options:

1. Exclude the item at index ‘i’. In this case, we will take whatever profit we get from the sub-array excluding this item => dp[i-1][c]
2. Include the item at index ‘i’ if its weight is not more than the capacity. In this case, we include its profit plus whatever profit we get from the remaining capacity and from remaining items => profits[i] + dp[i-1][c-weights[i]]

Finally, our optimal solution will be maximum of the above two values:

    dp[i][c] = max (dp[i-1][c], profits[i] + dp[i-1][c-weights[i]])

**Space and Time Complexity of solutuon is O( capacity \* Number of items)**

**Code :**

#include <iostream>

#include <vector>

using namespace std;

void INPUT()

{

#ifndef ONLINE\_JUDGE

freopen("C:/Users/arvind/Desktop/Current/input.txt", "r", stdin);

freopen("C:/Users/arvind/Desktop/Current/output.txt", "w", stdout);

#endif

}

void printSelectedElements(vector<vector<int>> &dp, const vector<int> &weights,

const vector<int> &profits, int capacity) {

//dp index 1,2 .... mapped to wt index 0,1,2

cout << "Selected weights:";

int totalProfit = dp[weights.size()][capacity];

for (int i = weights.size(); i > 0; i--) {

if (totalProfit != dp[i - 1][capacity]) {

cout << " " << weights[i - 1];

capacity -= weights[i - 1];

totalProfit -= profits[i - 1];

}

}

cout << endl;

}

int solveKnapsack(const vector<int> &profits, const vector<int> &weights, int capacity) {

// basic checks

if (capacity <= 0 || profits.empty() || weights.size() != profits.size()) {

return 0;

}

int n = profits.size();

vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

//first column and row = 0 , (capacity =0 and no items resp.)

//dp index 1,2 .... mapped to wt index 0,1,2

for (int i = 1; i <= n; i++) {

for (int c = 1; c <= capacity; c++) {

int profit1 = 0, profit2 = 0;

// include the item, if it is not more than the capacity

if (weights[i - 1] <= c) {

profit1 = profits[i - 1] + dp[i - 1][c - weights[i - 1]];

}

// exclude the item

profit2 = dp[i - 1][c];

// take maximum

dp[i][c] = max(profit1, profit2);

}

}

printSelectedElements(dp, weights, profits, capacity);

// maximum profit will be at the bottom-right corner.

return dp[n][capacity];

}

int main(int argc, char \*argv[]) {

INPUT();

int n, maxCapacity;

cin >> n;

vector<int> profits(n);

vector<int> weights(n);

for (int i = 0; i < n; ++i)

cin >> profits[i];

for (int i = 0; i < n; ++i)

cin >> weights[i];

cin >> maxCapacity;

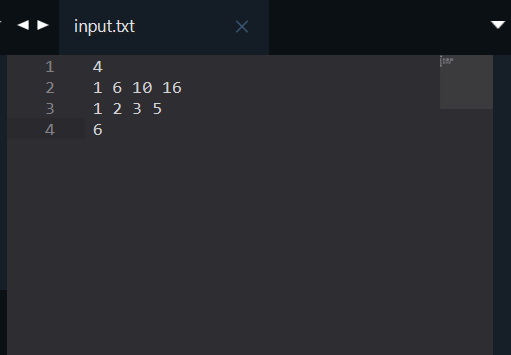
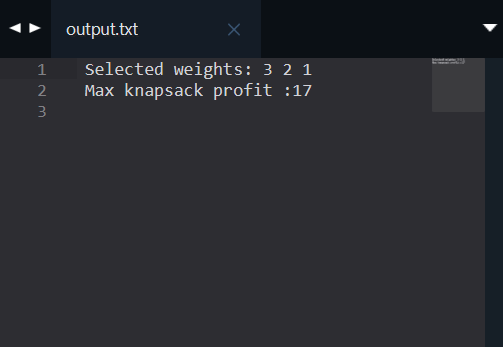
int maxProfit = solveKnapsack(profits, weights, maxCapacity);

cout << "Max knapsack profit :" << maxProfit << endl;

return 0;

}

**Output**

1. **To Implement solution of partitioning into two subsets so that subset sum difference is minimum**

**Problem Statement :**

Given a set of integers, the task is to divide it into two sets S1 and S2 such that the absolute difference between their sums is minimum.   
If there is a set S with n elements, then if we assume Subset1 has m elements, Subset2 must have n-m elements and the value of absolute value of (sum(Subset1) – sum(Subset2)) should be minimum.

