Software Testing

Lab 1: Debugging

1. Introduction

The purpose of this lab is to learn how to debug using Eclipse and/or IntelliJ built-in debugger tools and how to combine that with several debugging heuristics. As there are many ways to approach problems, this lab aims to give an overview of some of them in combination with debugger tools.

a) Some useful vocabulary:

- **Debugging** is the process of localising the source of a failure in the code and determining the cause behind it (with the purpose of then fixing it). This can mean simply looking at a chunk of code, running the code and checking the console log or using a debugger.
- **Using a debugger** and debugging are not synonymous. Debugger is a tool that can be used to debug a failure.
- A **breakpoint** indicates a line of code or program at which you want the execution of an application to pause, a **watchpoint** indicates a data item whose change in value causes the execution of your application to pause. (2) Sometimes these are used synonymously. In this lab we focus on breakpoints according to the given definition.
- **Heuristic** comes from the Greek word "Εὑρίσκω" which means "find" or "discover". The term refers to techniques for problem solving and solution discovering (1). Debugging heuristics are techniques to use when it is impractical to go through the code line-by-line and the purpose is to speed up the debugging process. This can simply be common sense or the idea that reducing input can make it easier to understand what a chunk of code does.

b) Eclipse and IntelliJ buttons and meanings:

- **Step Into** – Step into whatever function call is on the line you are at. If there is no function call there, program will continue to the next line.
- **Step Over** – Step over the line you are in and onto the next one.

- **Drop to Frame (Eclipse) – Drop Frame (IntelliJ).** In Eclipse you go back to the start of the function you are in. In IntelliJ you go back to the call of the function you are in.
- **Step Return** – Step to the returned value of the function you are in (or the line after this function was called).
- **Resume** – continue running the program as normal until the next breakpoint is found.
- **Variables tab** – this is your most useful tool, there you can see all the current values in the function you're in – integers, strings, arrays etc. In both IntelliJ and Eclipse, you can see an array better if you go deeper in the variables tab. In both of them open the little arrow of the array in the variables tab, in Eclipse click on the array (such as "heapList"), then you can see the array at the bottom of the variables tab. In IntelliJ you need to right-click on the list that opened, choose "View as" and "toString".

Note: In IntelliJ you can see the variables in the editor on the lines you have already passed. In Eclipse you see them only in the variables tab.

c) Overview of the systems to be tested:

System 1: Max binary heap

- Binary heap is a data structure where all elements are in a tree form with up to 2 child elements. In a maximum binary heap the root element is the largest and the child elements have to always be smaller than their parent. There is no order to the left and right child elements as long as this condition is met (i.e. the left child does not have to be smaller than the right child).
- The purpose of this program is to sort a list of positive integers into a heap and allow adding and removing of elements from that list while still preserving the heap data structure. This program is meant to operate only with positive integers, failures due to negative input values should not be considered faults.
- Links to learn more can be found in Chapter 4.

System 2: Genetic algorithm for the 8 queens' problem

- Genetic algorithms solve a problem by generating an amount of random specimens (a population) of the data (in this case, states of a chessboard with 8 queens on it), evaluating the specimens, and mating the fittest specimens with each other. It simulates the survival of the fittest with the addition of randomness by adding a mutation similar to biology.
- The 8 queens' problem is the task of putting 8 queen chess pieces on a chessboard so that none of them threaten any of the others. In chess, queens can move any amount of squares in any direction (left-right, up-down, diagonals).
- The program provided for you should generate new generations of individuals (specimens) until it finds a desired solution. There are 92 possible correct states of the chessboard that the program could reach. Reaching any one of them is considered a correct solution.
- Read more about the given genetic algorithm program flow in Appendix A.
- There are links to learn more about genetic algorithms and this 8 queens' problem in Chapter 4.

d) Debugging heuristics:

- As explained before, heuristics are techniques used to simplify and quicken a problem solving process. Table 1 shows a list of heuristics that we have provided for you. This is not a conclusive list of debugging heuristics, just some that will help you when debugging.

