**Objective**

Use Hadoop Map-Reduce to evolve Genetic Algorithm by parallelizing the operations and output the solution for the target given the various parameters like chromosome length, mutation rate, crossover rate and population size.

**Assumption**

In genetic algorithm, the two good (but not best) chromosomes should be selected so that the good characteristics of parents are included in Crossover and best will remain in the population. For this implementation, we are assuming that the chromosomes which have the high fittest score will crossover and mutate to give the best output solution.

**Solution**

Based on the serial GA, we can naturally come up with the Mapper and Reducer design. Several GA parallelization strategies exist depending on the grain of parallelization to achieve. Basically, three levels of parallelization can be exploited:

* fitness evaluation level (i.e., global parallelization model);
* population level (i.e., coarse-grained parallelization or island model);
* individual level (i.e., fine-grained parallelization or grid model).

In the global parallelization model, a node acting as a master, manages the population (i.e., applying genetic and selection operators) and distributes the individuals among slave nodes which compute only the fitness values of the individuals. The main advantage of using such a model is that it does not require any change to the design of traditional GA since the individual fitness evaluation is independent from the rest of the population.

The underlying idea in using MapReduce to parallelize the Genetic Algorithm is to encapsulate each iteration of the GA as a separate MapReduce job and parallelize the chromosome fitness evaluation assigning such task to several Mappers, while a single Reducer is responsible to collect the results and to perform the genetic operations (i.e., parent’s selection, crossover and mutation, and survival selection)

needed to produce a new generation following a global parallelization model. The MapReduceJob is the

core of the Parallel Genetic Algorithm module since it allows us to parallelize fitness evaluations and

distribute the computation over the nodes. A MapReduceJob consists of three phases (i.e., Split, Map, and

Reduce) in which each component performs its proper task.

In the split phase, the InputFormat module gets the current population and processes it in order to split it in crunch of data (i.e., input split) to be distributed among the Mapper modules.

In the map phase, each Mapper carries out its task on the received input split in a parallel and independent way.

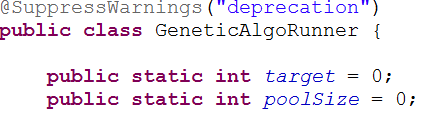
Once such evaluation is completed, each Mapper generates a new pair, where value is a pair, while the new key will be used by the Master module to properly assign the Reducers.**Code Explanation**

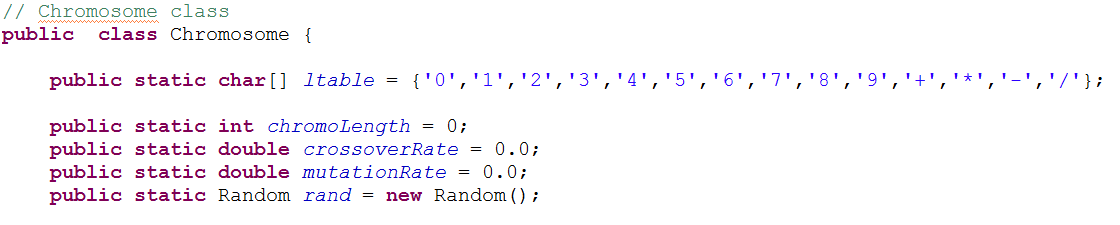
1. **Read the Input File**

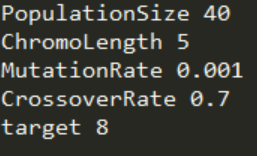
In the first MapReduce job, we have only mapper, the **ReadMapper** is going to read the input file which will have parameters like population size, crossover rate, mutation rate, chromosome length and target. We are parametrizing or taking the inputs from the users, which will make the job flexible. Mutation rate (**mutationRate**) should be very low, we have took it as **0.001** for running the algorithm. Crossover rate (**crossoverRate**) of **0.7** is good for the Genetic Algorithm. Population size we have taken as **40**, but we have run it for various population size, it is called as **poolSize**. Chromosome length which we are considering for this iteration is **5** and we are calling it as **chromoLength**. Chromosome length should be odd, because then only expression will be valid. For expression should be valid the number should be followed by an operator and end in a number. We have a function called **isValid()**  for validating the function in the Chromosome class. We are also defining the **target** value in the input file, what target we want’s to achieve using genetic algorithm, the value we have taken is **8**.

