LTAT.05.006: Software Testing

Lecture 01:
Introduction to Software Testing

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Spring 2021
Structure of Lecture 1

• Introduction and Motivation
• Course Information
• Basic Vocabulary
• Lab 1
2012: Knight Capital loses 440M USD

- August 12th: New Trading Software installed
- Administrator forgets to deploy on one out of eight server nodes
- New code repurposed a flag previously used for testing scenarios
- On that one server node, old trading algorithm interprets flag differently and starts buying and selling 100 different stocks randomly without human verification
- NYSE has to suspend trade of several stocks
- Knight Capital loses 440 Mio USD in only 30 minutes, until system is suspended
- Investors have to raise 400 Mio USD in order to rescue the company
Uber’s car fatal car crash in Arizona, 2018

“The fatal crash that killed pedestrian Elaine Herzberg in Tempe, Arizona, in March occurred because of a software bug in Uber's self-driving car technology (...). According to two anonymous sources (...), Uber's sensors did, in fact, detect Herzberg as she crossed the street with her bicycle. Unfortunately, the software classified her as a "false positive" and decided it didn't need to stop for her.

Distinguishing between real objects and illusory ones is one of the most basic challenges of developing self-driving car software. Software needs to detect objects like cars, pedestrians, and large rocks in its path and stop or swerve to avoid them. However, there may be other objects—like a plastic bag in the road or a trash can on the sidewalk—that a car can safely ignore. Sensor anomalies may also cause software to detect apparent objects where no objects actually exist.

Software designers face a basic tradeoff here. If the software is programmed to be too cautious, the ride will be slow and jerky, as the car constantly slows down for objects that pose no threat to the car or aren't there at all. Tuning the software in the opposite direction will produce a smooth ride most of the time—but at the risk that the software will occasionally ignore a real object.”

List of SW Problems from Guru99 /1

• In April 2015, Bloomberg terminal in London crashed due to software glitch affected more than 300,000 traders on financial markets. It forced the government to postpone a 3bn pound debt sale.

• Nissan cars have to recall over 1 million cars from the market due to software failure in the airbag sensory detectors. There has been reported two accident due to this software failure.

• Starbucks was forced to close about 60 percent of stores in the U.S and Canada due to software failure in its POS system. At one point store served coffee for free as they were unable to process the transaction.
List of SW Problems from Guru99 /2

• Some of the Amazon’s third party retailers saw their product price is reduced to 1p due to a software glitch. They were left with heavy losses.

• Vulnerability in Window 10. This bug enables users to escape from security sandboxes through a flaw in the win32k system.

• In 2015, fighter plane F-35 fell victim to a software bug, making it unable to detect targets correctly.

• In may of 1996, a software bug caused the bank accounts of 823 customers of a major U.S. bank to be credited with 920 million US dollars.
List of SW Problems from Guru99 /3

- In April of 1999, a software bug caused the failure of a $1.2 billion military satellite launch, the costliest accident in history
- China Airlines Airbus A300 crashed due to a software bug on April 26, 1994, killing 264 persons
- In 1985, Canada's Therac-25 radiation therapy machine malfunctioned due to software bug and delivered lethal radiation doses to patients, leaving 3 people dead and critically injuring 3 others.
What is Software Testing?
What is Software Testing? (Static & Dynamic)

Confirm quality (pass-test) vs. Find defects (fail-test)
Cost of Testing / Cost of not Testing

2014 industrial survey of 1543 executives from 25 countries:
• Testing and quality assurance of software-intensive systems accounts for roughly 26% of IT budgets [1]

2013 study by researchers at the University of Cambridge:
• Global cost of locating and removing bugs from software has risen to $312 billion annually and it makes up half of the development time of the average project [2].

Sources:
Recall Exercise: A Pen

- Quality?
- Testing?
Software Quality – Definition

• **Software quality is the degree of conformance to explicit or implicit requirements and expectations**

Explanation:
• *Explicit*: clearly defined and documented
• *Implicit*: not clearly defined and documented but indirectly suggested
• *Requirements*: business/product/software requirements
• *Expectations*: mainly end-user expectations
Software Product Quality Model
– ISO 25010 Standard

Safety?
Software Product Quality Model
– ISO 25010 Standard
Software Quality Assurance (SQA)

versus

Software Quality Control (SQC)
Software Quality Assurance (SQA)

- **SQA is a set of activities for ensuring quality in software engineering processes (that ultimately result in quality in software products).**

It includes the following activities:
- Process definition
- Process implementation
- Auditing
- Training

