Genetic Algorithm: Bin Packing Problem

Rajasekhar Reddy Duddugunta

Final Draft: April 15, 2018

Problem

We have created a Genetic Algorithm to solve the Bin Packing problem. The problem states: In the **bin packing problem**, objects of different volumes must be packed into a finite number of bins or containers each of volume V in a way that minimizes the number of bins used. In computational complexity theory, it is a combinatorial NP-hard problem. The decision problem (deciding if objects will fit into a specified number of bins) is NP-complete

Implementation

Genetic Code & Expression: Representation is designed with bins in mind, not individual items so each gene in a chromosome represents a single bin (group of items) not an individual item. This allows crossover operations to handle bins correctly and allows them to pass whole bins to offspring chromosomes instead of cutting them in the middle and disrupting good bins.

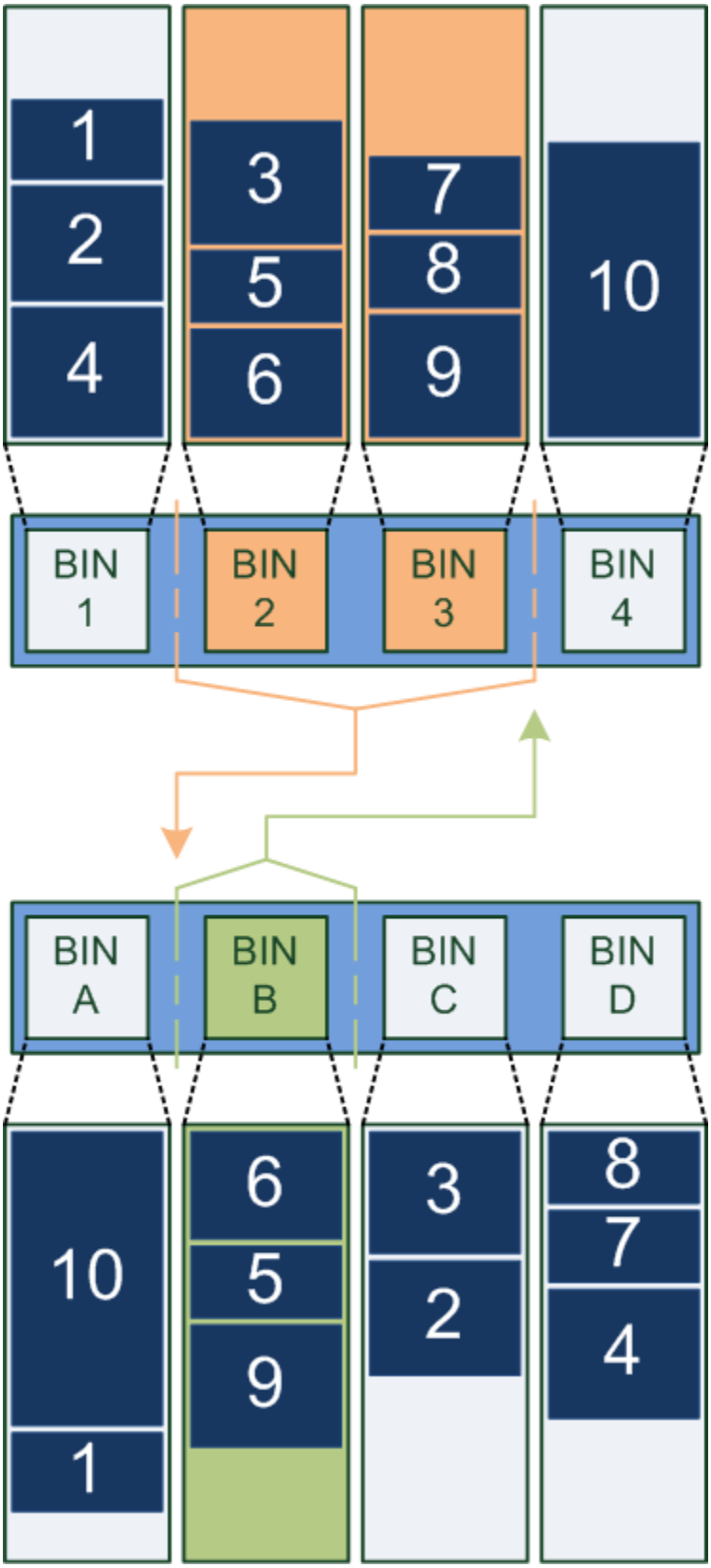
Fitness Function: The most obvious fitness function for this problem is the number of items used by the solution, but it does not create smooth search space that genetic algorithm can explore. To make search space smooth, function that takes the fill of bins in the solution into account is used and it looks like this:

$$F = \frac{\sum_{i=1}^{n} \left(\frac{f_i}{c}\right)^k}{n}$$

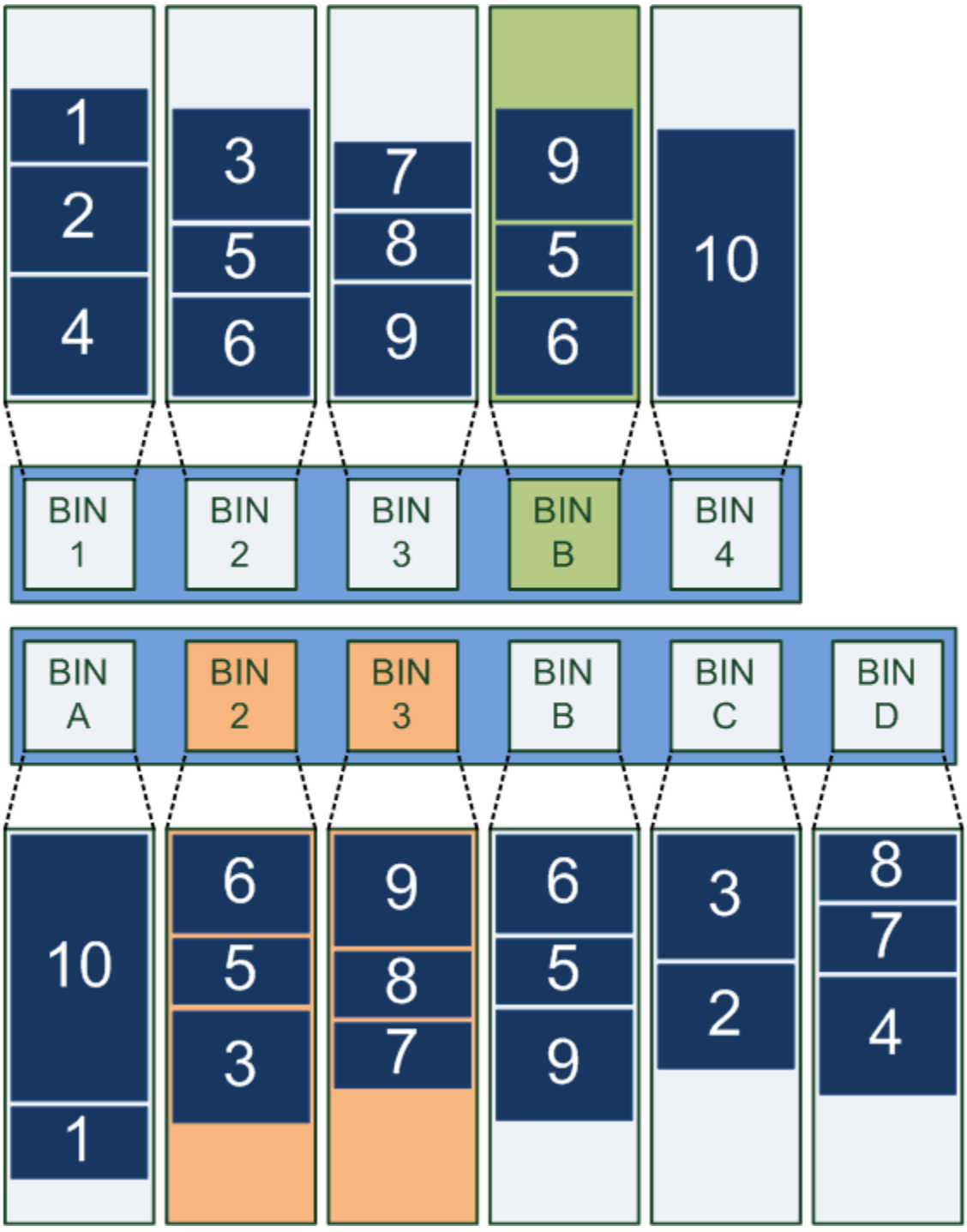
F - fitness of the solution, ***n*** - number of bins, f_i - fill of the ***i***th bin, ***c*** - capacity of the bin, ***k*** - constant greater then 1

k constant controls whether the more filled bins are preferred or equally filled bins. Larger values should be used in case more filled bins are preferred. This example sets value of ***k*** to 2.

Crossing Over: The first thing crossover operation do to produce offspring is creation selected parents' clones. These clones are going to be used as base of offspring chromosomes. Then the operation chooses two crossover points per clone.

