Yasaman Mirmohammad 9431022

Implementation of Algorithms:

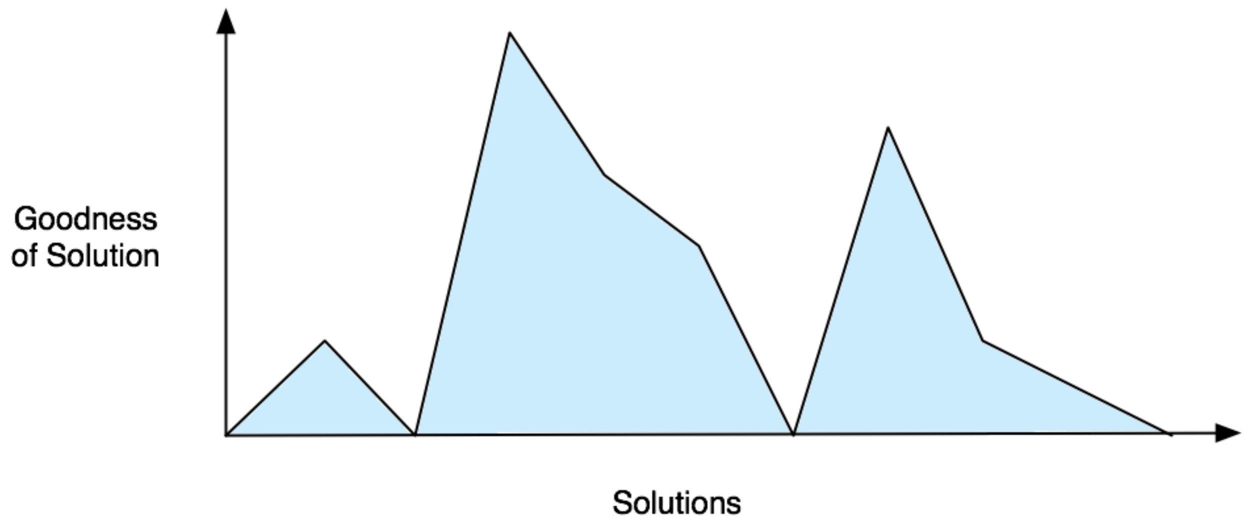
- Simulated Annealing
- Hill climbing
- Genetic

SA:

Language:Python

Simulated annealing is a method for finding a good (not necessarily perfect) solution to an optimization problem.

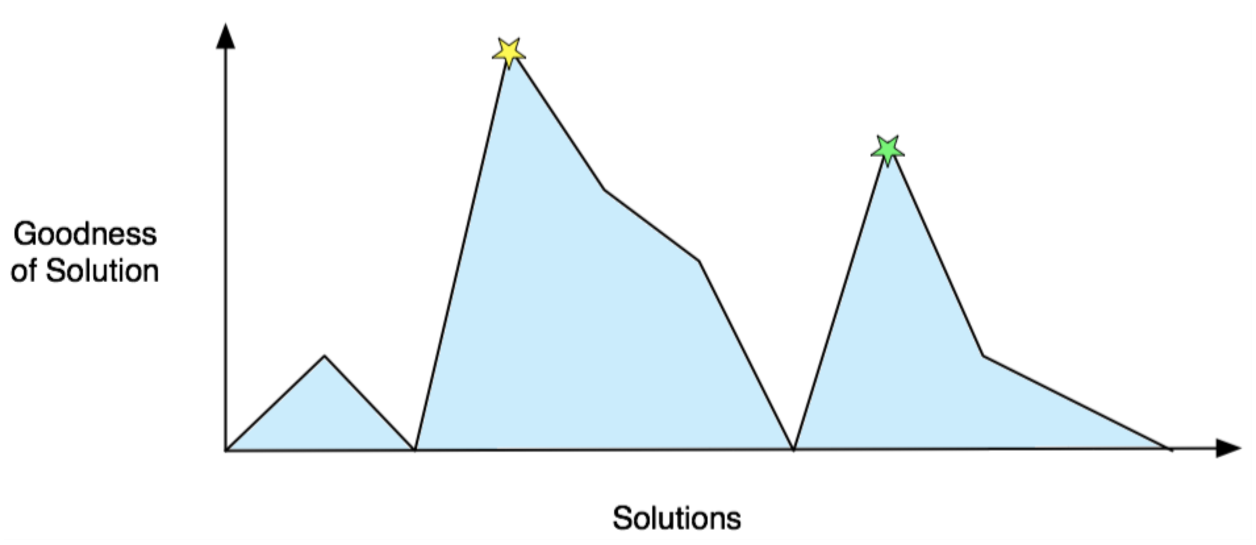
You can visualize this by imagining a 2D graph like the one below. Each x-coordinate represents a particular solution (e.g., a particular itinerary for the salesman). Each y-coordinate represents how good that solution is (e.g., the inverse of that itinerary's mileage).