Processes could be:
- Software Development Methodology
- Project Management
- Configuration Management
- Requirements Development/Management
- Estimation
- Software Design
- Testing
- ...
Software Quality Control (SQC)

- **SQC is a set of activities for ensuring quality in software products.**

It includes the following activities:

- **Reviews**
  - Requirement Review
  - Design Review
  - Code Review
  - Deployment Plan Review
  - Test Plan Review
  - Test Cases Review

- **Testing**
  - Unit Testing
  - Integration Testing
  - System Testing
  - Acceptance Testing
Verification versus Validation
Validation versus Verification

Requirements Backlog
(e.g., User Stories)

Work Product
Work Product
...
Work Product

Development Process

End Product
(i.e., the Software that is delivered/deployed)
Validation versus Verification

Requirements Backlog (e.g., User Stories)

Work Product

Work Product

Work Product

Development Process

End Product (i.e., the Software that is delivered/deployed)
Validation versus Verification

- Requirements Backlog (e.g., User Stories)
- Work Product
- Work Product
- Work Product...
- End Product (i.e., the Software that is delivered/deployed)

Development Process
Verification vs. Validation

Source: SEI at CMU, Donald Firesmith
Verification

Definition

- The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase.

Objective

- To ensure that the product is being built according to the requirements and design specifications. In other words, to ensure that work products meet their specified requirements.

Question

- Are we building the product *right*?
Validation

Definition
• The process of evaluating software during or at the end of the development process to determine whether it satisfies specified (or implicit) business requirements.

Objective
• To ensure that the product actually meets the user’s needs, and that the requirements were correct in the first place. In other words, to demonstrate that the product fulfills its intended use when placed in its intended environment.

Question
• Are we building the right product?

Evaluation Items:
- User requirements, Final product/software

Activities:
- Requirements review, Acceptance testing
Test Complexity – Quiz

Example:
• 30 variables, 2 levels
• Test all combinations

How long does it take to test, if 5 tests/sec can be executed automatically?

Answer choices:
1. Less than 10 sec
2. Less than 1 min
3. Less than 1 hour
4. Less than 1 day
5. Less than 1 year
6. More than 1 year
Test Complexity

Example:
• 30 variables, 2 levels
  -> \(2^{30} \approx 10^9\) combinations to test
• 5 tests/second ->
  214748364.8 sec or
  6.8 years of testing!
How to test when correct SW output is unknown?
How to test when correct SW output is unknown?

Idea: Metamorphic Testing
Example of Metamorphic Testing: Google Maps Navigator
Structure of Lecture 1

• Introduction and Motivation
• Course Information
• Basic Vocabulary
• Lab 1
Course Information/Overview

• Level: Course at bachelor's level (in English), 2nd year
• Credits: 6 ECTS
• Prerequisite:
  – Compulsory: MTAT.03.094/LTAT.05.003 Software Engineering (6 ECTS)
  – Recommended: MTAT.03.130 Object-oriented Programming (6 ECTS)
• Work load:
  – Lectures (incl. practical work): 64 person-hours – incl. lab and exam sessions
  – Independent work (outside classroom): 92 person-hours
• Assessment:
  – 10 Homework Assignments (work in pairs) – 60% of grade (~5 ph per lab = 55 ph)
  – 10 Quizzes (individual) – 10% of grade (~10 ph)
  – Exam (written) – 30% of grade (~27 ph)
• Grade scale: A (90%+), B (80%+), C (70%+), D (60%+), E (50%+), F
Letter Grades

ECTS recommended distribution:
A: 10% B: 25% C: 30% D: 25% E: 10%

ECTS = European Credit Transfer and Accumulation System

- **A** - An excellent performance, clearly outstanding. The candidate demonstrates excellent judgement and a high degree of independent thinking.
- **B** - A very good performance. The candidate demonstrates sound judgement and a very good degree of independent thinking.
- **C** - A good performance in most areas. The candidate demonstrates a reasonable degree of judgement and independent thinking in the most important areas.
- **D** - A satisfactory performance, but with significant shortcomings. The candidate demonstrates a limited degree of judgement and independent thinking.
- **E** - A performance that meets the minimum criteria, but no more. The candidate demonstrates a very limited degree of judgement and independent thinking.
- **F** - A performance that does not meet the minimum academic criteria. The candidate demonstrates an absence of both judgement and independent thinking.
Lectures (Δelta room 1021)