- We strongly recommend using at least some of these in addition to using the debugger as it can make your work process a lot easier and faster. You can also try to figure out or find from search engines some more heuristics to use. If you do use some that are not listed here, explain them briefly in your lab report in the heuristics column (see Table 2).

Table 1: List of debugging heuristics

No.	Remark	Why is it useful
1	Whenever you reach a new function call, step into it.	If you have reached a function call and don't know the source of the failure yet, it is likely that the problem is deeper in the code. This means you need to check whether the body for this function works as it should.
2	Whenever you reach a new function call that takes a parameter, check if the parameter given is logical.	The cause for a failure may just be that all the functions work properly, just that one of them is called with incorrect parameter value(s).
3	Check the documentation of the code for hints.	The comments in the code from the author can be very useful in determining what the code is supposed to do and how.
4	Change the input code from main method	Reducing input can make it easier to follow the run of the code or even cause an error that makes it easier to localise the problem. Changing it into something that you know the correct output for can also make it easier to make sure the code does what it's supposed to.
5	If the data is in a structure that is difficult to visualize, construct it on a paper while debugging	If the data is a tree, a matrix or some other structure that is difficult to keep visualizing all the time, it can be useful to draw out (with pen and paper) the state you have in your debugger while debugging (using the variables view). That makes it easier to be sure that you did not miss anything from the variables and states you were viewing.
6	Change the value of a variable in the debugger. ("Change value" command)	This can give you a more varied overview of the ways the code acts with different values without having to restart the debugger from the start with a new original call parameter.
7	Run only part of the program at a time, if possible.	If you run the full program you will have a lot of code to debug through, which does not save you much time when compared to exhaustive solving. Therefore find a way to only run some of the program and check that before continuing on with the rest.
8	If you already know something is going to work correctly in a loop, don't step through every iteration again.	You can skip for loops to a specific run of the cycle. This allows you to skip the runs of the loop that you already know will be correct and immediately get the run where you think a problem might be. To do this, set a breakpoint on the for loop line (You can do this even when the

		debugger is already running). In IntelliJ, right click on the breakpoint, write the iteration number in the Condition field (e.g. i==3), and then run the debugger. In Eclipse, right click on the breakpoint, choose Breakpoint properties, tick Conditional and write the condition in the field below (e.g. i==3).

e) Thought process for localizing bugs:

Figure 1 shows step-by-step what you should think about while debugging and when documenting your bug resolution tables (see Table 2 below, example will be done with the lab instructor in lab) for each bug. Use this while debugging to document your steps.