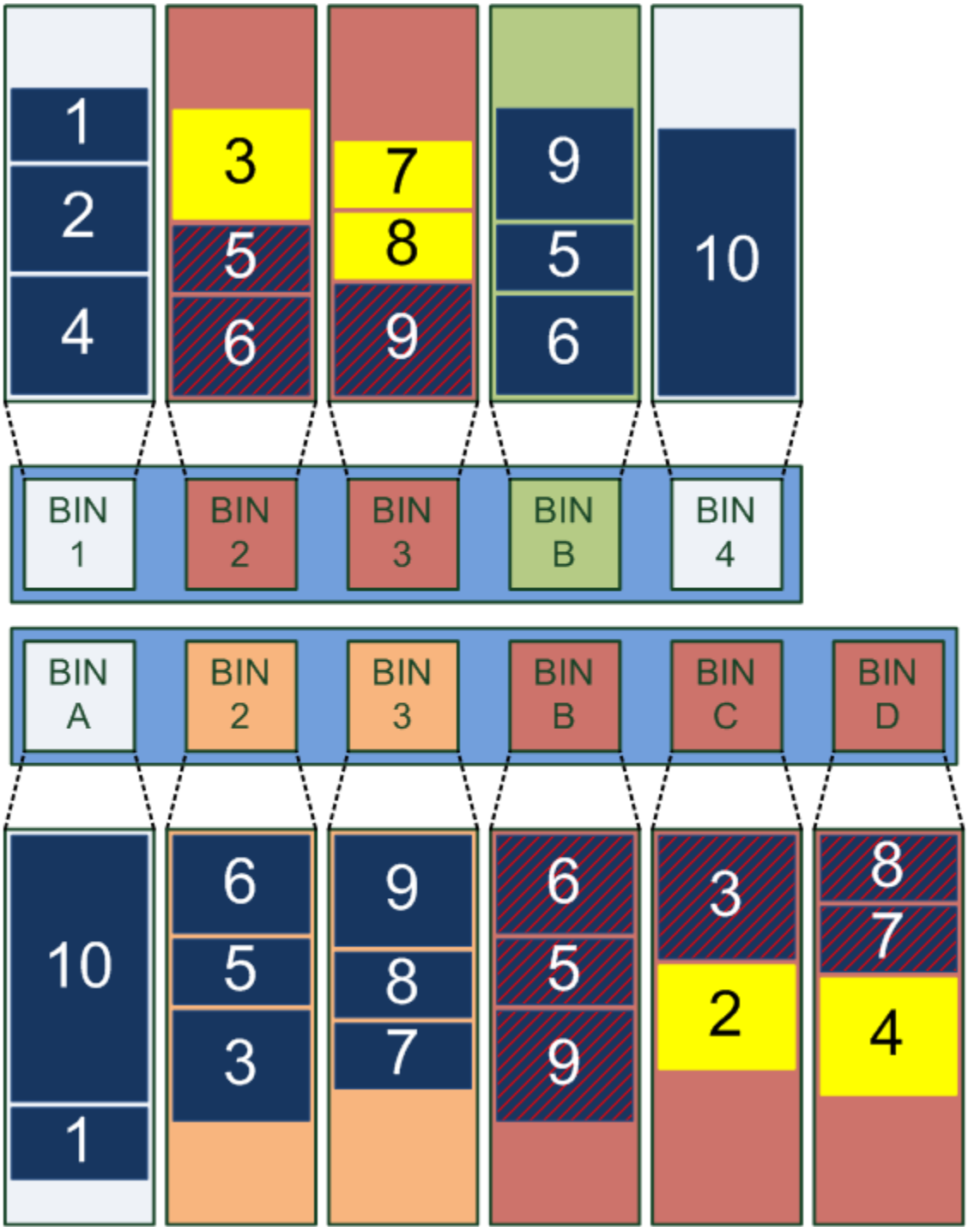

1
2
4
3
5
6
7
8
9
10
BIN 1
BIN 2
BIN 3
BIN 4
BIN A
BIN B
BIN C
BIN D
10
1
6
5
9
3
2
8
7
4

The next step is transferring bins between parents. Crossover copies and inserts bins between crossover points from the first parent into second parent's clone at the first crossover point. Then it swaps parents and copies bins from second parent into the first parent's clone at the second crossover point.

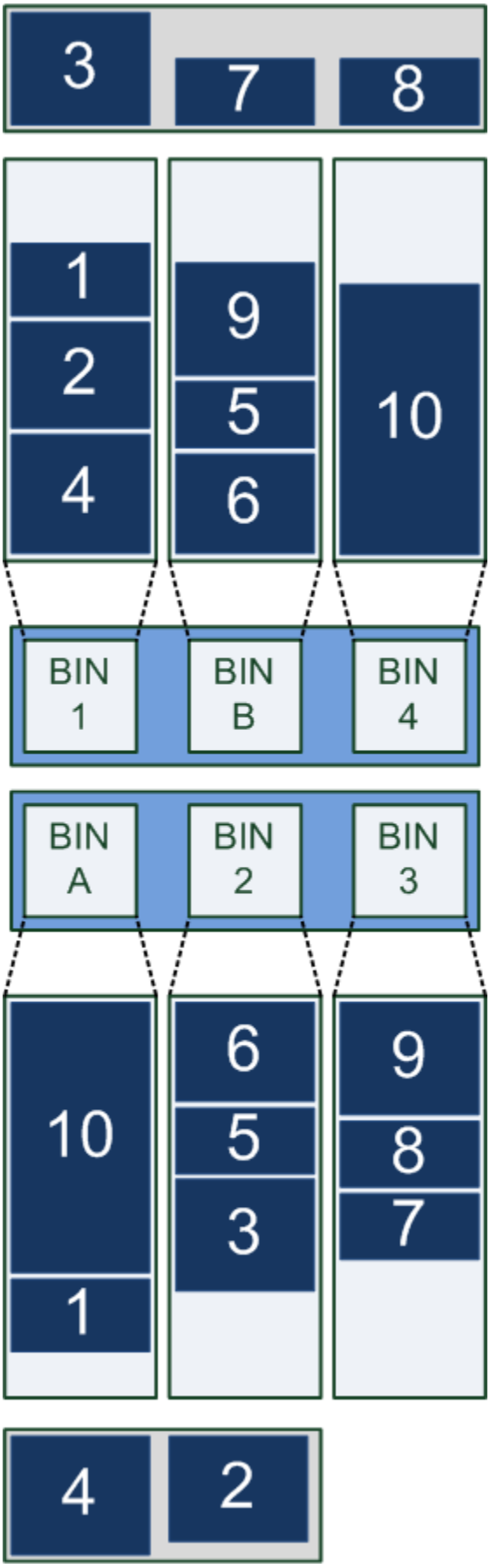


Copying bins like this will produce invalid solutions, as the image illustrates. Now there are items that appear twice in the solution. Because of this algorithm has to perform corrections to make valid

solution. To do so, it searches for finds all bins not copied from the other parent that contains duplicates and removes them. In this way gene transferred from the other parent are preserved, which is the point of crossover operation.