- Lecture 1 (11.02) – Introduction to Software Testing
- Lecture 2 (18.02) – Basic Black-Box Testing Techniques: Boundary Value Analysis & Equivalence Class Partitioning
- Lecture 3 (25.02) – BBT advanced: Combinatorial Testing
- Lecture 4 (04.03) – Basic White-Box Testing Techniques: Control-Flow Coverage
- Lecture 5 (11.03) – BBT adv.: State-Transition, Metamorphic, Random Testing
- Lecture 6 (18.03) – Test Levels, Test Tools, Test Automation
- Lecture 7 (25.03) – BBT adv.: Exploratory Testing, Behaviour Testing
- Lecture 9 (08.04) – Data-Flow Testing / Test-Suite Effectiveness: Mutation Testing
- Lecture 10 (15.04) – WBT adv.: Symbolic Execution, Static Code Analysis, Review
- Lecture 11 (22.04) – Defect Estimation / Test Documentation, Organisation and Process Improvement (Test Maturity Model)
- Lectures 12+13 (29.04 + 06.05) – Industry Guest Lectures + Advanced Topics
- Lecture 14 (13.05) – Exam Preparation
### Lectures / Labs (HW) / Quiz Schedule

<table>
<thead>
<tr>
<th>W24</th>
<th>W25</th>
<th>W26</th>
<th>W27</th>
<th>W28</th>
<th>W29</th>
<th>W30</th>
<th>W31</th>
<th>W32</th>
<th>W33</th>
<th>W34</th>
<th>W35</th>
<th>W36</th>
<th>W37</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lec 1</td>
<td>Lec 2</td>
<td>Lec 3</td>
<td>Lec 4</td>
<td>Lec 5</td>
<td>Lec 6</td>
<td>Lec 7</td>
<td>Lec 8</td>
<td>Lec 9</td>
<td>Lec 10</td>
<td>Lec 11</td>
<td>Lec 12</td>
<td>Lec 13</td>
<td>Lec 14</td>
</tr>
<tr>
<td>X</td>
<td>HW 1</td>
<td>X</td>
<td>HW 2</td>
<td>HW 3</td>
<td>HW 4</td>
<td>HW 5</td>
<td>HW 6</td>
<td>HW 7</td>
<td>HW 8</td>
<td>HW 9</td>
<td>HW 10</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Q 1</td>
<td>Q 2</td>
<td>Q 3</td>
<td>Q 4</td>
<td>Q 5</td>
<td>Q 6</td>
<td>Q 7</td>
<td>Q 8</td>
<td>Q 9</td>
<td>Q 10</td>
<td>Q 11</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Lab Assignments (HW) to be submitted via course wiki**

(1 week time / strict deadlines / 24 h grace period with penalty – afterwards 0 points)

**Quizzes to be done in Moodle**

(2 attempts / 15 min each / Fri 9:30 am– Mon 17:30 am / 10 best count)
Lab Sessions (Delta -... various rooms)

Preparation, Execution, Report – Work in Pairs

Lab 1 (week 25: Feb 16 & 17) - Debugging (10 marks)

Lab 2 (week 27: Mar 02 & 03) - Basic Black-Box-Testing (10 marks)
Lab 3 (week 28: Mar 09 & 10) - Combinatorial Testing (10 marks)
Lab 4 (week 29: Mar 16 & 17) - Basic White-Box Testing (10 marks)
Lab 5 (week 30: Mar 23 & 24) - Random Testing (10 marks)
Lab 6 (week 31: Mar 30 & 31) - Automated Web-App Testing (10 marks)
Lab 7 (week 32: Apr 06 & 07) - Automated GUI || Visual Testing (10 marks)
Lab 8 (week 33: Apr 13 & 14) - Mutation Testing (10 marks)
Lab 9 (week 34: Apr 20 & 21) - Static Code Analysis (10 marks)
Lab 10 (week 35: Apr 27 & 28) - Doc Inspection and Defect Prediction (10 marks)

Send reports via submission button on course wiki before your next lab starts.
Only PDF files will be accepted.
Attend Labs !!!!!

(if you don’t, you might not fully understand the HW assignment)
Final Exam

Written exam (30%)
- Based on textbook, lectures and lab sessions
- Multiple-choice part closed book / Other parts open book & open laptop

Dates:
- Exam 1: Thursday, 20-May-2021 at 10:15-11:55 – to be confirmed
- Exam 2: Thursday, 27-May-2021, 16:15-17:55 – to be confirmed
- Retake Exam (resit): to be announced
Books on SW Testing

Software Testing: From Theory to Practice
• By: Maurício Aniche and Arie van Deursen (TU Delft, The Netherlands)
  Link: https://sttp.site

The Fuzzing Book - Tools and Techniques for Generating Software Tests
• By: Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler
  Link: https://www.fuzzingbook.org

Introduction to Software Testing
• By: P. Ammann and J. Offutt
• Year 2017 (2nd ed.)
Software Testing 2020/21 spring

Due to continuing COVID-19 restrictions in the Institute of Computer Science (and the University at large), the first lectures and practice sessions (labs) will be given online via Zoom and/or BBB. Details will be announced in the first week of February. New instructions regarding online versus in-class teaching will be announced before end of March 2021.