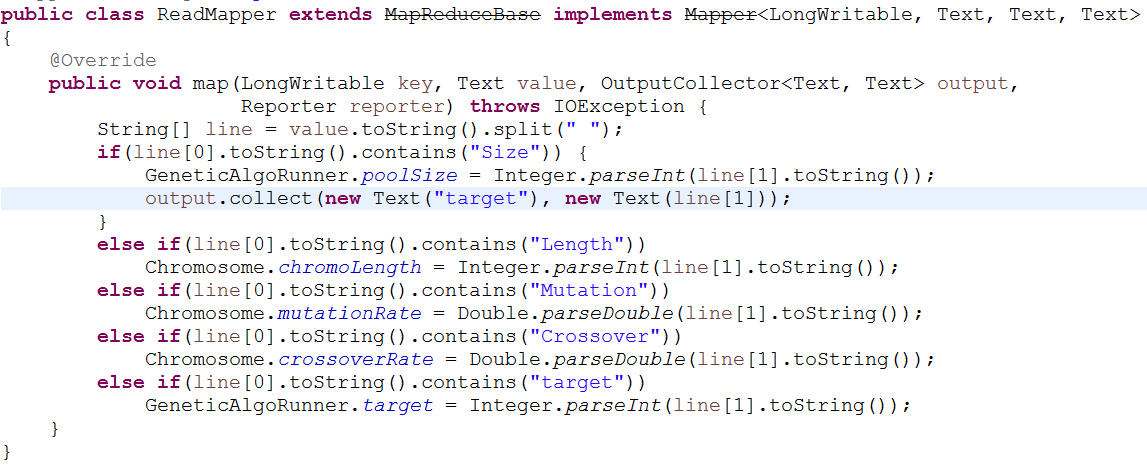


We are going to read the input file and set the value of global variable which we have defined in Chromosome and GeneticAlgoRunner class. Mutation Rate, crossover rate, chromosome length are declared in Chromosome class, Pool size and target in GeneticAlgoRunner class





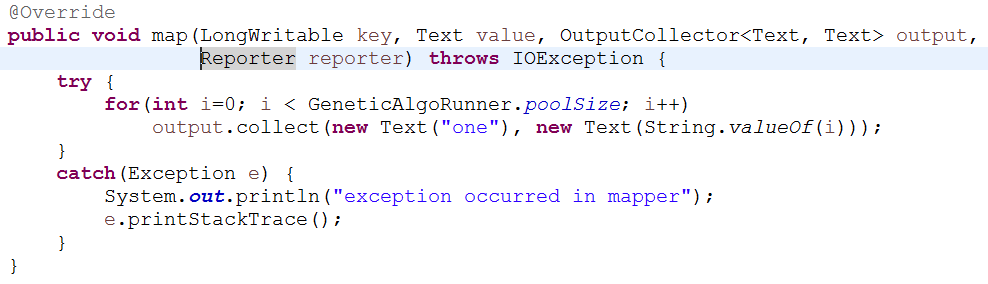




1. **Creation of unique chromosomes**

In the second MapReduce Job, in mapper, we are going to create new chromosome and compute the fitness score given the population size as 40 and each chromosome of 4 bit.

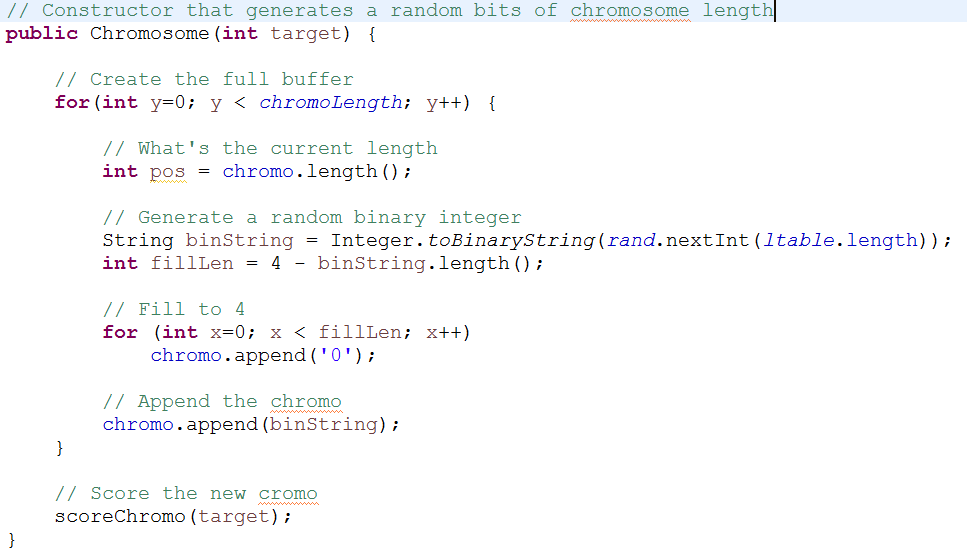
In **ChromosomeMapper**, we are going to iterate through the size of the population and emit the “**one**” as key and the chromosome number i.e, “**index number”** as value. We are emitting all the population with the same key because the population is not that big and we can generate all the chromosome in a single place.There is no use taking them in multiple reducers since our population size is not that huge. If in future, we want to take the larger population size we can employ multiple reducers so that we can distribute our work in that case.



