**Genetic Algorithm to solve Subset Sum Problem**

**Abstract**

The Subset Sum Problem(SSP) is a classical NP-complete problem in computer science. It is about to find subsets in a given number set, meanwhile number sum of the subset is equal to appointed value. In our experiment, we use Genetic Algorithm (GA) to solve the Subset Sum Problem. The GA individuals are obtained by using some rule-based permutations of the facilities, which are then improved towards the optimum by means of specially designed crossover and mutation operators.

**Problem Description**

In computer science, the Subset Sum Problem(SSP) is an important problem in complexity theory which is a classical NP-complete problem in graph theory. Subset Sum Problem is this: given a set of integers and an integer *s*, does any non-empty subset sum to *s*? For example, given the set {−6, −4, −1, 3, 10}, does any non-empty subset sum to 4? The answer is yes because the subset {−6, 10} sums to 4. The problem is [NP-complete](https://en.wikipedia.org/wiki/NP-complete), meaning roughly that while it is easy to confirm whether a proposed solution is valid, it may inherently be prohibitively difficult to determine in the first place whether any solution exists.

？  **sum**

Integer

4

！ **sum**

Integer

4

**Background**

Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection.

Initiliza Population

Evaluate Fitness

Satisfy Constraints?

Crossover and

mutation

Select Survivors

**Yes** **No**

**Procedure**

1. The evolution starts from a population of randomly generated individuals, marked as generation one;
2. Evaluate the fitness of generation one;
3. If the result doesn’t satisfy constrains, it goes to select survivors and chose part of individuals as parents. And these parents breed a new generation. In the process of breeding, crossover and mutation may happen. And then, it goes into the next loop;
4. If the result satisfies constrains, it will come out of the loop to the end.

References: <https://en.wikipedia.org/wiki/Genetic_algorithm>

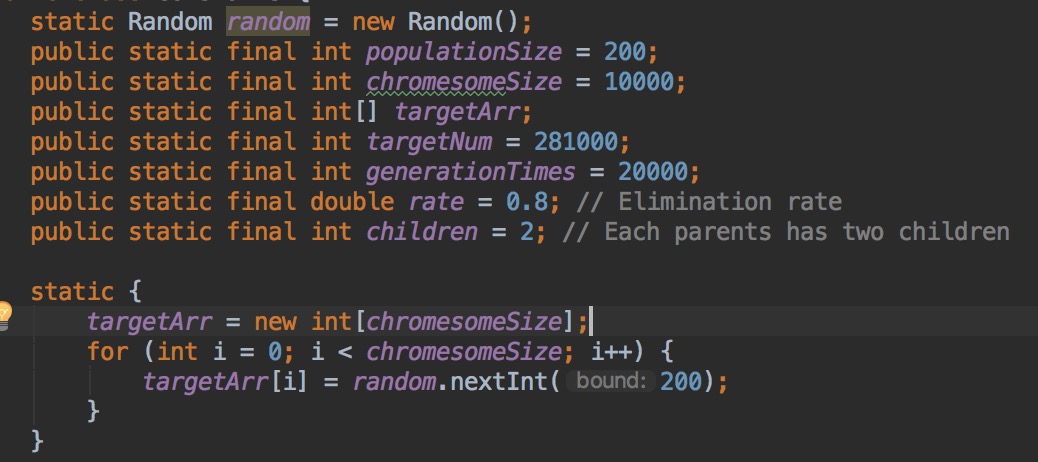
https://en.wikipedia.org/wiki/Subset\_sum\_problem

**Implementation**

In our application, there are mainly four classes, including Constants, Genetic, Individual and Population. We will instruct these classes in detail as follows.

1. **Constants class**

This class is used to store constant variables and it is convenient to change during testing.



*populationSize:* the number of individuals in the population

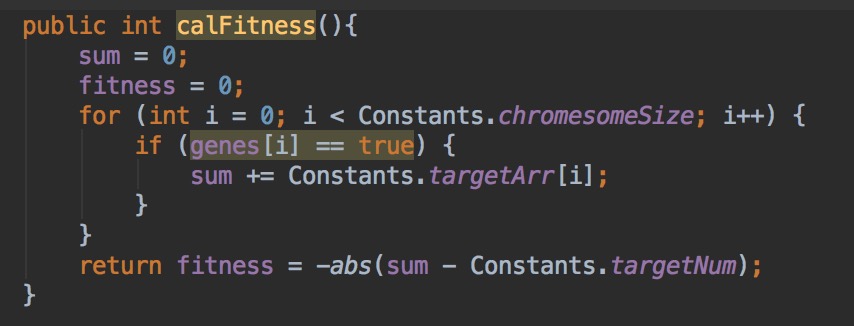
*chromesomeSize:* the number of bits

*rate:* preserve the rate of population to breed a new generation

*children:* the number of children each pair of parents has

1. **Individual class**

This class is used to create an instance of individual which has an array of genes (a Boolean array which has the same length with target array), and implements calFitness() function.