Software Testing

Course Responsible / Lecturer:
- Dietmar Pfahl (dietmar.pfahl at ut dot ee) - room: 3007 (Delta)

Lab Supervisors (TAs):
- Ezequiel Scott (ezequiel dot scott at ut dot ee)
- Alejandra Duque Torres
- Karoline Holter
- Alex Vil
- Hina Anwar

Lectures (begin in week 24 of the academic year, on 11-Feb-2021):
- Thursday 10:15 - 12:00, Narva mnt 18 - r1037 (Delta building)

Labs (practice learning; begin in week 25, on 16/17-Feb-2021):
- Group 1: Tuesday 10.15 - 11.45, Delta r2034 - Ezequiel Scott
- Group 2: Tuesday 10.15 - 11.45, Delta r2045 - Alejandra Duque Torres
- Group 3: Tuesday 12.15 - 13.45, Delta r2034 - Karoline Holter
- Group 4: Tuesday 12.15 - 13.45, Delta r2045 - Alex Vil
- Group 5: Wednesday 12.15 - 13.45, Delta r2034 - Ezequiel Scott
- Group 6: Wednesday 12.15 - 13.45, Delta r2045 - Hina Anwar

Quizzes:
- Quizzes are open after each lecture from Friday (9:30 am) to Monday (17:30 am). Quizzes are managed in Moodle (link)
SIGN UP TO MESSAGE BOARD (Slack)

(if you don’t, you will miss up to date info)
Sign-Up Link for Slack

• Before the first labs start next week, please sign up to the course Slack channel. You will get your homework feedback exclusively via Slack from the lab supervisors.

Here is the sign-up link:

• <see link in email of Monday, Feb 8>
Structure of Lecture 1

- Introduction and Motivation
- Course Information
- Basic Vocabulary
- Lab 1
Test Case

- A Test Case is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.

- Templates and examples of formal test case documentation can be found here:

  http://softwaretestingfundamentals.com/test-case/
Test Case

A **Test Case** consists of:

- A set of inputs + expected outputs
- Execution conditions

Example of ‘execution condition’:

When pressing the ‘save’ button of a word processor, what happens depends on what you did previously (e.g., what you typed in or deleted)

---

**Test Suite** = set of Test Cases

**Test Data** = input to a Test Case

**Test Oracle** = condition that determines whether a test case passed or failed (→ fail happens if actual output is different from expected output)

**Test Verdict** = decision of whether a test passed or failed
Test Case – Recommendations

- As far as possible, write test cases in such a way that you test only one thing at a time. Do not overlap or complicate test cases. Attempt to make your test cases ‘atomic’.

- Ensure that all positive scenarios and negative scenarios are covered.

- Language:
  - Write in simple and easy to understand language.
  - Use active voice: Do this, do that.
  - Use exact and consistent names (of forms, fields, etc).

- Characteristics of a good test case:
  - Accurate: Exacts the purpose.
  - Economical: No unnecessary steps or words.
  - Traceable: Capable of being traced to requirements.
  - Repeatable: Can be used to perform the test over and over.
  - Reusable: Can be reused if necessary.
Test Script

- A **Test Script** is a set of instructions (written using a scripting/programming language) that is performed on a system under test to verify that the system performs as expected.
  - Test scripts are used in automated testing.

- Examples of Test Frameworks supporting test scripting:
  - JUnit, Selenium, Sikuli, …
Test Script – Examples

JUnit

@test
class Customer { // Renter
    @Test
class Rental { // contains Movie
        type ("TextA")
        click (ImageButtonA)
        assertExist (ImageResultA)
    }

    @Test
    public void shortRegularRental() {
        Customer customer = new Customer("Cust");
        Movie movie = new Movie("Groundhog Day", REGULAR);
        Rental rental = new Rental(movie, 2); // 2 days rental = short
        customer.addRental(rental);

        String expected = "Rental Record for Cust\n"
        expected += "Groundhog Day\t2.0\n"
        expected += "Amount owed is 2.0\n"
        expected += "You earned 1 frequent renter points"

        Assert.assertEquals(expected, customer.statement());
    }

    def sample_test_script (self):
        type ("TextA")
        click (ImageButtonA)
        assertExist (ImageResultA)
What is a ‘Bug’ in SE?
First ‘Computer Bug’ in 1947

The term "bug" was used in an account by computer pioneer Grace Hopper, who publicized the cause of a malfunction in an early electromechanical computer. (Harvard’s Mark II relay computer)

Source: https://en.wikipedia.org/wiki/Software_bug
What is a Bug in SE?