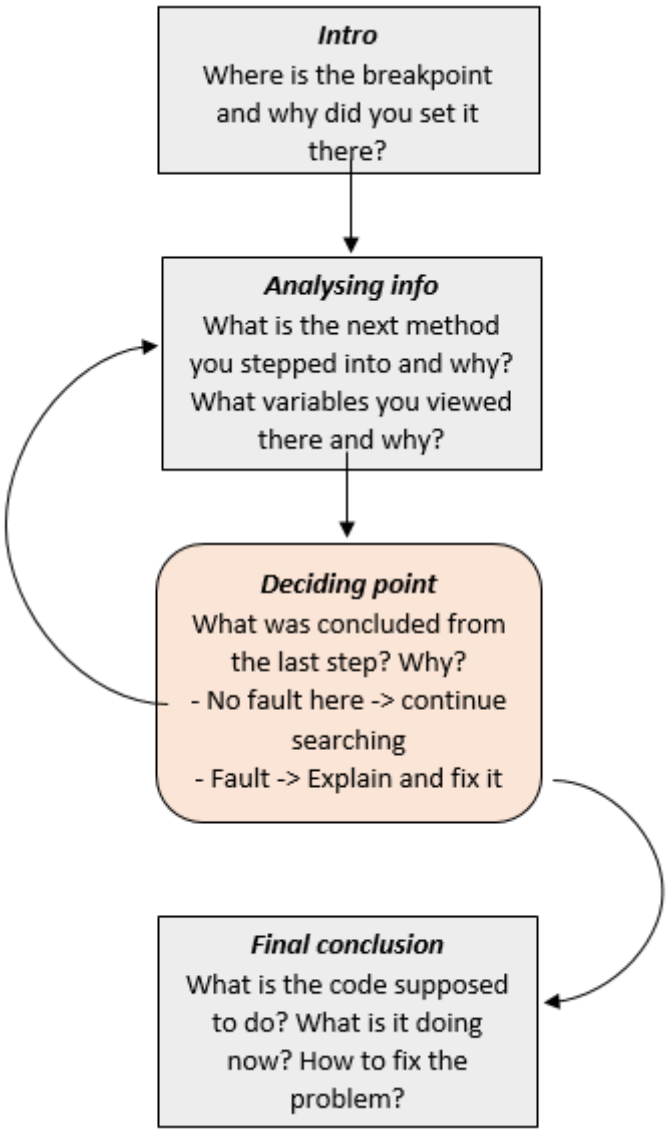


Figure 1: Thought process for localizing a bug

You always start by setting at least one breakpoint in the code and then start reviewing the variables in the program and comparing the state in the debugger to what you think the program should do.

f) Bug resolutions in your Lab Report:

Table 2 shows how bug resolutions shall be documented. This reporting table structure will be used for both of the tasks. Here are descriptions of the columns and what they should contain:

- Method: The method column should state which method are you looking at, for the first row you could state which method you set the breakpoint in.
- Variables: The variables column should state what the variables are, that you are currently reviewing in the Variables tab, with their values. If you just set a breakpoint, state the line of code where you put it (and variables if applicable).
- *Intro* and *Analysing info*: This should correspond with *Intro* and *Analysing info* sections from Figure 1. Why are you reviewing these variables? Why is it necessary to consider this information? For the first row, why did you set the breakpoint there?
- *Deciding point* and *Final conclusion*: This column should correspond with the *Deciding point* and *Final conclusion* sections from Figure 1. State what you learned from these variables or by setting this breakpoint? (e.g. "these variables were calculated correctly" or "since I know this program should do this, I can see from these variables that it does something else")
- Heuristics used: The heuristics column should state the heuristic(s) you used while debugging. You might use heuristics proposed in Table 1 (**in that case, you can simply write "Heuristic 1", "Heuristic 2", etc.**) or you can use heuristics that you found yourself (then you must briefly describe them). It is not necessary to have a heuristic for each row of each table but at least some heuristics should have been useful for you in solving each bug. If it seems difficult to attach a heuristic to a specific row in the table (but you used them nonetheless), you can simply put all heuristics into the heuristics column of the first row.

Table 2: Bug resolution table

Method	Variables	*Intro* and *Analysing info*	*Deciding point* and *Final conclusion*	Heuristics used
…				

2. Homework Tasks and Reporting

Your homework consists of 2 tasks – debugging a maximum binary heap algorithm and debugging a genetic algorithm. There are two Java projects provided for you (HeapsortStudent and GeneticAlgorithmStudent). You should import them as projects to your chosen editor (IntelliJ or Eclipse). Your lab report should include a list of faulty code lines with their corrections and bug resolution report tables (see Table 2) grouped by issues for each task.

That means you list the faulty line and the solved line for each issue (under the relevant task) and then provide the bug resolution process tables for the same bugs.

Task 1 – Debugging "Max binary heap"

Read the entire task before you start solving it!

1. The faulty code is in the Heap.java file, there is a test class provided to run this program in TestHeap.java file of the same project. As a hint we can already tell you, that there are 6 bugs in this program. The resolution process for the first bug will be done with the lab instructor during the lab session. You have to state the faulty line of code, the fixed line of code and to document the bug resolution process for the remaining 5 bugs in your lab report to get full marks for this task.
2. There are 3 issue reports for this program. This does not mean that there are exactly 3 bugs. Several bugs can result in 1 issue. The given input code is already provided for you in the test class of the program.
   The issue reports for this program are in Appendix B of this file.
   Your job is to debug these issues using the Eclipse or IntelliJ built-in debugger and the heuristics given in Table 1 or additional heuristics you found or thought of (explain those in the table).
   Keep in mind that **each issue can correspond to one or several bugs in the code!**
   If you use different input and that results in a new failure you can mention it in the lab report but you only get points for solving the issues given to you.
3. Document you debugging process. In the lab report you need to have 2 parts for each issue.
   - Firstly, state the issue and then make a list of all the bugs (faulty lines of code) you found for that issue with the corrected lines of code.
   - Secondly, for **each bug**, document your resolution process using the format of Table 2.