**Implementation : Bottom-Up DP**

This problem is variation of 0/1 Knapsach problem.

We are needed to create two Subsets S1 and S2 such that their absolute sum difference is minimum. Suppose S1 with sum either less than or equal to subset sum S2 .

We can reframe the statement to minimize absolute sum difference of subset S1  and (TotalSum – sum of subset S1 ). We can observe that problem is similar to equal subset sum partition (where difference will be 0). To met the reqiurement of minimum sum difference, we try to get the sum of subset S1 as close as possible to TotalSum/2 (the sum of subset S1 lies in range 0 and TotalSum/2 both inclusive, as S1 is smaller sum subset). Here TotalSum is sum of all the given elements.

We can check for all the sums in range [0, TotalSum/2], that can we form a subset (subset S1 ) from given original numbers. Then we can choose subset S1 with sum as close as posiible to TotalSum/2 and all elements not in S1 wll form subset S2 .

We can apply bottom-up dynamic programming to solve problem. Here, dp[i][j] will hold a boolean value, where true represent whether we can form a subset sum ‘j’ with first ‘i’ elements. The elements of dp matrix is initialized to false and first column elements to true.

So, for each item at index ‘i’ (0 <= i <= number of elements) and capacity ‘j’ (0 <= j <= Totalum/2).

**Space and Time Complexity of solutuon is O( Number of elements\* TotalSum/2)**

**Code :**

#include <iostream>

#include <vector>

#include<climits>

using namespace std;

void INPUT()

{

#ifndef ONLINE\_JUDGE

freopen("C:/Users/arvind/Desktop/Current/input.txt", "r", stdin);

freopen("C:/Users/arvind/Desktop/Current/output.txt", "w", stdout);

#endif

}

void printSubsets(const vector<vector<bool>> &dp, const vector<int> &numbers, int firstSum) {

int n = numbers.size();

vector<bool> SubsetIndex(n, false);

//true in first subset else n second (probale greater sum subset)

int numberIndex = n;

while (firstSum and numberIndex > 0) {

if (dp[numberIndex][firstSum]) {

numberIndex--;

if (dp[numberIndex ][firstSum])

continue;

SubsetIndex[numberIndex] = true;

firstSum = firstSum - numbers[numberIndex] + 1;

}

firstSum--;

}

cout << "\nFirst Subset :";

for (int i = 0; i < n; ++i)

{

if (SubsetIndex[i])

cout << " " << numbers[i];

}

cout << "\nSecond Subset :";

for (int i = 0; i < n; ++i)

{

if (!SubsetIndex[i])

cout << " " << numbers[i];

}

cout << endl;

}

int findMin(vector<int> &numbers, int n) {

int sum = 0, total = 0;

for (int i : numbers)

total += i;

sum = total / 2;

//Subset Sum Problem

vector<vector<bool>> dp(n + 1, vector<bool>(sum + 1, false));

//rows - num

//col = sum [0-tatal/2]

for (int i = 0; i <= n; ++i)

dp[i][0] = true;

for (int i = 1; i <= n; ++i) {

for (int j = 1; j <= sum; ++j)

{

dp[i][j] = dp[i - 1][j];

if (j - numbers[i - 1] >= 0)

dp[i][j] = dp[i][j] || dp[i - 1][j - numbers[i - 1]];

}

}

int diff = INT\_MAX;

int firstSum = total, secondSum = 0;

for (int i = sum; i >= 0 ; --i)

{

if (dp[n][i]) {

firstSum = i;

secondSum = total - firstSum;

diff = secondSum - firstSum;

break;