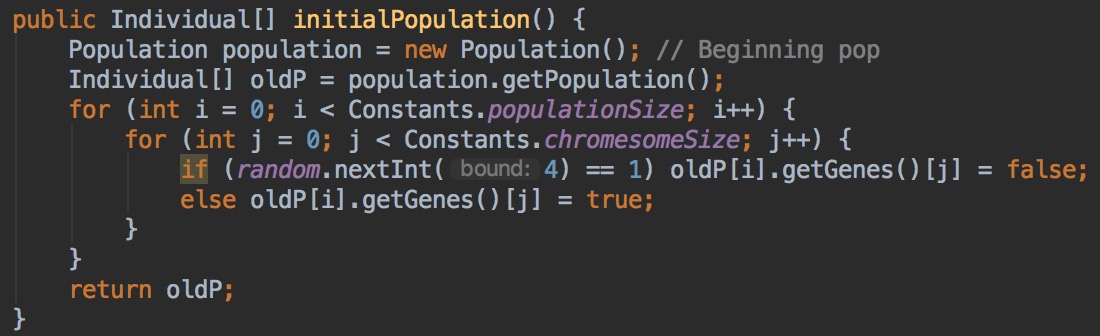
calFitness(): used to compute the fitness of individual. It calculates the difference between the *targetNum* and the *sum* of subset. The value of fitness is the difference. The smaller this value is, the better result is.

1. **Population class:**

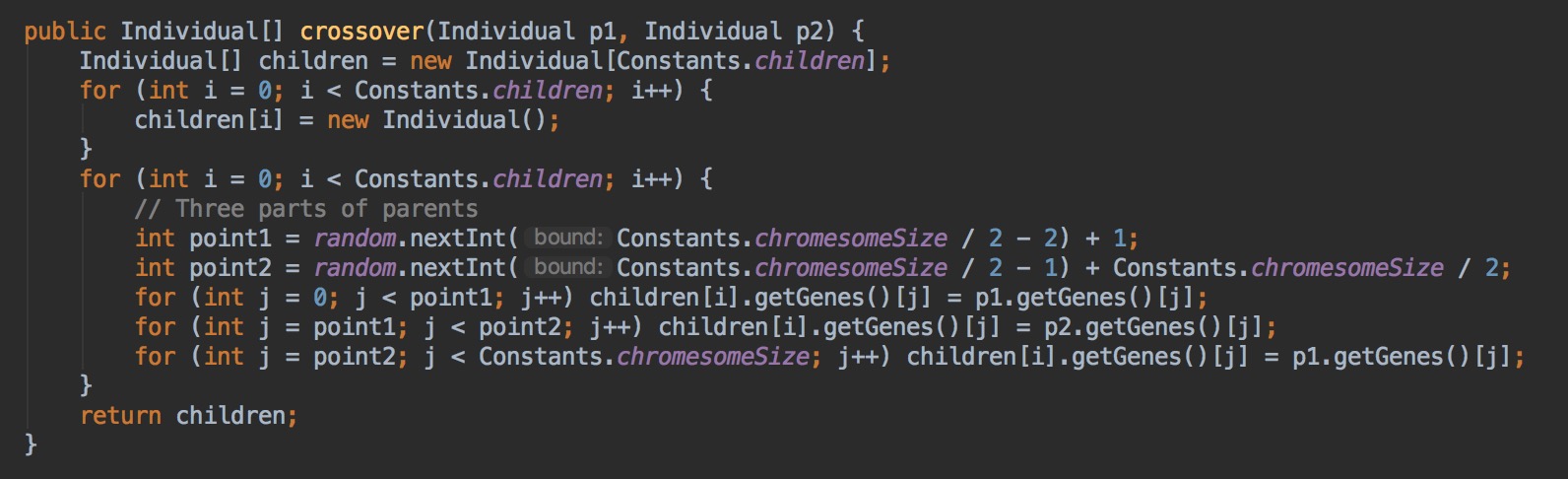
This class is used to create an instance of population which has an array of individuals.