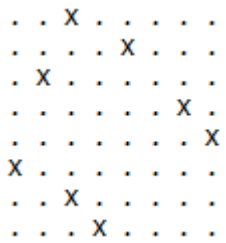
   Important: Each bug resolution (and your solving process) needs to correspond to **1 separate resolution table** and that table needs to include the answers to the questions in the solving process in Figure 1. Only provide the main parts of the process (not every single "Step Over" click etc.) of the debug sessions that led you to finding the bugs.

Task 2 – Debugging "Genetic algorithm"

Read the entire task before you start solving it!

1. The faulty code is in the Algorithm.java file. It contains 4 bugs. The test class is provided for you with relevant input code (TestGA.java).
2. The expected output for this algorithm is
   - the printout of each generation's best fitness
   - the correct solution as list
   - the correct solution as an 8x8 table where 'X' marks a queen on a chess board.
3. There are 2 issue reports for this program. This does not mean that there are exactly 2 bugs. Several bugs can result in 1 issue. The given input code is already provided for you in the test class of the program.
   The issue reports for this program are in Appendix C!
   - **NB1! Issue 2 occurs only if you have fixed the first issue!**
4. Document your issue debugging process. The lab report and thought process structure is the same as in Task 1. (Check Task 1, point 3)
5. Hints
   a. In the TestGA.java file of this program, there are methods to run each algorithm method separately without running the full program. It is very useful to use these to debug each algorithm method separately and to make sure they work as expected.
   b. Check Appendix A for more information about the program flow.

c. The chessboard state is saved as a list of 8 integers with each element being a row and its' value being a column in that row. This means that [2,4,1,6,7,0,2,3] describes the following chess board state:



3. Grading Scheme

- You can get a maximum of 10 points for this lab.
- You will get 1 point for attending the lab.
- You will get 1 point for each full bug resolution (up to 9p). A complete bug resolution includes:
  - the faulty code line
  - the correction of the faulty code line
  - the resolution process table with your work process and reasoning.

  If any of these three parts is missing, incorrect or incomplete, you will lose points.

4. Links and Appendixes

Useful links

Heap sort:

- http://www.geeksforgeeks.org/heap-sort/
- https://en.wikipedia.org/wiki/Binary_heap

Genetic Algorithm:

- https://www.analyticsvidhya.com/blog/2017/07/introduction-to-genetic-algorithm/
- https://en.wikipedia.org/wiki/Eight_queens_puzzle

Debugger tools:

- IntelliJ https://www.jetbrains.com/help/idea/debugger-basics.html
- Eclipse https://www.eclipse.org/community/eclipse_newsletter/2017/june/article1.php

References

1. Testing Heuristics & Mnemonics by Karen N. Johnson 2012 Available at: http://karennicolejohnson.com/wp-content/uploads/2012/11/KNJohnson-2012-heuristics-mnemonics.pdf
2. Debugging Features and Techniques in Eclipse, Using Breakpoints and Watchpoints Available at: http://documentation.microfocus.com/help/index.jsp?topic=%2Fcom.microfocus.eclipse.infocenter.visualcobol.eclipseux%2FGUID-C2FB4B87-7C87-47C8-9712-A3E1F209F480.html

Appendix A – Program flow for “Genetic Algorithm”

Here is a flow chart of the genetic algorithm program when it works completely correctly. There is also a more detailed description of the flow. In the detailed description there are some questions provided which you should ask when making sure that the program does what was intended by the developer. These questions are labelled “Useful questions”. This is not a definitive list of questions and others could be asked when analysing the program.