}

}

printSubsets(dp, numbers, firstSum);

return diff;

}

int main(){

INPUT();

int n;

cin >> n;

vector<int> input(n);

for (int i = 0; i < n; ++i)

cin >> input[i];

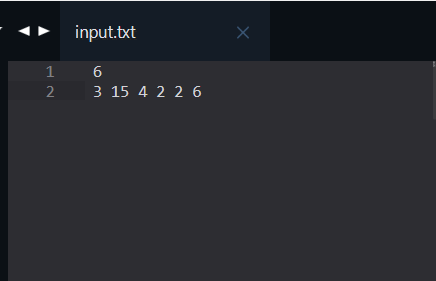
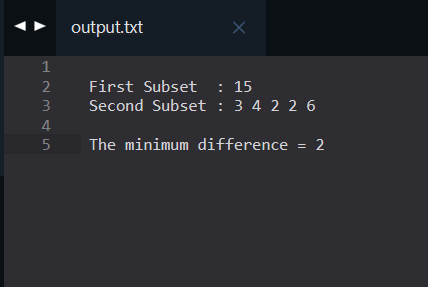
int ans = findMin(input, n);

cout << "The minimum difference = " << ans;

return 0;

}

**Output**

**LAB : 3**

**OBJECTIVE :**

WAP to implement the genetic algorithm.

**Requirements :**

* Windows/ Mac / Linux Pc
* JDK installed (here using JDK 15)
* Text Editor / IDE (here using VS Code)

**Problem Statement :**

Create a random pouulation of size ‘n’ , where each individual (technically a chromosome, which is a solution also) is represented by a binary string of 0’s and 1’s (initialized randomly). In other words each gene can have two values 0 and 1 , and string of ‘n’ genes represents an individual. We nedd to apply Algorithm (GA) to get a solution with all 1’s.

**Implementation :**

**Individual.java :**

package Lab3;

import java.lang.Math;

public class Individual {

private int[] chromosome;

private double fitness = -1;

public Individual(int[] chromosome) {

this.chromosome = chromosome;

}

public Individual(int chromosomeLength) {

this.chromosome = new int[chromosomeLength];

for (int gene = 0; gene < chromosomeLength; gene++) {

if (0.5 < Math.random()) {

this.setGene(gene, 1);

} else {

this.setGene(gene, 0);

}

}

}

public int[] getChromosome() {

return this.chromosome;

}

public int getChromosomeLength() {

return this.chromosome.length;

}

public void setGene(int offset, int gene) {

this.chromosome[offset] = gene;

}

public int getGene(int offset) {

return this.chromosome[offset];

}

public void setFitness(double fitness) {

this.fitness = fitness;

}

public double getFitness() {

return this.fitness;

}

public String toString() {

String output = "";

for (int gene = 0; gene < this.chromosome.length; gene++) {

output += this.chromosome[gene];

}

return output;

}

}

**Populaton.java :**

package Lab3;

import java.util.Arrays;

import java.util.Comparator;

import java.util.Random;

public class Population {

private Individual population[];

private double populationFitness = -1;

public Population(int populationSize) {

this.population = new Individual[populationSize];

}

public Population(int populationSize, int chromosomeLength) {

this.population = new Individual[populationSize];

for (int individualCount = 0; individualCount < populationSize;

individualCount++) {

Individual individual = new Individual(chromosomeLength);

this.population[individualCount] = individual;

}

}

public Individual[] getIndividuals() {

return this.population;

}

public Individual getFittest(int offset) {

Arrays.sort(this.population, new Comparator<Individual>() {

@Override

public int compare(Individual o1, Individual o2) {

if (o1.getFitness() > o2.getFitness()) {

return -1;

} else if (o1.getFitness() < o2.getFitness()) {

return 1;

}

return 0;

}

});

return this.population[offset];

}

public void setPopulationFitness(double fitness) {

this.populationFitness = fitness;

}

public double getPopulationFitness() {

return this.populationFitness;

}

public int size() {

return this.population.length;

}

public Individual setIndividual(int offset, Individual individual) {

return population[offset] = individual;