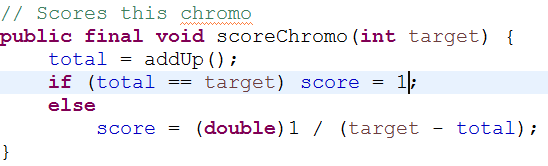
**Input**: population size (refer to pool size in code)

**Output**: Chromosome numbers to be given to one reducer

In **ChromosomeReducer**, we are going to generate new Chromosomes by looping over the chromosome number which were emitted by the mapper. After generating new chromosomes, we can compute the score of the chromosome. The below function is going to generate a random bit and create a chromosome of chromosome length defined based on the target value.



Once the chromosome is generated, we have a function called **scoreChromo()** which is going to compute the fitness score of the chromosome. The general idea is to give a higher fitness score the closer a chromosome comes to solving a problem. The score of the chromosome is **1 / (target – total)** where target is the expected value and total is computed value of that chromosome. The closed we get to a solution, the closed we get to a 1. Therefore, when target == total, then the score will be one. Assume the target number of the remainder of the story is 42 and the chromosome is 23, then the score of that chromosome will be **1 / (42 – 23) = (1 / 19)**.

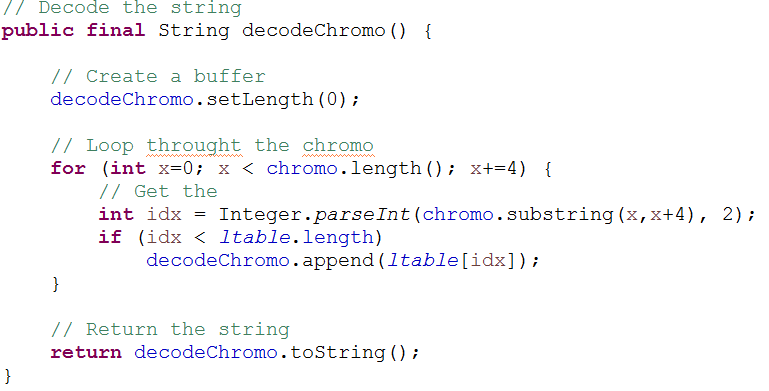


We are going to use **addUp()** function, to compute the total of the chromosome, so that we can compute the fitness score. The **addUp() function** is going to use the **decode()** function, in which we are going to use **ltable**. Ltable is a character array having value **{'0','1','2','3','4','5','6','7','8','9','+','\*','-','/'}.** We are going to take the 4 bits of the chromosome and convert it into decimal and then decode it to one of the element from the ltable. In our use case **1 is represented 0001, 2 as 0010, 9 as 1001, ‘+’ as 1010, ‘\*‘ as 1011, ‘-‘ as 1100 and ‘/’ as 1101**.



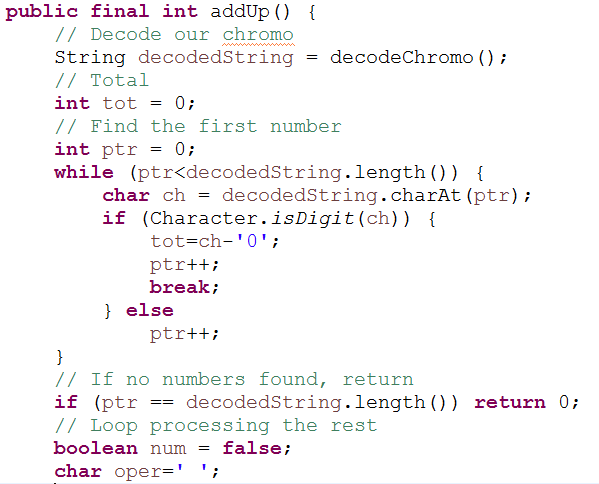
**Decode Function**: Decode the chromosome from binary string to number and operator.