an optimization algorithm searches for the best solution by generating a random initial solution and "exploring" the area nearby. If a neighboring solution is better than the current one, then it moves to it. If not, then the algorithm stays put.

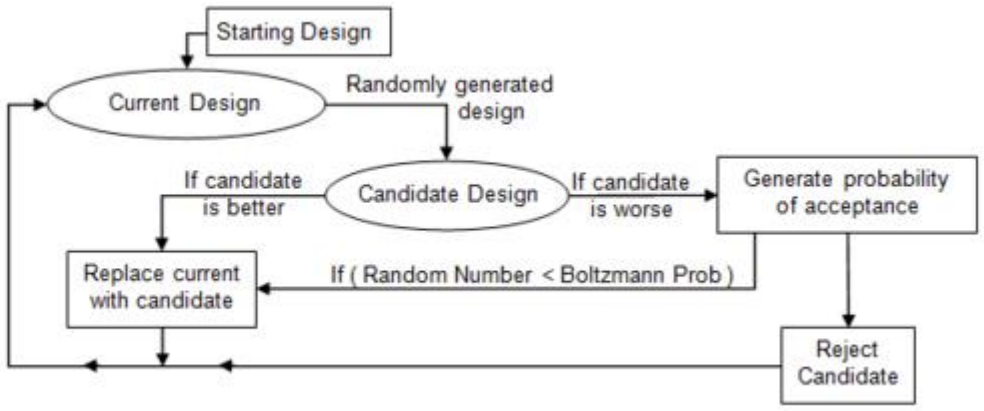
This is perfectly logical, but it can lead to situations where you're stuck at a sub-optimal place.

In the graph below, the best solution is at the yellow star on the left. But if a simple algorithm finds its way to the green star on the right, it won't move away from it: all of the neighboring solutions are worse. The green star is a local maximum.



Steps:

1. First, generate a random solution
2. Calculate its cost using some cost function you've defined
3. Generate a random neighboring solution
4. Calculate the new solution's cost
5. Compare them:
   - If $c_{new} < c_{old}$: move to the new solution
   - If $c_{new} > c_{old}$: *maybe* move to the new solution
6. Repeat steps 3-5 above until an acceptable solution is found or you reach some maximum number of iterations.


Starting Design
Current Design
Randomly generated design
If candidate is better
Candidate Design
If candidate is worse
Generate probability of acceptance
Replace current with candidate
If (Random Number < Boltzmann Prob)
Reject Candidate

A useful additional optimization is to always keep track of the best solution found so far so that it can be returned if the algorithm terminates at a sub-optimal place.