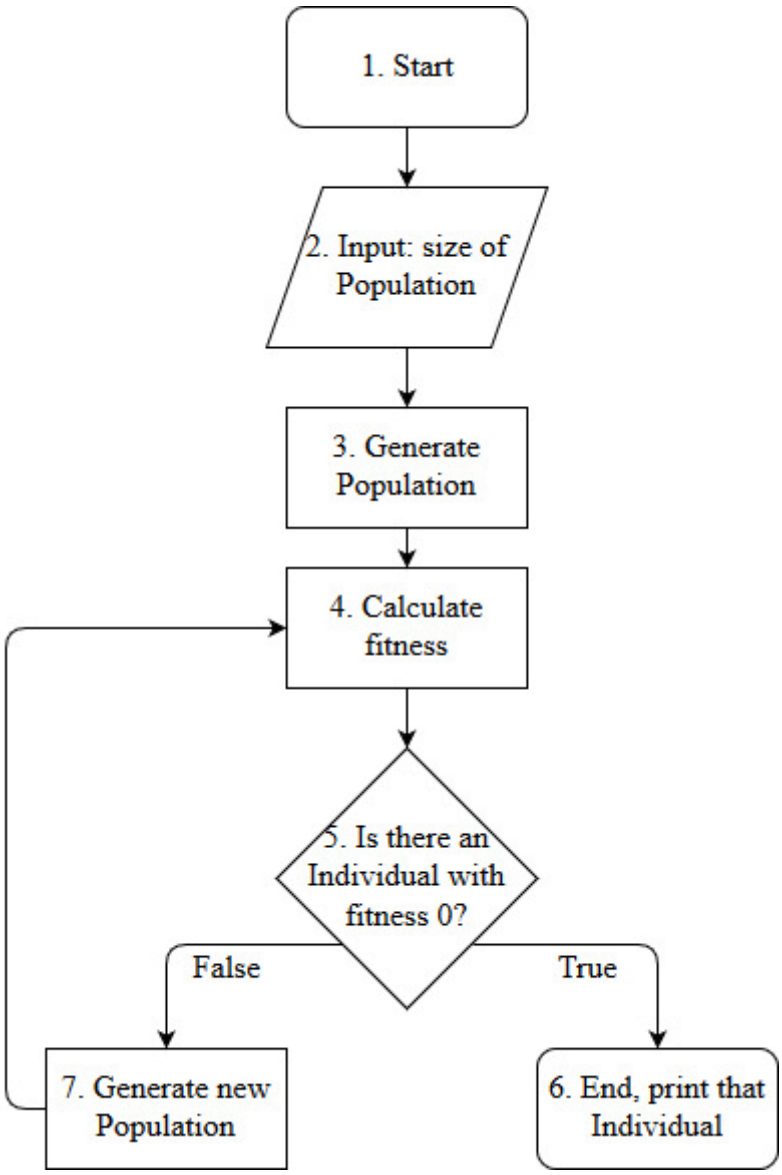


Figure 2. Program flow for “Genetic Algorithm”

1. The program gets the amount of Individuals as input (Figure 2, point 2) and generates a Population (Figure 2, point 3). By default, the input is 100.
   1.1. One Individual is one chess board state as a list with length of 8. Each of these 8 elements represents a row on the chess board. The values of these 8 elements represent the corresponding column in the viewed row with the values ranging 0…7. See Figure 3 for a visual explanation of this.
   1.2. The program generates the given amount of Individuals and stores them as a Population list (i.e. list of lists).

Here is an image of a Population with size 4 to describe how the data is stored.