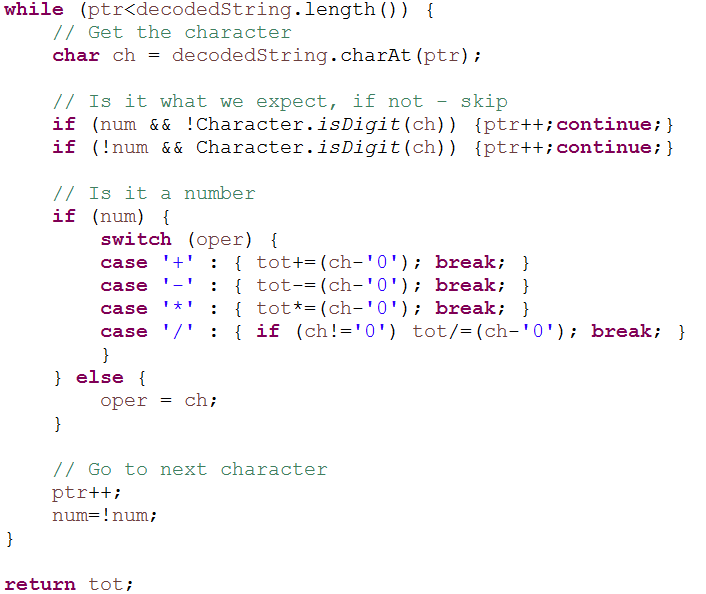
**Example**: **0110 1010 0110 1100 0100 1101 0010 1010 0001. Decoded as: 6 + 6 – 4 / 2 + 1**



**AddUp Function**: is going to add the decode chromosome and operator will be applied sequentially from left to right.

The addUp function will generate the **value as 5** for the above chromosome.





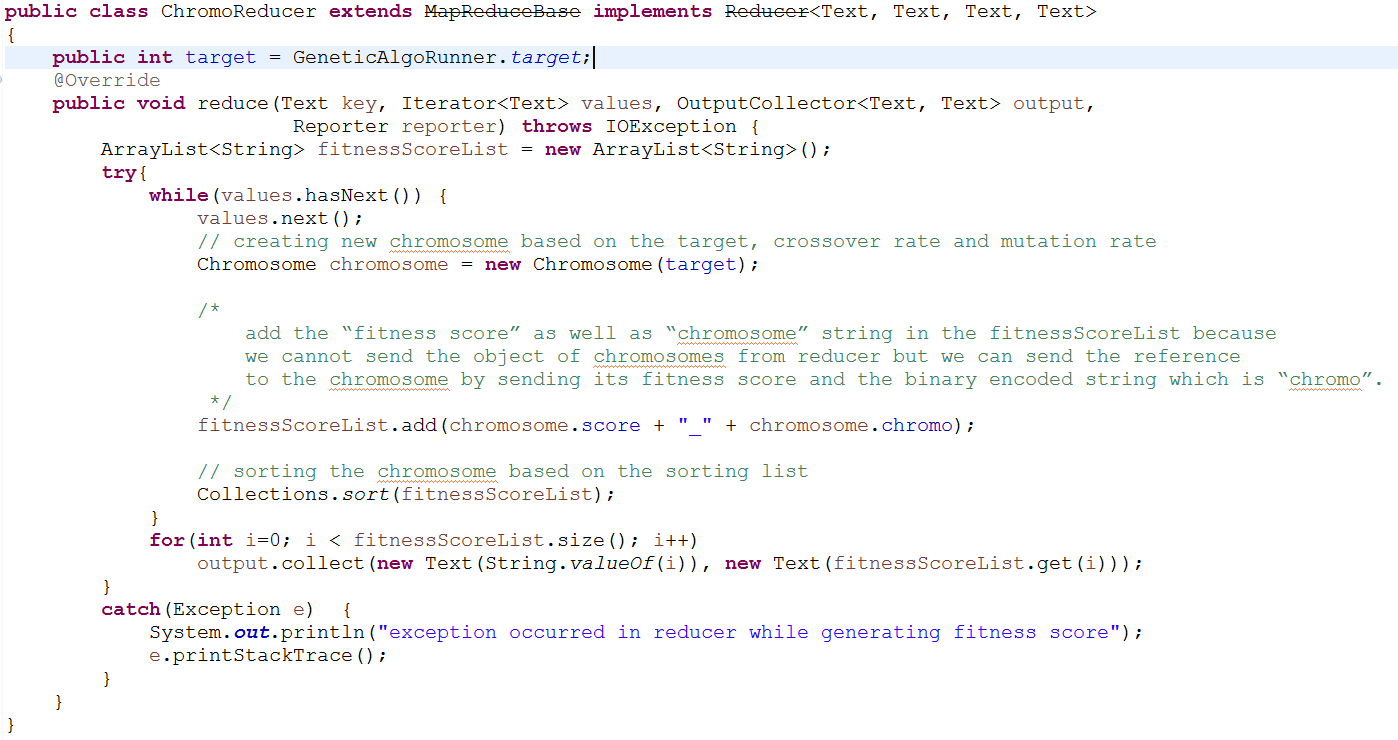
Once we have the chromosome and the fitness score, we will create the fitness score list which will have chromosome number that is basically a line number or **“index of population” as key** and **values as “chromosome score\_chromosomeBits”**. We have to add the “fitness score” as well as “chromosome” string in the fitnessScoreList because we cannot send the object of chromosomes from reducer but we can send the reference the chromosome by sending its fitness score and the binary encoded string which is “chromo”. After putting this, we are going to sort the list based on the fitness score. The keys will be 0 to 39 for the iteration since the we have population size of 40.