Pseudo Code:

```
i <- generate an individual randomly
best_so_far <- i
while stopping criterion has not been met:
    get i's bit string and convert it to the problem representation (int or float)
    increment or decrement one of the genes by the step size
    if the resulting individual has higher fitness
        replace i with this individual and increase the step size
    else
        decrease the step size
    if the step size reaches zero and increments and decrements of the current gene have been tested
        move on to the next gene
    if i is at a local optimum
        if fitness(i) > fitness(best_so_far)
            best_so_far <- i
        i <- generate an individual randomly
```

Boltzmann probability factor:

$$P = \exp\left(\frac{-\Delta E}{k_b T}\right) = e^{\left(-\Delta E / k_b T\right)}$$

```
p = math.exp(-DeltaE / (DeltaE_avg * t))
```

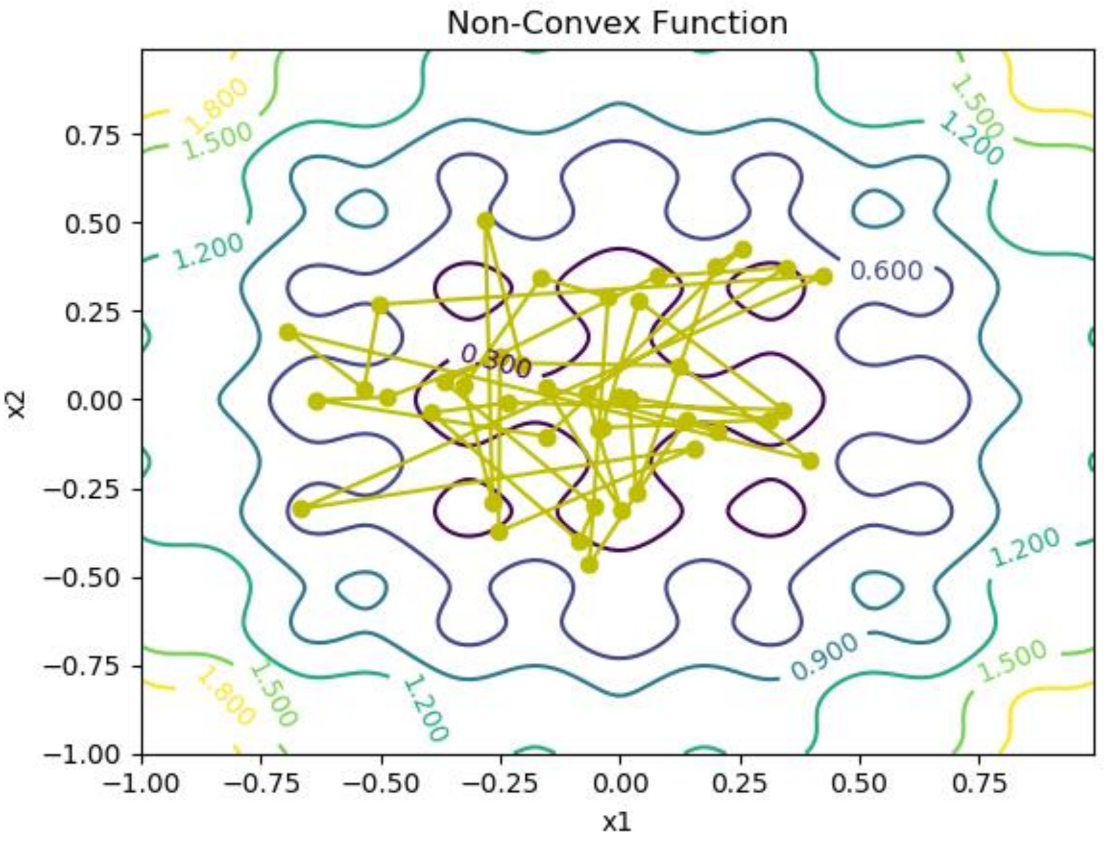
where k_b is the Boltzmann constant and T is the current temperature. By examining this equation we should note two things: the probability is proportional to temperature--as the solid cools, the probability gets smaller; and inversely proportional to --as the change in energy is larger the probability of accepting the change gets smaller.