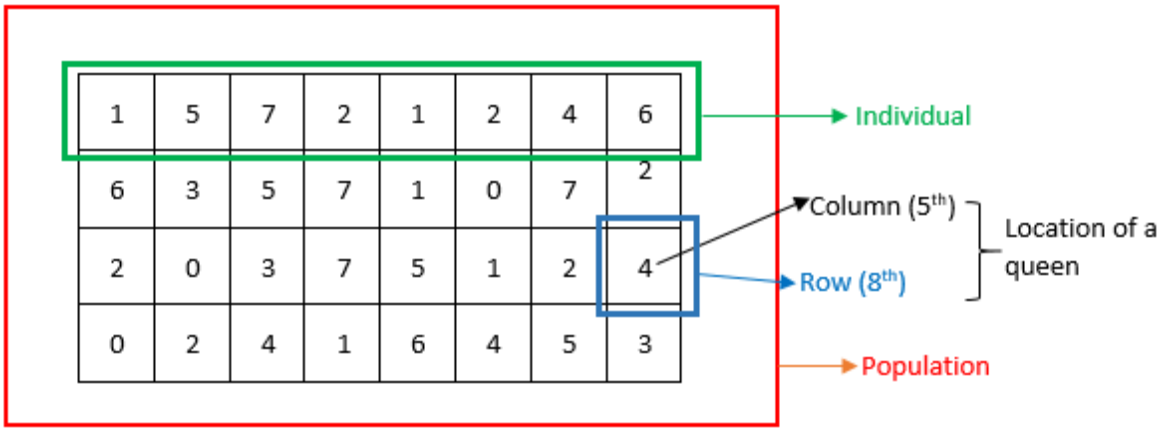


Figure 3. Data structure for the "Genetic Algorithm"

Useful questions! Does the size of the list of Individuals match the given Population size (100 by default)? Is the Individual in the described shape (1.1)? Are the Individuals' values in the declared range?

2. For **each** Individual, the program does the following:
   2.1. Calculates fitness (Figure 2, point 4)
      2.1.1. Calculates how many clashes exist in the board state columns. +1 to the fitness value per each queen per each clash.

      Useful questions! Does the program check all Individuals? Does the program check all columns? Does the program add +1, whenever it sees a clash? Does the program store the amount of clashes correctly? When there are no clashes, does the program use a 0?

      2.1.2. Calculates how many clashes exist in the board state diagonals. +1 to the fitness value per each queen with a clash. This means that the fitness value is increased by 1 for one queen with a clash in its diagonals, even if the same queen has several clashes in the diagonals. NB! Different from columns! For each queen: +1 PER each clash in columns; +1 IF there's at least one clash in any diagonal.

      Useful questions! Does the program check all Individuals? Does the program check all diagonals? Does the program stop looking, when it finds a clash? Does the program add +1 when it finds a diagonal clash for the viewed queen? Does the end fitness value match your count of clashes?

   2.2. Checks if at least one Individual matches the end condition. (Figure 2, point 5)
      2.2.1. The end condition is a fitness value of 0 for an Individual

      Useful questions! Does the program check all Individuals until it finds one that matches the condition?

      2.2.2. If an Individual matches the end condition – stop (Figure 2, point 6)

      Useful questions! Does the program stop at the right moment? Does the program check anything after reaching the end condition?

      2.2.3. If not – generate a new Population (Figure 2, point 7)

      Useful questions! Does the program continue only if fitness is >0?

3. If end condition was not met, the program generates a new Population based on the previous one and checks its fitness. (Figure 2, point 7)
   3.1. The program sorts all Individuals in the current Population by their fitness in ascending order.
   3.2. The program takes the first half of the sorted Population list.

Useful questions! Does the program actually take the Individuals with better fitness? Does the program take exactly half of the list? Does the program round down when the list size is uneven?

3.3. The program takes Individuals two at a time (i and i+1); for the last Individual, it is combined with the first one. Based on these pairs of Individuals, the program creates 2 new Individuals through randomized crossover (i.e. mating) as seen in Figure 4.

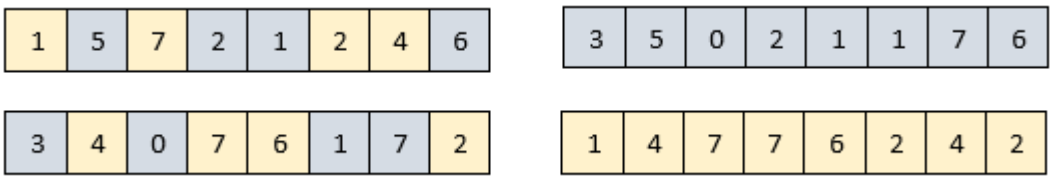


Figure 4. Structure for the mating (i.e. crossover) part of the "Genetic Algorithm"

Useful questions! Does the program take the values on the correct position in the Individual? Is the Individual list size the same? Is the Population size the same? Are the 4 values distinct? Are all values used? Do the values of the new Individuals differ from the old ones? Does the program pair the Individuals correctly (i and i+1)?