**Input**: Chromosome numbers which work just like line numbers

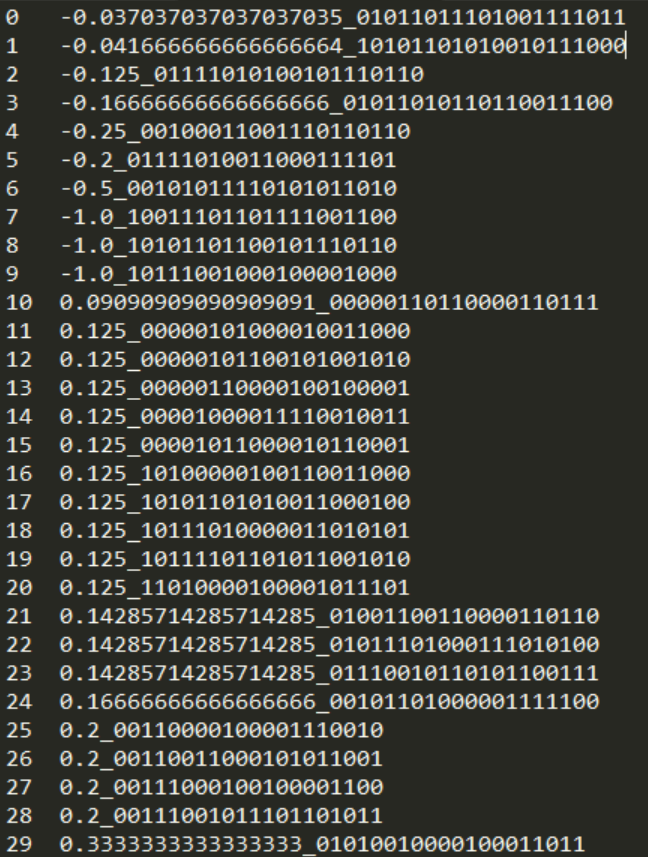
**Output**:

**Key:** chromosome number

**Value:** fitnessScore\_chromo, Example: (0.16666666666666666\_00100010001001010111)



**Screenshot of the Output (Output Directory)**

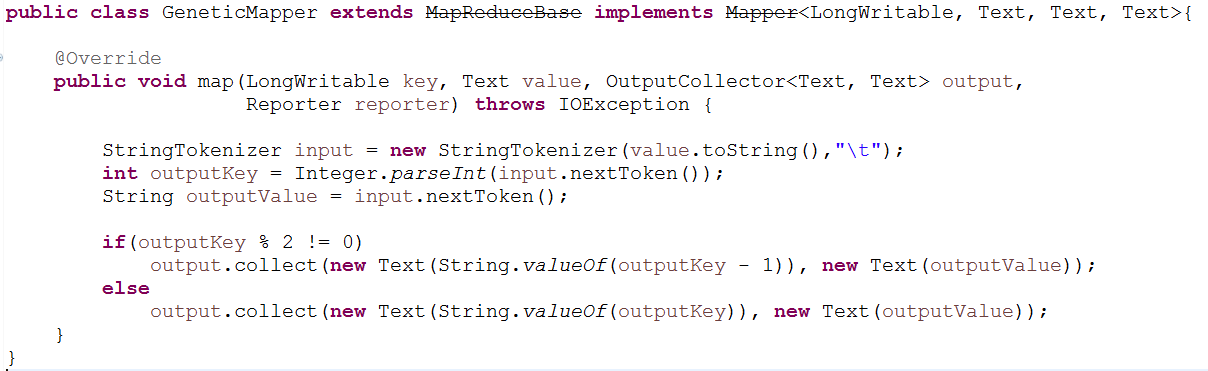


1. **Send the most probable chromosomes to reproduce and mutate to the same Reducer**