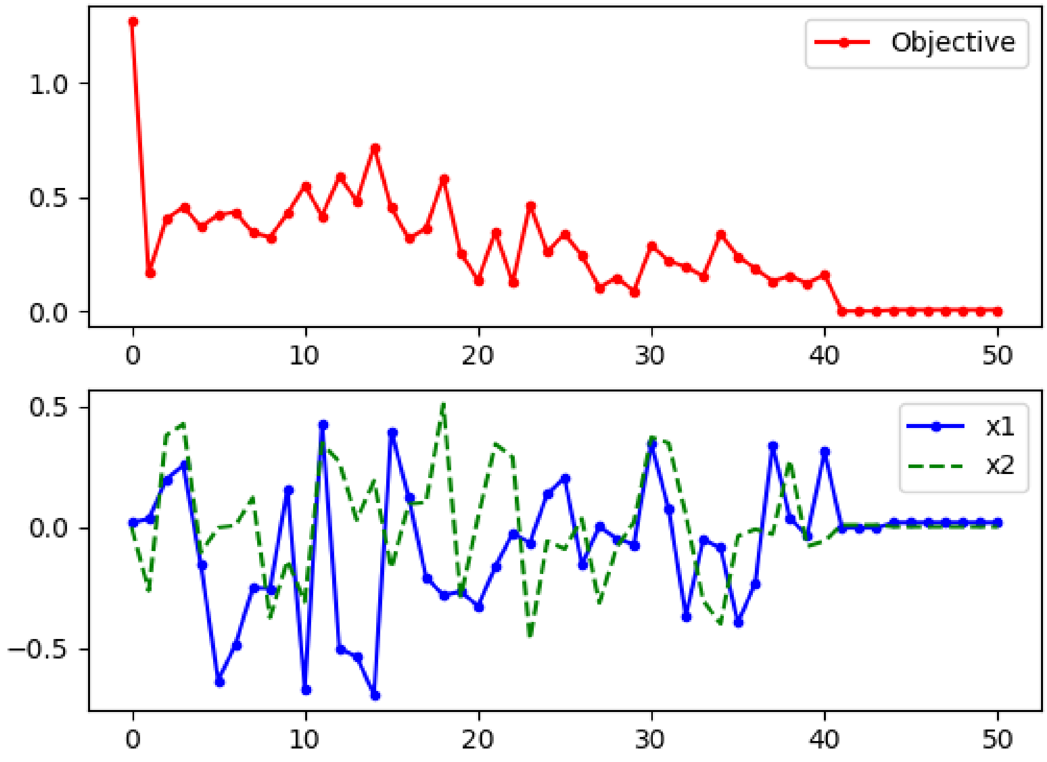
Result:

```
Cycle: 0 in Temperature: 2.8036732520571284

Cycle: 1 in Temperature: 2.6391299733669826

Cycle: 2 in Temperature: 2.484243487080245
```

```
Cycle: 3 in Temperature: 2.33844705087681

Cycle: 4 in Temperature: 2.2012071836732217

Cycle: 5 in Temperature: 2.0720217135717602

Cycle: 6 in Temperature: 1.9504179403723987

Cycle: 7 in Temperature: 1.8359509059241148

Cycle: 8 in Temperature: 1.7282017659866262

Cycle: 9 in Temperature: 1.6267762576450626

Cycle: 10 in Temperature: 1.531303256669716

Cycle: 11 in Temperature: 1.4414334195421339

Cycle: 12 in Temperature: 1.3568379051786155

Cycle: 13 in Temperature: 1.2772071716737936

Cycle: 14 in Temperature: 1.202249843661488

Cycle: 15 in Temperature: 1.1316916461484114

Cycle: 16 in Temperature: 1.0652744009195394

Cycle: 17 in Temperature: 1.002755081842906

Cycle: 18 in Temperature: 0.9439049256171138

Cycle: 19 in Temperature: 0.8885085947077039

Cycle: 20 in Temperature: 0.836363389409508

Cycle: 21 in Temperature: 0.7872785061518497

Cycle: 22 in Temperature: 0.7410743393326751

Cycle: 23 in Temperature: 0.6975818241269669

Cycle: 24 in Temperature: 0.6566418178647233

Cycle: 25 in Temperature: 0.6181045177149135

Cycle: 26 in Temperature: 0.5818289125446678

Cycle: 27 in Temperature: 0.5476822669480108
```

```
Cycle: 28 in Temperature: 0.5155396355561552

Cycle: 29 in Temperature: 0.4852834058521797

Cycle: 30 in Temperature: 0.45680286781721074

Cycle: 31 in Temperature: 0.4299938088334096

Cycle: 32 in Temperature: 0.4047581323614812

Cycle: 33 in Temperature: 0.38100349899741437

Cycle: 34 in Temperature: 0.3586429885950507

Cycle: 35 in Temperature: 0.33759478221816164

Cycle: 36 in Temperature: 0.3177818627582694

Cycle: 37 in Temperature: 0.29913173312274866

Cycle: 38 in Temperature: 0.28157615096203553

Cycle: 39 in Temperature: 0.2650508789652898

Cycle: 40 in Temperature: 0.249495449810821

Cycle: 41 in Temperature: 0.23485294491121345

Cycle: 42 in Temperature: 0.2210697861435598

Cycle: 43 in Temperature: 0.20809553980272782

Cycle: 44 in Temperature: 0.19588273206030876

Cycle: 45 in Temperature: 0.18438667525399666

Cycle: 46 in Temperature: 0.17356530437177747

Cycle: 47 in Temperature: 0.16337902313260994

Cycle: 48 in Temperature: 0.15379055910039502

Cycle: 49 in Temperature: 0.14476482730108406

Best solution: [0.01944931 0.00153748]

Best objective: 0.007067488493017082
```