3.4. Program takes each new Individual from the created list and mutates one random value.

Useful questions! Is the entire list covered? Is the randomness fair? Do the Individual and Population sizes remain the same? Is the new value in the correct range? Is only one value changed?

3.5. Program calls the generation and fitness calculation recursively on the new Population until it reaches the end condition.

Appendix B – Issue Reports for "Max Binary Heap"

- **Issue report 1:**

  Description:

  The program should heapify any given list of positive integers but the resulting tree (and list) does not meet the max binary heap structure.

  Input:

  heapifying a list of integers - [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0]

  Input code:

```
List<Integer> heapList = new ArrayList<Integer>();
heapList.add(1);
heapList.add(2);
heapList.add(5);
heapList.add(7);
heapList.add(6);
heapList.add(8);
heapList.add(11);
heapList.add(10);
heapList.add(3);
heapList.add(4);
heapList.add(9);
heapList.add(1);
heapList.add(0);
System.out.println("List before heapifying:");
```

```
System.out.println(heapList);
Heap heap = new Heap(heapList);
System.out.println("After heapifying: ");
heap.printAsList();
heap.printAsTree();
```

Expected output:	Actual output:
List before heapifying: [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0] After heapifying: [11, 10, 8, 7, 9, 1, 5, 2, 3, 4, 6, 1, 0] 5 8 0 1 1 11 6 9 4 10 3 7 2	List before heapifying: [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0] After heapifying: [11, 9, 5, 3, 9, 7, 1, 6, 3, 4, 2, 1, 0] 1 5 0 7 1 11 2 9 4 9 3 3 6

- **Issue report 2:**

  Description:

  When adding elements to a heap, the element should automatically go to the correct location in the heap. However, after adding elements, the resulting list (tree) does not match the expected max bin heap structure, there are different elements than what should be there (some more, some missing).

  Input:

  heapifying a list of integers - [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0], then add integers 13 and 4

  Input code:

```
List<Integer> heapList = new ArrayList<Integer>();
heapList.add(1);
heapList.add(2);
heapList.add(5);
heapList.add(7);
heapList.add(6);
heapList.add(8);
heapList.add(11);
heapList.add(10);
heapList.add(3);
heapList.add(4);
heapList.add(9);
heapList.add(1);
heapList.add(0);
Heap heap = new Heap(heapList);
heap.addElem(13);
```

```
heap.addElem(4);
System.out.println("After adding elements: ");
heap.printAsList();
heap.printAsTree();
```

Expected output:	Actual output:
After adding elements: [13, 10, 11, 7, 9, 1, 8, 2, 3, 4, 6, 1, 0, 5, 4] 4 8 5 11 0 1 1 13 6 9 4 10 3 7 2	After adding elements: [11, 13, 5, 13, 9, 7, 13, 6, 3, 4, 2, 1, 0, 13, 4] 4 13 13 5 0 7 1 11 2 9 4 13 3 13 6

- **Issue report 3**

  Description:

  When removing a given element (the first occurrence of that element) the element should be gone, the rest of the heap in a correctly sorted order, and the list itself one element shorter. The element is gone, but the heap is not in the correct order and the size of the list is not smaller.

  Input:

  heapifying a list of integers - [1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0], add elements 13 and 4, then remove element 10 and try to remove non-existing element 99.