}

public Individual getIndividual(int offset) {

return population[offset];

}

public void shuffle() {

Random rnd = new Random();

for (int i = population.length - 1; i > 0; i--) {

int index = rnd.nextInt(i + 1);

Individual a = population[index];

population[index] = population[i];

population[i] = a;

}

}

}

**GeneticAlgorithm.java :**

package Lab3;

public class GeneticAlgorithm {

private int populationSize;

private double mutationRate;

private double crossoverRate;

private int elitismCount;

public GeneticAlgorithm(int populationSize, double mutationRate, double crossoverRate, int elitismCount) {

this.populationSize = populationSize;

this.mutationRate = mutationRate;

this.crossoverRate = crossoverRate;

this.elitismCount = elitismCount;

}

Population population = new Population(this.populationSize, chromosomeLength);

return population;

}

public double calcFitness(Individual individual) {

int correctGenes = 0;

for (int geneIndex = 0; geneIndex < individual.getChromosomeLength();

geneIndex++) {

if (individual.getGene(geneIndex) == 1) {

correctGenes += 1;

}

}

double fitness = (double) correctGenes / individual.getChromosomeLength();

individual.setFitness(fitness);

return fitness;

}

public void evalPopulation(Population population) {

double populationFitness = 0;

for (Individual individual : population.getIndividuals()) {

populationFitness += calcFitness(individual);

}

population.setPopulationFitness(populationFitness);

}

public boolean isTerminationConditionMet(Population population) {

for (Individual individual : population.getIndividuals()) {

if (individual.getFitness() == 1) {

return true;

}

}

return false;

}

public Individual selectParent(Population population) {

Individual individuals[] = population.getIndividuals();

// Spin roulette wheel

double populationFitness = population.getPopulationFitness();

double rouletteWheelPosition = Math.random() \* populationFitness;

// Find parent

double spinWheel = 0;

for (Individual individual : individuals) {

spinWheel += individual.getFitness();

if (spinWheel >= rouletteWheelPosition)

return individual;

}

return individuals[population.size() - 1];

}

public Population crossoverPopulation(Population population) {

Population newPopulation = new Population(population.size());

for (int populationIndex = 0; populationIndex < population.size(); populationIndex++) {

Individual parent1 = population.getFittest(populationIndex);

if (this.crossoverRate > Math.random() && populationIndex >=

this.elitismCount) {

Individual offspring = new Individual(parent1.getChromosomeLength( ));

Individual parent2 = selectParent(population);

for (int geneIndex = 0; geneIndex < parent1.getChromosomeLength(); geneIndex++) {

if (0.5 > Math.random()) {

offspring.setGene(geneIndex, parent1.getGene(geneIndex));

} else {

offspring.setGene(geneIndex, parent2.getGene(geneIndex));

}

}

newPopulation.setIndividual(populationIndex, offspring);

} else {

newPopulation.setIndividual(populationIndex, parent1);

}

}

return newPopulation;

}

public Population mutatePopulation(Population population) {

Population newPopulation = new Population(this.populationSize);

for (int populationIndex = 0; populationIndex < population.size(); populationIndex++) {

Individual individual = population.getFittest(populationIndex);

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);

}

}

}

newPopulation.setIndividual(populationIndex, individual);

}

return newPopulation;

}

}

**App.java :**

//import Lab3.Individual;

import Lab3.Population;

import Lab3.GeneticAlgorithm;

public class App {

private static final int numberOfBits = 20;

public static void main(String[] args) {

GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.001, 0.95, 2);

Population population = ga.initPopulation(numberOfBits);

ga.evalPopulation(population);

int generation = 1;

while (ga.isTerminationConditionMet(population) == false) {

System.out.println("Best solution: " + population.getFittest(0).toString());

population = ga.crossoverPopulation(population);

population = ga.mutatePopulation(population);

ga.evalPopulation(population);

generation++;

}

System.out.println("Found solution in " + generation + " generations");

System.out.println("Best solution: " + population.getFittest(0).toString());

}

}

**Output :**