Non-Convex Function
x1
x2
0.300
0.600
0.900
1.200
1.500
1.800
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75



As the temperature is gradually lowered, the probability that a worse design is accepted becomes smaller. Typically at high temperatures the gross structure of the design emerges which is then refined at lower temperatures.

2.Hill Climbing:

Language:Java

1. Define the current state as an initial state
2. Loop until the goal state is achieved or no more operators can be applied on the current state:
    1. Apply an operation to current state and **get a new state**

2. **Compare** the new state with the goal
3. **Quit** if the goal state is achieved
4. Evaluate new state with heuristic function and **compare it with the current state**
5. **If the newer state is closer** to the goal compared to current state, **update the current state**

Pseudo code:

```
Discrete Space Hill Climbing Algorithm
   currentNode = startNode;
   loop do
      L = NEIGHBORS(currentNode);
      nextEval = -INF;
      nextNode = NULL;
      for all x in L
         if (EVAL(x) > nextEval)
              nextNode = x;
              nextEval = EVAL(x);
      if nextEval <= EVAL(currentNode)
         //Return current node since no better neighbors exist
         return currentNode;
      currentNode = nextNode;
```

```
Continuous Space Hill Climbing Algorithm
   currentPoint = initialPoint;    // the zero-magnitude vector is common
   stepSize = initialStepSizes;    // a vector of all 1's is common
   acceleration = someAcceleration; // a value such as 1.2 is common
   candidate[0] = -acceleration;
   candidate[1] = -1 / acceleration;
   candidate[2] = 0;
   candidate[3] = 1 / acceleration;
   candidate[4] = acceleration;
   loop do
      before = EVAL(currentPoint);
      for each element i in currentPoint do
         best = -1;
         bestScore = -INF;
         for j from 0 to 4         // try each of 5 candidate locations
            currentPoint[i] = currentPoint[i] + stepSize[i] *
candidate[j];
            temp = EVAL(currentPoint);
            currentPoint[i] = currentPoint[i] - stepSize[i] *
candidate[j];
            if(temp > bestScore)
                 bestScore = temp;
                 best = j;
         if candidate[best] is 0
            stepSize[i] = stepSize[i] / acceleration;
         else
            currentPoint[i] = currentPoint[i] + stepSize[i] *
candidate[best];
            stepSize[i] = stepSize[i] * candidate[best]; // accelerate
      if (EVAL(currentPoint) - before) < epsilon
         return currentPoint;
```

3.Genetic Algorithm

Genetic algorithms are a part of evolutionary computing, which is a rapidly growing area of artificial intelligence.