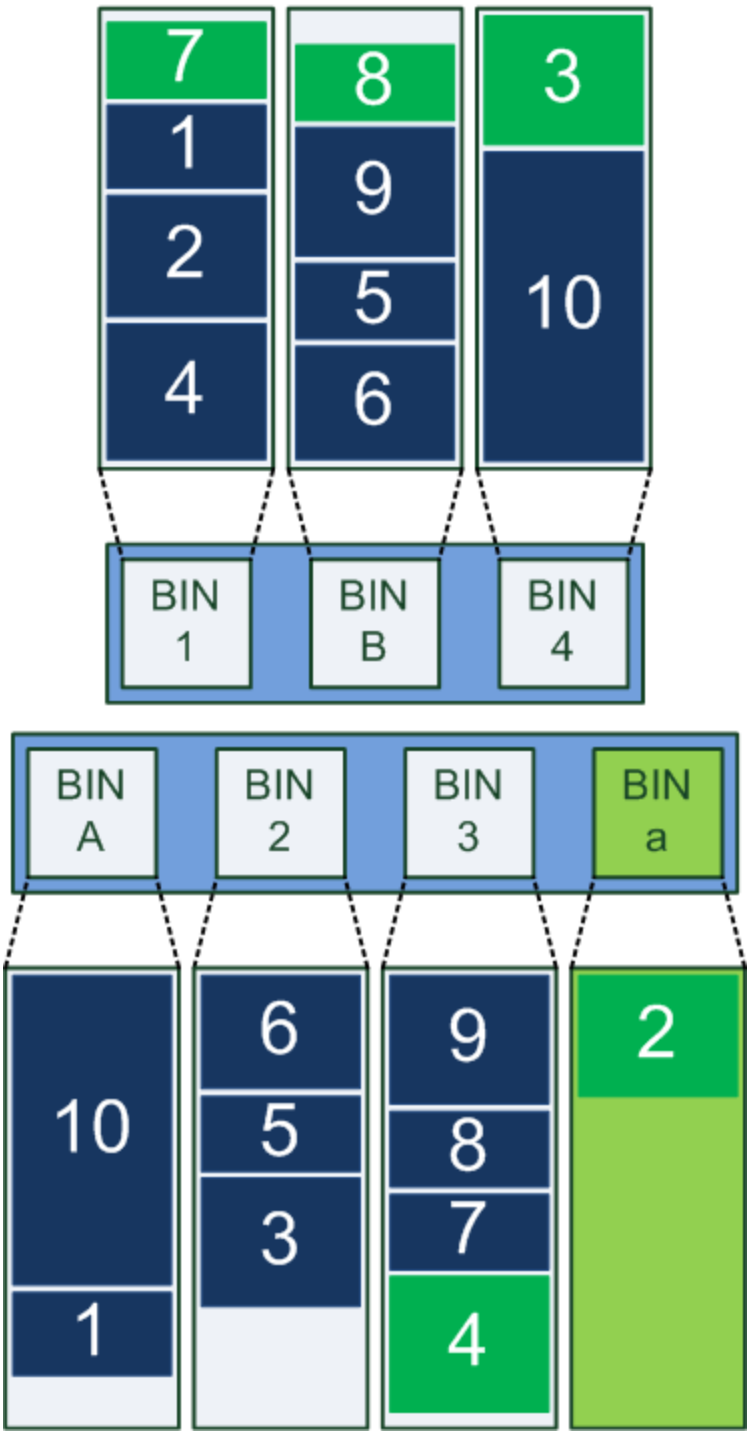


Removing these bins will also cause removal of some items that are not duplicates, because they were stored in the same bins with duplicates. Algorithm needs to create list of these items (unassigned items), and reinserts them back in to the solution.


3
7
8
1
2
4
9
5
6
10
BIN 1
BIN B
BIN 4
BIN A
BIN 2
BIN 3
10
1
6
5
3
9
8
7
4
2

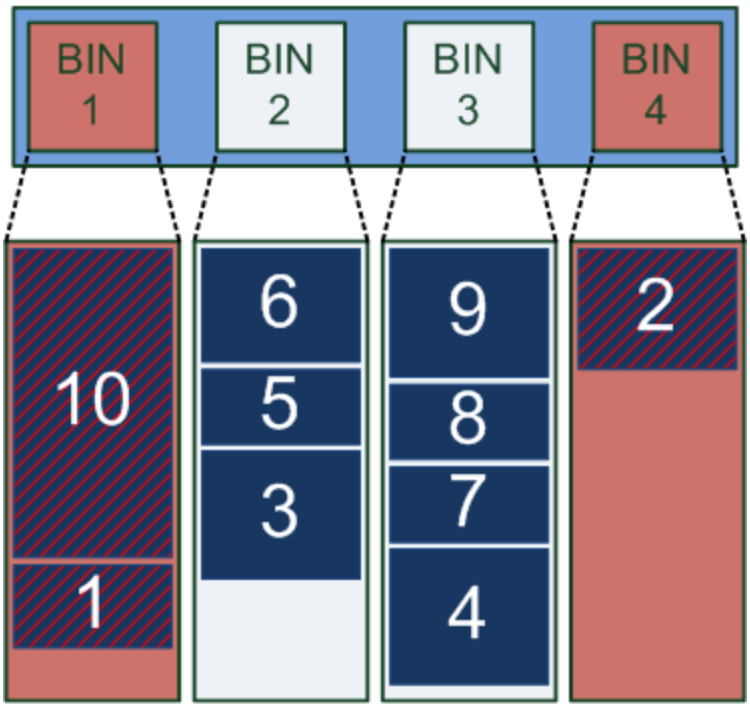
Inserting unassigned items back into solution is process that has two steps. The first step is replacement of items, which will be discussed later in section that explains mutation.