1. **Genetic class:**

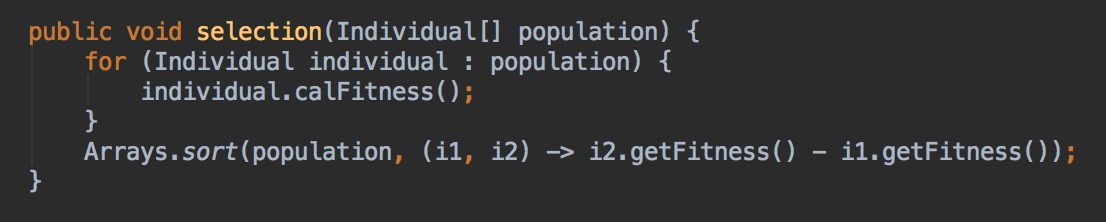
This class is used to implement initialPopulation(), crossover(), mutation(), selection() functions.



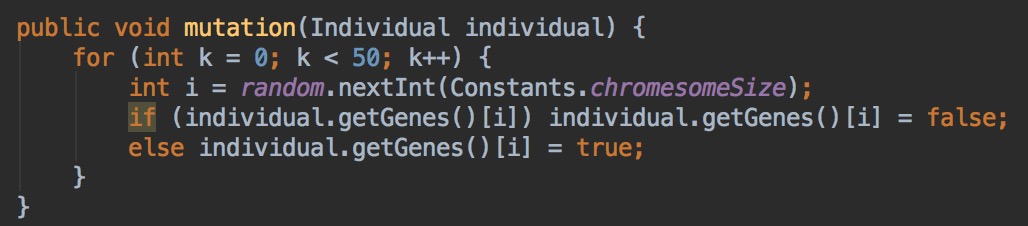
initialPopulation(): initiate population and individuals of generation one



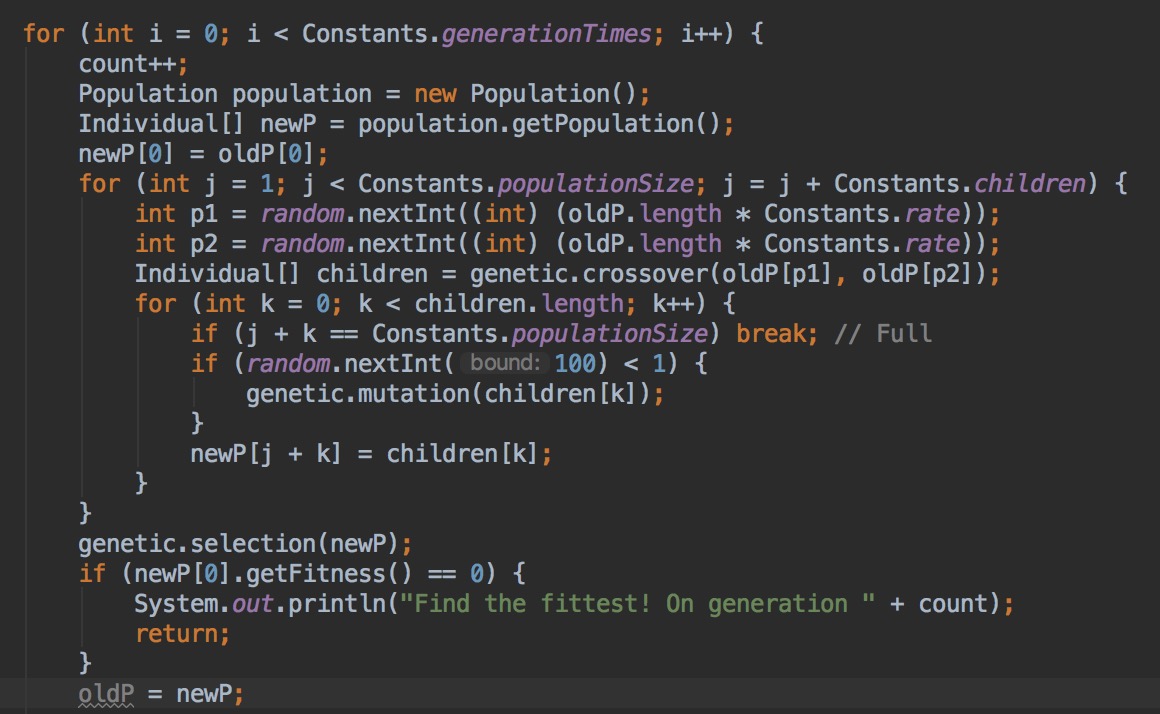
crossover(): is used to breed a new generation. Firstly, it randomly calculates two points. And then, these two points cut off genes of generation one into three partitions. The new generation receives the first and third partition of a parent’s genes, while gets the second partition from the other parent’s genes.