An algorithm starts with a **set of solutions** (represented by **individuals**) called **population**. Solutions from one population are taken and used to form a **new**

population, as there is a chance that the new population will be better than the old one.

Individuals that are chosen to form new solutions (**offspring**) are selected according to their **fitness** — the more suitable they are, the more chances they have to reproduce.

3.1. Initialization

In the initialization step, we **generate a random *Population* that serves as a first solution**. First, we need to decide how big the *Population* will be and what is the final solution that we expect:

3.2. Fitness Check

In the main loop of the program, we are going to **evaluate each *Individual* by the fitness function** (in simple words, the better the *Individual* is, the higher value of fitness function it gets):

3.3. Offspring

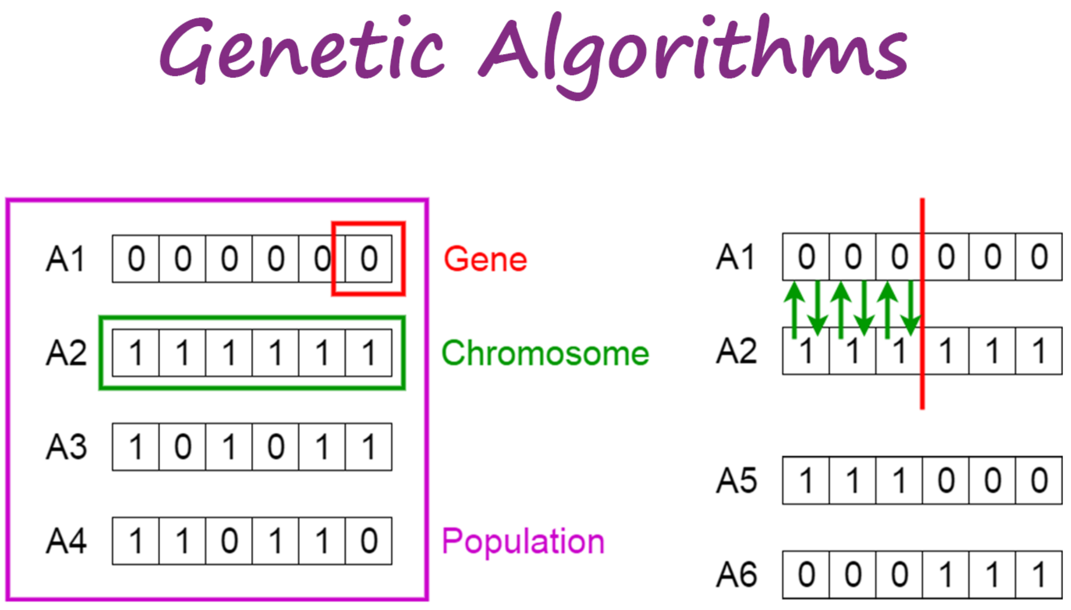
In this step, we need to create a new *Population*. First, we need to **Select** two parent *Individual* objects from a *Population*, according to their fitness. Please note that it is beneficial to allow the best *Individual* from the current generation to carry over to the next, unaltered. This strategy is called an **Elitism:**

after the crossover, we place new offspring in a new *Population*. This step is called the **Acceptance.**

Finally, we can perform a **Mutation**. Mutation is used to maintain genetic diversity from one generation of a *Population* to the next. We used the **bit inversion** type of mutation, where random bits are simply inverted:

In order to **implement an efficient genetic algorithm**, we need to tune a set of parameters. This section should give you some basic recommendations how to start with the most importing parameters:

- **Crossover rate** – it should be high, about **80%-95%**
- **Mutation rate** – it should be very low, around **0.5%-1%**.
- **Population size** – good population size is about **20-30**, however, for some problems sizes 50-100 are better
- **Selection** – basic **roulette wheel selection** can be used with the concept of **elitism**
- **Crossover and mutation type** – it depends on encoding and the problem



Initial Population

The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve.