When replacement step is finished and there are still unassigned items, crossover operation will sort list of remaining items by their size in descending order and insert them into the first bin that has enough free space. If there is no bin which can accommodate item, new bin is created and the item is stored in it. This technique is called first-fit descending heuristic.



Mutation Operation:

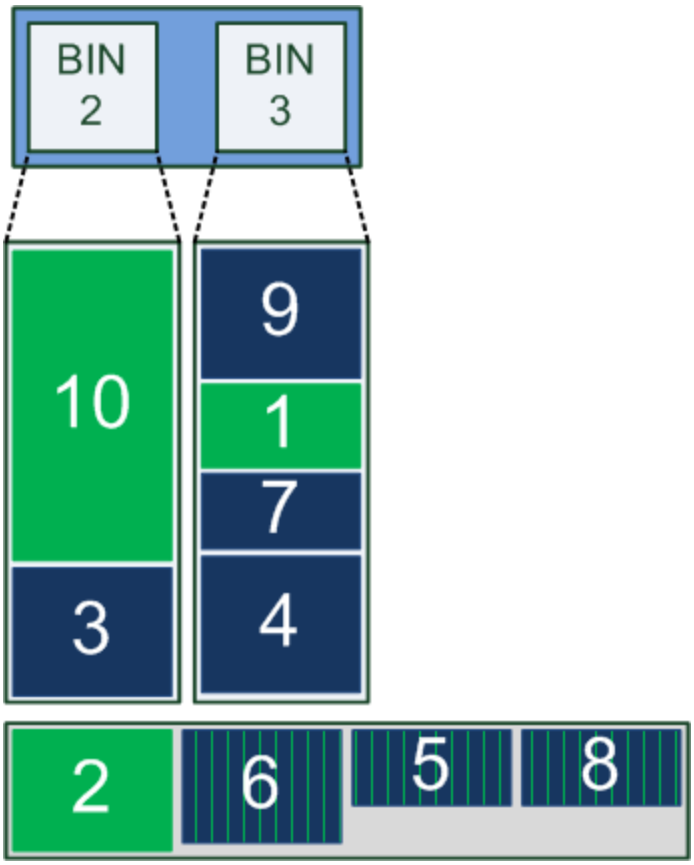
Mutation operation is simple: few bins are selected randomly and destroyed. Destruction of bins will allow those items that were in destroyed bins to be rearranged after reinsertion. This, hopefully, will lead to improvements of bin space usage.



The items that were in removed bins are preserved in the list of unassigned items.

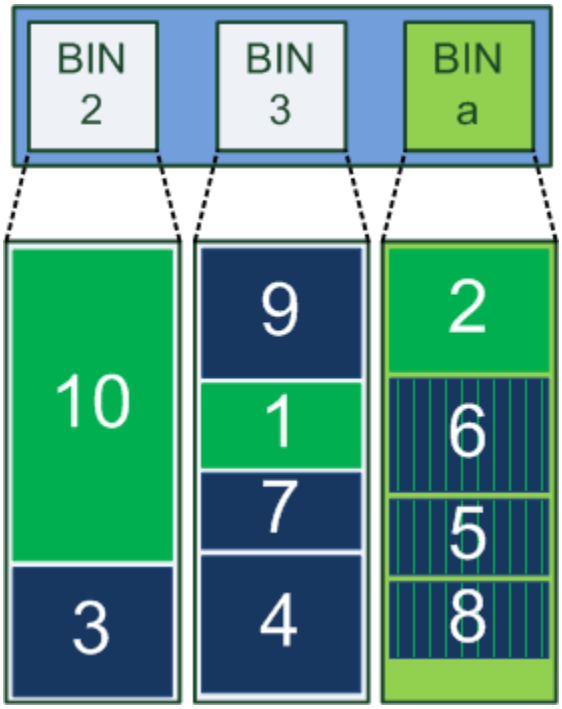
The next step of the mutation algorithm is to reinsert those unassigned items into bins in the same fashion as it is done in crossover operation.

This is good opportunity to explain replacement of items mentioned during the discussion about crossover operation. Replacement work in the following way: If there is currently unassigned item ***U*** and set of items ***P*** in a single bin ***B***, such that ***size(U) > size(P)*** and ***size(B) - size(P) + size(U) <= capacity(B)***, then items from ***P*** are removed from ***B*** and ***U*** is inserted instead of them. Number of items that algorithm search for set ***P*** is limited to three for performance reasons. The idea behind this technique is discussed in the original article.



Here, the item 10 replaced items 6 and 5 from bin 2, as the sum these two items is less than size of item 10 and it can fit the bin. Then item 1 replaced item 5 from bin 3 for the same reason.