selection(): is used to evaluate the fitness of population. After calculating the values of fitness, we sort these values and chose part of the top values and related individuals.



mutation(): when mutation happens, 50 genes will change to the reverse state.

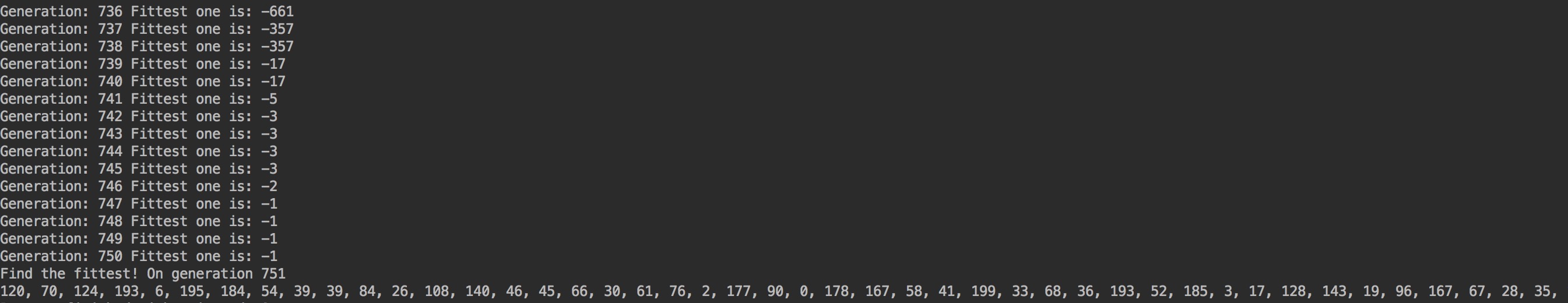


Main (part):

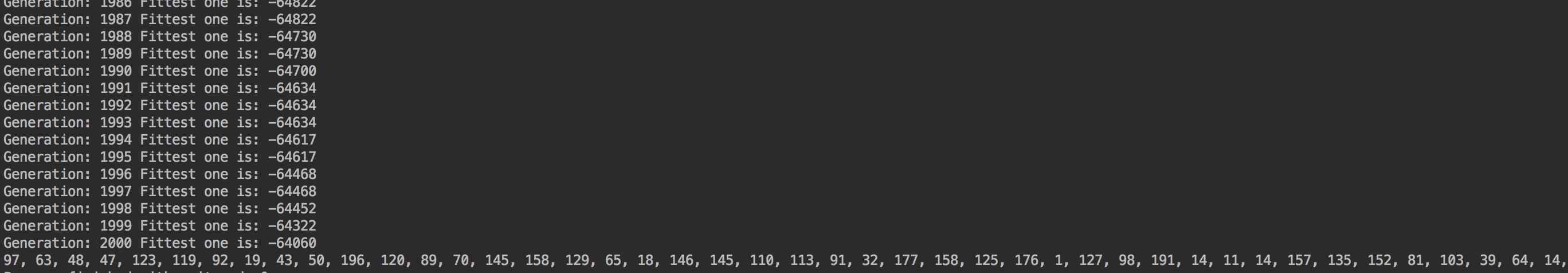
1. The evolution starts from a population of randomly generated individuals, marked as generation one;
2. Evaluate and sort the fitness of generation one, the best one will be preserved to the new generation directly;
3. Select survivors (‘*rate’* can control the number of survivors) and chose two individuals as parents. And these two parents breed some children (‘*children’* can control the number of children for each pair of parents), every new individual may has mutation (1%);
4. It goes into loops until the best individual whose fitness is closest to *targetNum* comes out .