An individual is characterized by a set of parameters (variables) known as **Genes**. Genes are joined into a string to form a **Chromosome** (solution).

In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.

Fitness Function

The **fitness function** determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a **fitness score** to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

Selection

The idea of **selection** phase is to select the fittest individuals and let them pass their genes to the next generation.

Two pairs of individuals (**parents**) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a **crossover point** is chosen at random from within the genes.

Mutation

In certain new offspring formed, some of their genes can be subjected to a **mutation** with a low random probability. This implies that some of the bits in the bit string can be flipped.

Mutation occurs to maintain diversity within the population and prevent premature convergence.

Termination

The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.

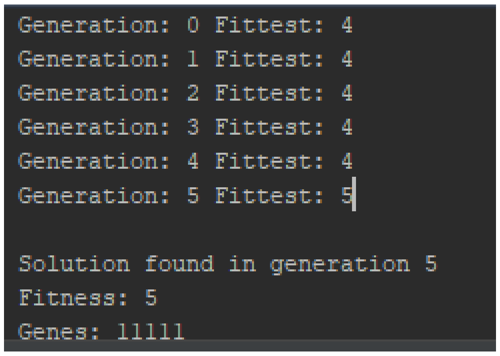
Psuedocode

```
START
Generate the initial population
Compute fitness
REPEAT
    Selection
    Crossover
    Mutation
    Compute fitness
UNTIL population has converged
STOP
```

Results:

```
//Do mutation under a random probability
if (random.nextInt() % 7 < 5) {
    genetic.mutation();
}
```

```
Generation: 0 Fittest: 4

Generation: 1 Fittest: 4

Generation: 2 Fittest: 4

Generation: 3 Fittest: 4

Generation: 4 Fittest: 4

Generation: 5 Fittest: 5


Solution found in generation 5

Fitness: 5

Genes: 11111
```



```
Generation: 0 Fittest: 4
Generation: 1 Fittest: 4
Generation: 2 Fittest: 4
Generation: 3 Fittest: 4
Generation: 4 Fittest: 4
Generation: 5 Fittest: 5

Solution found in generation 5
Fitness: 5
Genes: 11111
```

Generation: 0 Fittest: 3

Generation: 1 Fittest: 3

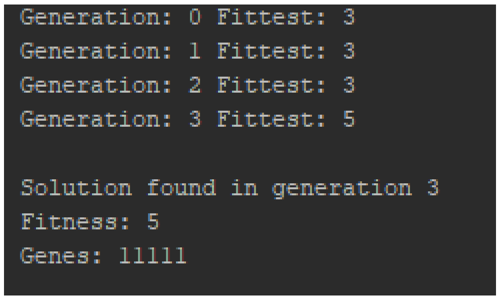
Generation: 2 Fittest: 3

Generation: 3 Fittest: 5

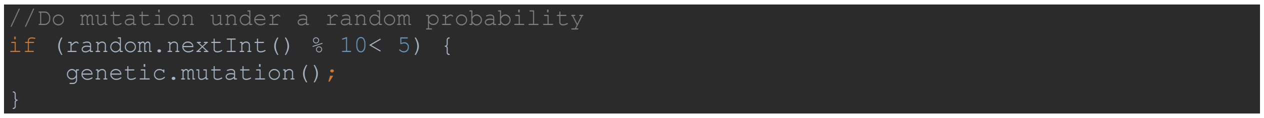
Solution found in generation 3

Fitness: 5

Genes: 11111



```
Generation: 0 Fittest: 3
Generation: 1 Fittest: 3
Generation: 2 Fittest: 3
Generation: 3 Fittest: 5

Solution found in generation 3
Fitness: 5
Genes: 11111
```



```
//Do mutation under a random probability
if (random.nextInt() % 10< 5) {
    genetic.mutation();
}
```

Solution found in generation 12

Fitness: 5

Genes: 11111

Problem Solving: Keyboard:

START

First we should omit the words containing alphabets that don’t exist in the table.