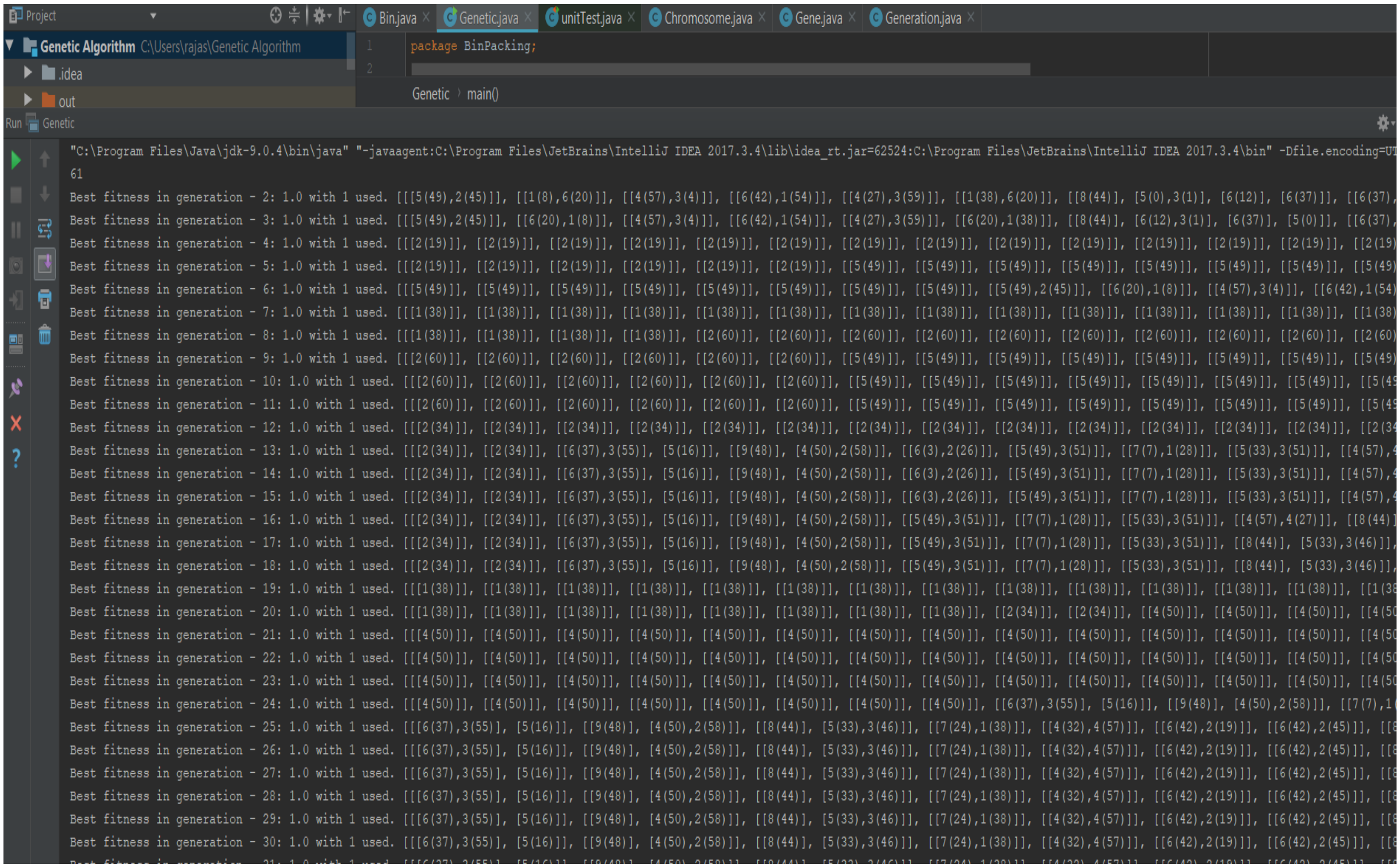
Just like it was the case with crossover operation, when algorithm cannot find any more items that satisfy criteria for replacement, it switches to first-fit descending heuristics to insert unassigned items into bins.



I this case, no items could fit into existing bin so new bin has to be created.