* Select two members from the current population, chances of chromosome being selected is proportional to the chromosome fitness score
* Choose bits from each chosen chromosome at a randomly chosen, dependent on the crossover rate, crossover the bits
* Crossover: is performed by selecting a random gene along the length of the chromosomes and swapping all the genes after that point
  + Crossover rate: the chance that two chromosomes will swap their bits. CrossoverRate is 0.7
* Mutation: Step through the chosen chromosomes bits and flip dependent on the mutation rate
  + Mutation rate: this is the chance that a bit within a chromosome will be flipped (0 becomes 1, 1 becomes 0). This is ususally a very low value for binary encoded genes, say 0.001

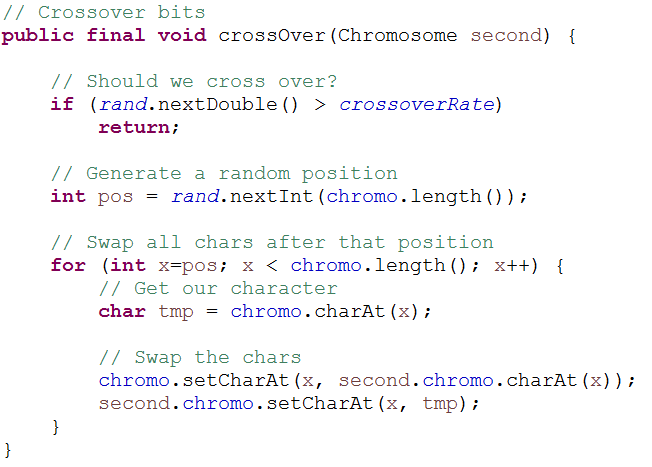
In **GeneticMapper**, we are going to receive the input which was emitted as output from the Chromosome Reducer, it will be in form line number as key and “fitnessScore\_chromo” as value. Since the input will be sorted as per the fitness score, we will emit the same key for consecutive chromosomes. We do this because we want to have the fittest chromosomes to mutate in order to give best output. The chromosomes which are in even place (line number) will emit the **key** that they have been associated

within the output of the above MapReduce. The chromosomes which are in odd place (line number) will emit the **(key – 1)** as key so that they go to the same reducer as their consecutive counterpart. **The number of reducers required will be half of the population size (**because we are sending to pairs to each reducer**).**