**Result**

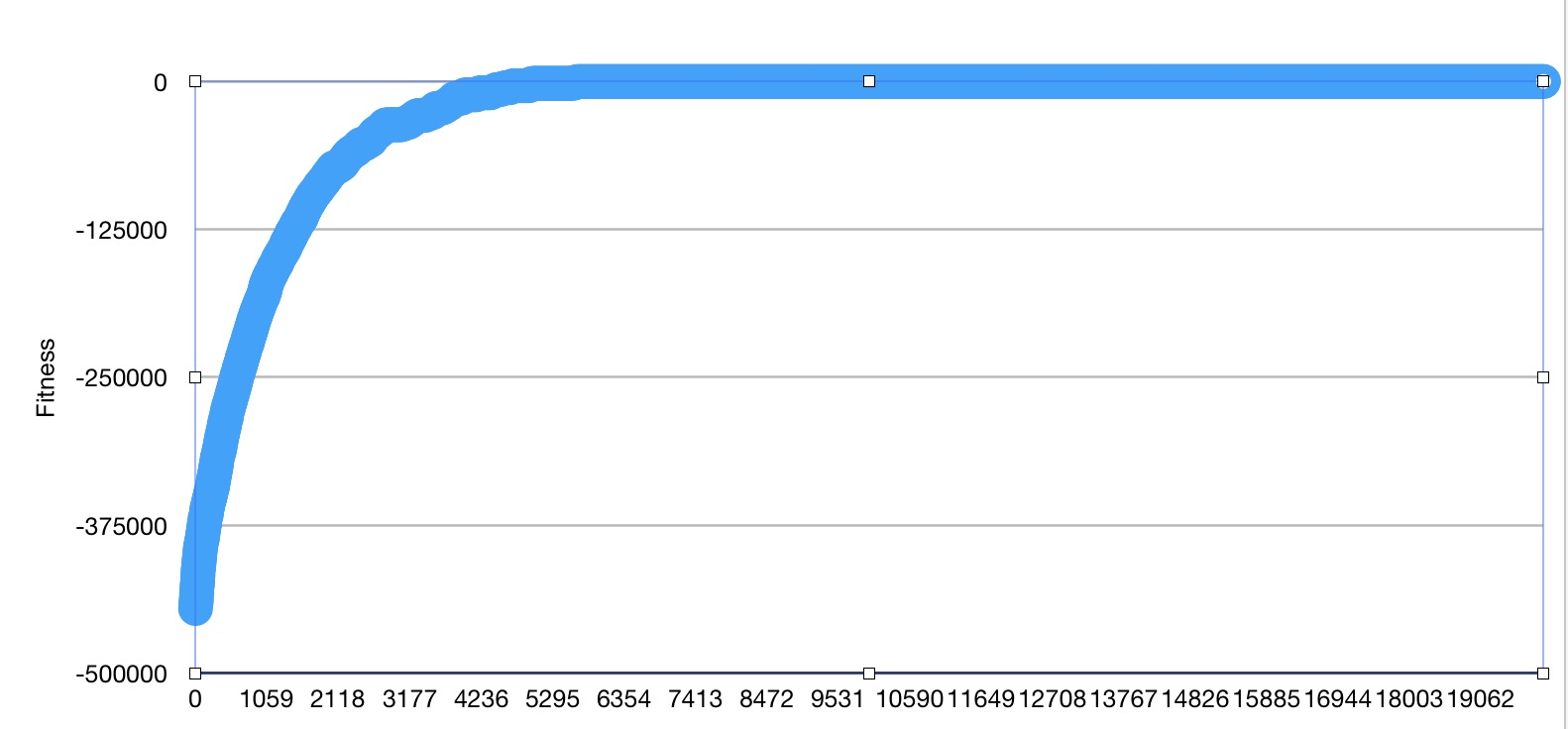
1. Find the best individual. Integers in the last line are the target numbers of gene.



1. Not find the best individual. Integers in the last line are the closest result to the target numbers of gene.



1. The results gradually converge to a stable value.



**Conclusion**

After the experiment, we can find that the Genetic Algorithm can generate a result which is quite close to the target number in the Subset Sum Problem. According to these graphs, we can see that the result gradually converges to a stable value. So we make a conclusion that Genetic Algorithm is a good way to solve Subset Sum Problem.

**UnitTest**

It is used to test whether methods are correct.