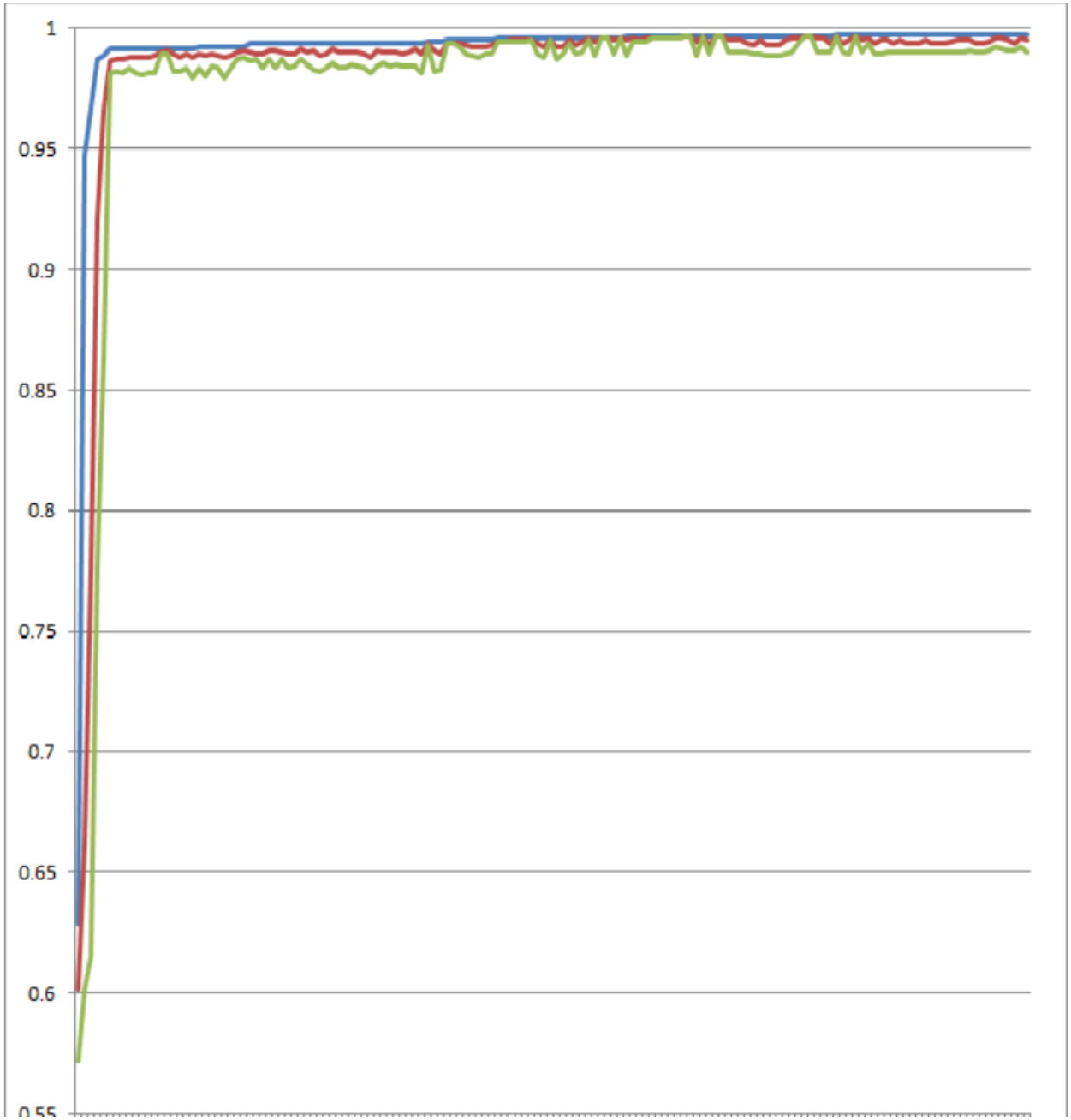
Images taken from net.

Results:



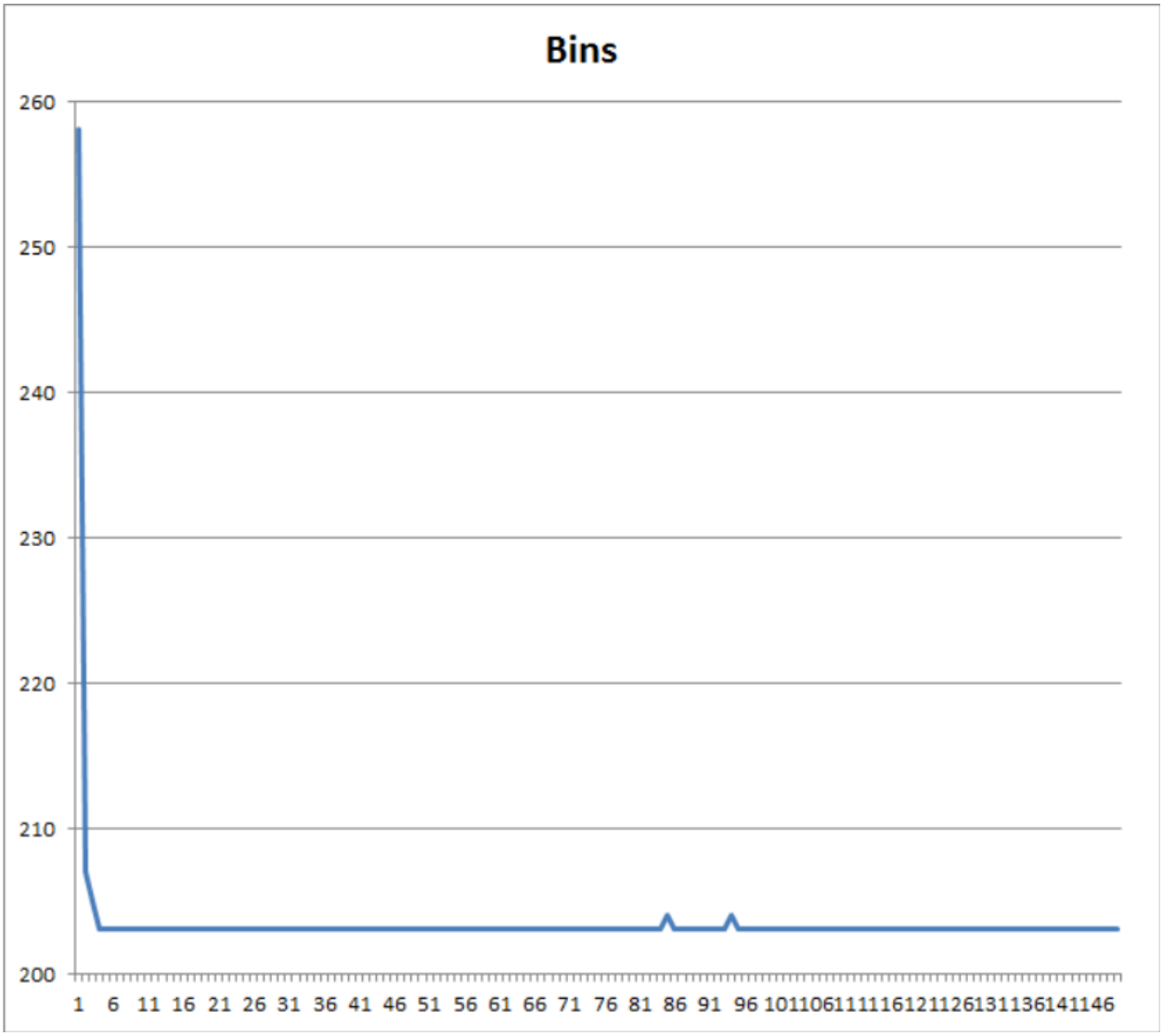
Graph 1:

Progress of fitness values during generations

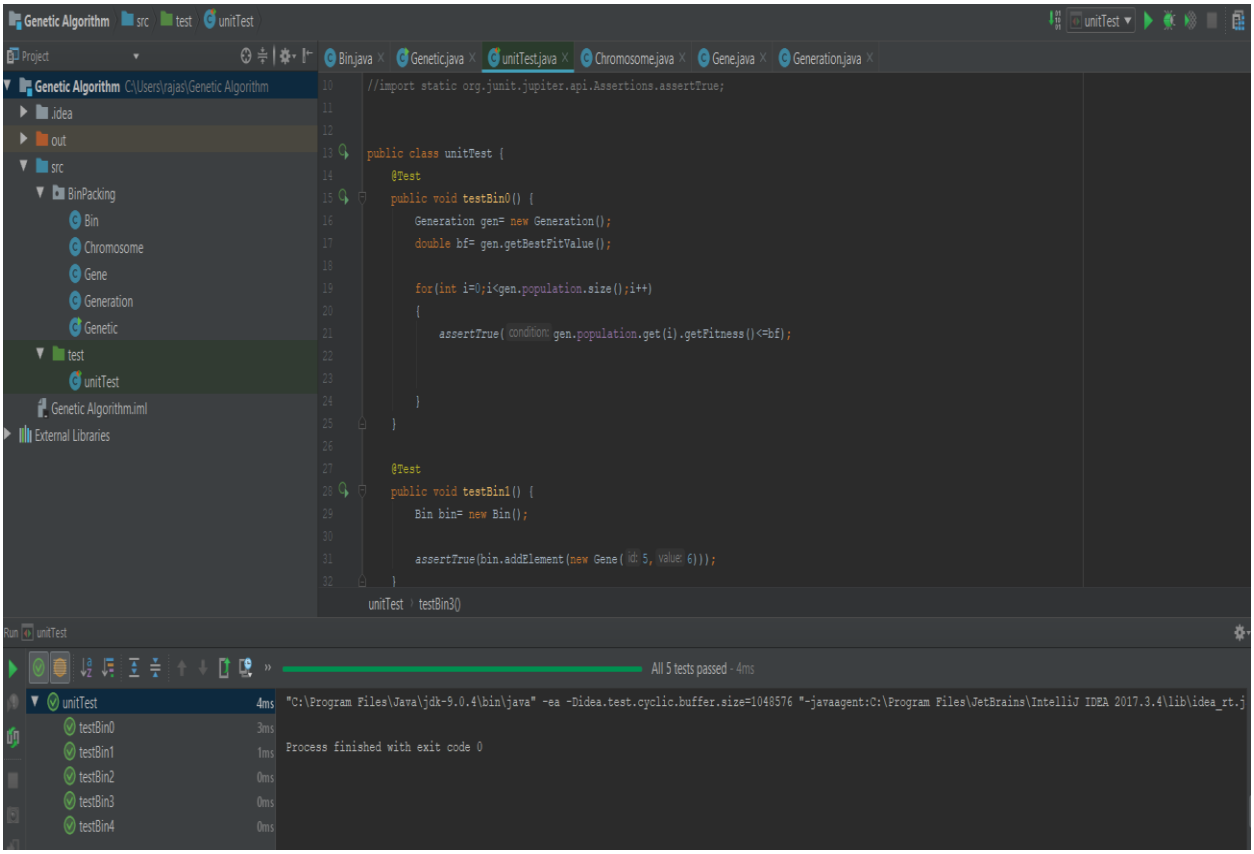


Graph 2 :

Number of bins used by the best solution in the generation



Test Case Output :


Genetic Algorithm
src
test
unitTest
unitTest
Project
Genetic Algorithm C:\Users\rajas\Genetic Algorithm
.idea
out
src
BinPacking
Bin
Chromosome
Gene
Generation
Genetic
test
unitTest
Genetic Algorithm.iml
External Libraries
Bin.java
Genetic.java
unitTest.java
Chromosome.java
Gene.java
Generation.java
//import static org.junit.jupiter.api.Assertions.assertTrue;
public class unitTest {
@Test
public void testBin0() {
Generation gen= new Generation();
double bf= gen.getBestFitValue();
for(int i=0;i<gen.population.size();i++)
{
assertTrue(condition: gen.population.get(i).getFitness()<=bf);
}
}
@Test
public void testBin1() {
Bin bin= new Bin();
assertTrue(bin.addElement(new Gene(id: 5, value: 6)));
}
unitTest
testBin3()
Run
unitTest
All 5 tests passed - 4ms
unitTest 4ms
testBin0 3ms
testBin1 1ms
testBin2 0ms
testBin3 0ms
testBin4 0ms
"C:\Program Files\Java\jdk-9.0.4\bin\java" -ea -Didea.test.cyclic.buffer.size=1048576 "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2017.3.4\lib\idea_rt.j
Process finished with exit code 0