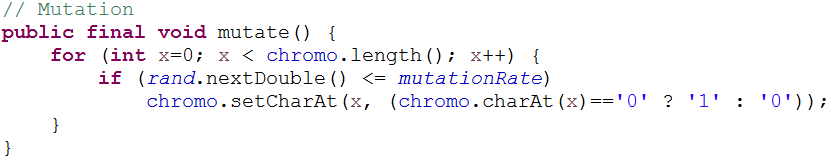


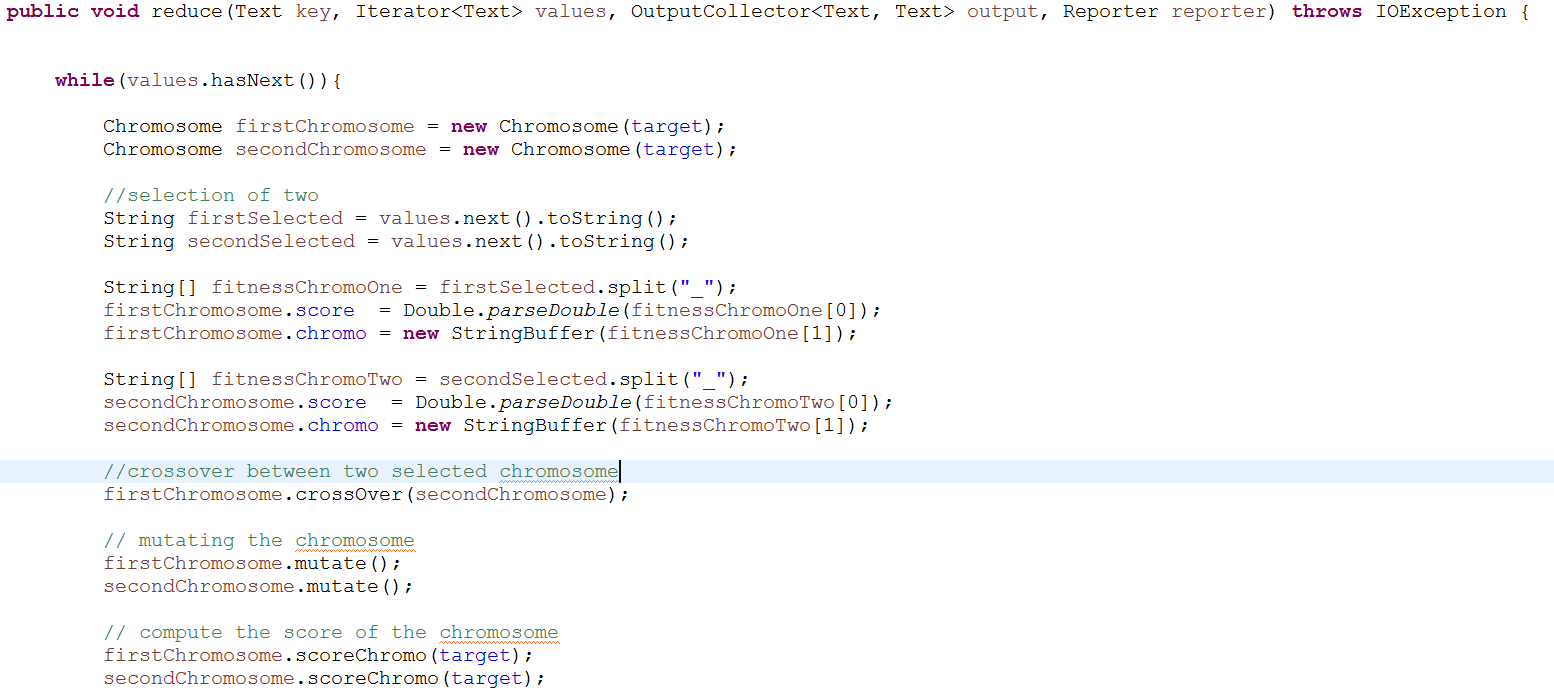
**In GeneticReducer,** we are going to get the pair of chromosomes (Fitness score and chromo which is binary encoded string) as input. After receiving the chromosome, we are going to create chromosome object and the fitness score and chromo to two different chromosome. After doing that, the next step is to perform genetic algorithm

* Crossover the two chromosomes, based on crossover rate which is read from the input file in the ReadMapper



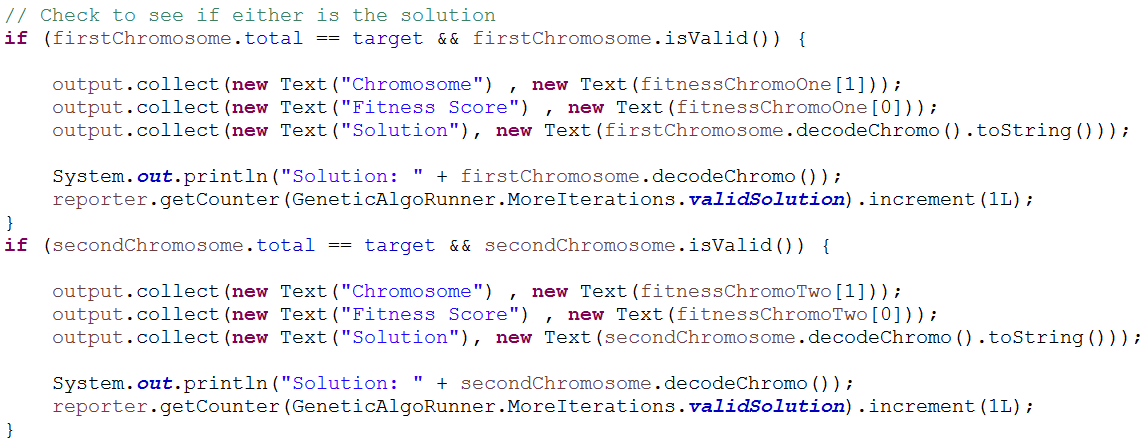
* Mutate the two chromosomes, based on mutation rate which is read from the input file in the ReadMapper





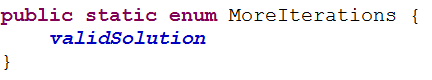
* Check if the total score is equal to the target and if it is valid chromosome (based on number -> operator -> number -> operator -> number)
* Iterate till we get the best chromosome mutation

Once we get the best chromosome, we are going to decode the binary string and output the **“Solution” as key** and **“Decoded chromosome” as value.** Along with that we are going to output the Fitness score and the chromosome.