  Input code:

```
List<Integer> heapList = new ArrayList<Integer>();
heapList.add(1);
heapList.add(2);
heapList.add(5);
heapList.add(7);
heapList.add(6);
heapList.add(8);
heapList.add(11);
heapList.add(10);
heapList.add(3);
heapList.add(4);
heapList.add(9);
heapList.add(1);
heapList.add(0);
Heap heap = new Heap(heapList);
heap.addElem(13);
heap.addElem(4);
```

```
heap.removeElem(10);
heap.removeElem(99);
System.out.println("After removing elements: ");
heap.printAsList();
heap.printAsTree();
```

Expected output:	Actual output:
After removing elements: [13, 9, 11, 7, 6, 1, 8, 2, 3, 4, 4, 1, 0, 5] 8 5 11 0 1 1 13 4 6 4 9 3 7 2	After removing elements: [11, 13, 5, 13, 9, 7, 13, 6, 3, 4, 2, 1, 0, 13, 4] 4 13 13 5 0 7 1 11 2 9 4 13 3 13 6

Appendix C – Issue Reports for "Genetic Algorithm"

- **Issue report 1 (Hint: corresponds to 3 bugs)**

  Description:
  When running the program, it should return a list of generations and the correct solution that it found. Instead, it throws an exception after 1000 generations.
  Input:
  Running the code with the population size of 100.
  Input code:

  ```
  Population pop = new Population(100);
  runAlgorithm(pop);
  ```

  Expected output:
  Generation: 1 Current highest fitness: <?>
  Generation: 2 Current highest fitness: <?>
  ...
  Found suitable board state on generation <?>: <[x1,x2,x3,x4,x5,x6,x7,x8]>
  Here is the found solution as a board where . marks an empty spot and, X marks a queen
  <printout of 1 correct solution of 92 possible>

  Actual output:
  Generation: 1 Current highest fitness: 22
  Generation: 2 Current highest fitness: 22
  Generation: 3 Current highest fitness: 26
  ...
  Generation: 1000 Current highest fitness: 38

Exception in thread "main" java.lang.Exception: Didn't find solution in 1000 generations
at Algorithm.generation(Algorithm.java:119)
at Algorithm.generation(Algorithm.java:129)

Comments:

This issue might not be reproducible line-to-line due to randomness in the algorithm, meaning the current highest fitness can vary. But the core of the issue is reproducible (exception).

Hints:

1. There are 3 bugs that correspond to this issue
2. You can consider this issue fixed only when all 3 bugs have been fixed. In order to be sure that you have fixed the correct bugs, run the program multiple times. The expected output may sometimes be seen because of the randomness of the data, make sure that the correct output appears every time you run the program.
3. While you have not yet fixed all the bugs, depending on the order in which you find and fix them, you might see any of the following output:
    - The described exception is thrown.
    - The program outputs a board state that it claims to be correct. However, the board state is not correct as all the queens are positioned in one single diagonal on the board.
    - The program outputs a board state that it claims to be correct. However, the board state is not correct as there is at least one clash visible on the board.

- **Issue report 2 (Hint: corresponds to 1 bug)** (This issue appears after Issue 1 has been fixed)

Description:

Based on past projects using genetic algorithms, the average amount of generations should be less than 87 and the program should produce the correct output in less than 100 generations on at least 75% of the runs. However, the performance is much worse, the average amount of generations is over 100 and it only gets the solution in under 100 generations in less than 62% of the time. On very few runs, the program throws an exception of not finding a solution in under 1000 generations.

Input:

Calculated average amount of generations and percentage of runs where solution was found in under 100 generations with population size 100 and 1000 runs.

Input code:

```
public static List<Integer> generationCounts = new ArrayList<>();
public static void main(String[] args) throws Exception {
        for (int i = 0; i<1000; i++) {
                pop = new Population(100);
                generation(pop);
                generationCounts.add(counter+1);
                counter = 0;}
        System.out.println(calculateAverage(generationCounts));
        System.out.println(calculatePercent(generationCounts));
        generationCounts.removeAll(generationCounts); }
```

Expected output:

Average generation count < 87
P(generation count<=100) >75%

Actual output generalized	and specific:
Average generation count > 87	Average generation count: 109.916
P(generation count<=100) < 75%	P(generation count <100): 57.7%

Comments and hints:

As performance can be affected by many things, the issue reporter has provided their own insight as a hint. You may use this, but don't have to.

a) Genetic algorithms and their performance are strongly based on evaluations of states and fitness calculations.
b) It is important to check that the code does what the developer has intended it to do. To know what is intended, use Appendix A and helpful methods in the program (main class, the run<method_name>() methods)