**Iteration**

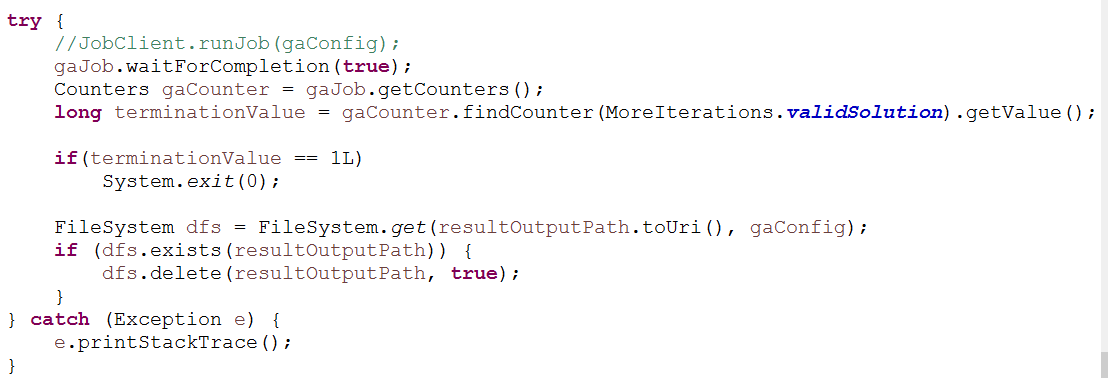
The genetic algorithm is iterative algorithm until we found a good solution, we are going to run the Chromosome MapReduce and Genetic MapReduce in a loop, until we found an appropriate solution. For stopping this Iterative MapReduce, we are using counters, counters are global variable which are defined as enum and are accessed globally in Mapper, Reducer and Driver/Runner.



Once we found an appropriate solution, we are going to increment its value in **GeneticReducer**



And we are constantly check for its value in **GeneticAlgoRunner** class, as soon as we found its value as 1. We are going to stop the iteration once that job is completed.

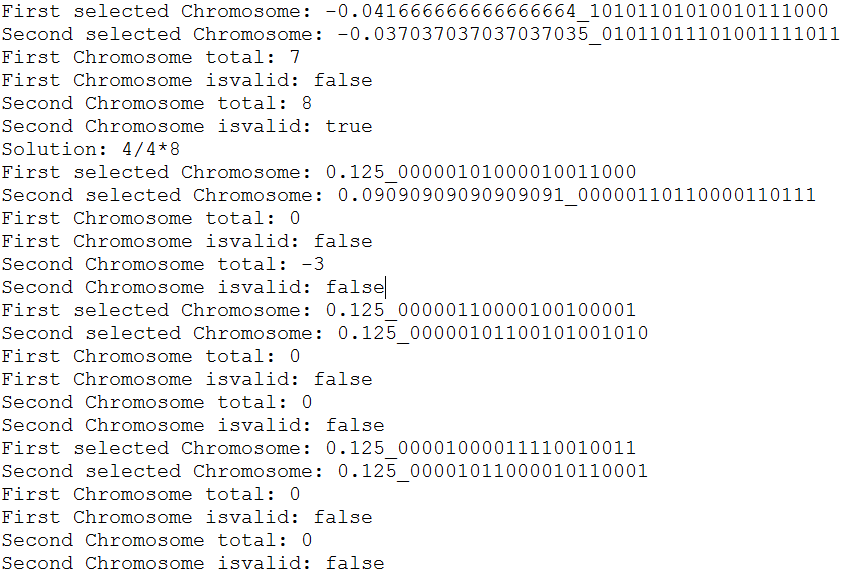


**Conclusion**

Genetic Algorithm is very difficult to run on Hadoop because it involves a lot of computations and takes a lot of iteration to find a single solution. Hence we do not prefer running genetic algorithm on Hadoop.

**Screenshots**

**Console Output**



**Final Output (Result Directory)**