Error?

Fault?

if amountOf(baby) > 1
    answer = "Twins";
if equals(baby, baby1)
    answer = "Twins";
print(answer);
...
Definition 1: Error – Fault – Failure
(according to IEEE Standard)

- **Failure** is an event caused by a **fault**, and a **fault** is an anomaly of the software caused by an **error**
  - **Error** – mistake made by human (e.g., programmer)
  - **Fault** – wrong/missing statement in the software (code)
  - **Failure** – inability to perform the program’s required functions (correctly)
  - Defect ? – Bug ?

- **Debugging** / Fault localization – localizing, repairing, re-testing.
Origins and Impact of Faults

Fault sources:
- Lack of skills/training
- Oversight
- Poor communication
- ‘Lost in translation’
- Immature process

Fault context:
- Errors
- Faults
- Failures

Impact on / of software program

User’s point of view:
- Poor quality software
- User dissatisfaction

Source:
Fig 3.1 in I. Burnstein: Practical Software Testing
Definition 2: Error – Fault – Failure
(as it is often used in IDEs/tools)

- **Failure** is an event caused by an **error**, **error** is a state of the program caused by a **fault** in the code.
  - **Fault** – wrong/missing statement in code (resulting in error)
  - **Error** – incorrect program state (may result in a failure)
  - **Failure** – inability to perform its required functions (correctly)
  - **Defect**? – **Bug**?

- **Debugging** / Fault localization – localizing, repairing, re-testing.
Definition 2: Error – Fault – Failure

Example:

```java
public static int numZero (int[] x) {
    // Effects: if x==null throw NullPointerException
    //          else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] == 0) {
            count++;
        }
    }
    return count;
}
```

Inputs: Correct (=Expected) result?
- x = [2,7,0]  Actual result?
- x = [0,7,2]  Fault? Error? Failure?

Program state: x, i, count, PC
Definition 2: Error – Fault – Failure

We have seen ...

Fault = Yes
Error = Yes
------------------
Failure = No
or
Failure = Yes
Definition 2: Error – Fault – Failure

Could any of this happen?

Fault = No  
Error = No  
-----------------  
Failure = No  
or  
Failure = Yes

Fault = No  
Error = Yes  
-----------------  
Failure = No  
or  
Failure = Yes

Fault = Yes  
Error = No  
-----------------  
Failure = No  
or  
Failure = Yes
Structure of Lecture 1

• Introduction and Motivation
• Course Information
• Basic Vocabulary
• Lab 1
Lab 1 – Debugging

System 1
- Issue 1
- Issue 2
- Issue 3

System 2
- Issue 1
- Issue 2

Faults?
Faults?
Faults?
Faults?

Localization steps &
heuristics
for each fault

Use IntelliJ
Debugger

Submission:
At 23:59 on day before next lab
Lab 1 – Debugging

- Thought process for setting breakpoints and deciding where to step in next to see the program state.
Starting Point: Issue Report

- Example Report
  - Admin Data
  - Short Description
  - Reproduction Steps (input)
  - Expected vs Actual Result
  - Additional Information (screen shots, stack traces, etc.)
- Comments / Discussion
Issue Report – Sys_1: HeapSort

- **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:**
  ```java
  List<Integer> heapList = new ArrayList<>();
  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);
  System.out.println("After heapifying:");
  System.out.println(Heap.heapify(heapList));
  System.out.println(heapList);
  ```

  **Expected 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]

  **Actual output:**
  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]
### System 1: HeapSort – Build the Heap

<table>
<thead>
<tr>
<th>Heap</th>
<th>New element</th>
<th>Swap elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6, 5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6, 5, 3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6, 5, 3, 1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6, 5, 3, 1, 8</td>
<td></td>
<td>5, 8</td>
</tr>
<tr>
<td>6, 8, 3, 1, 5</td>
<td></td>
<td>6, 8</td>
</tr>
<tr>
<td>8, 6, 3, 1, 5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8, 6, 3, 1, 5, 7</td>
<td></td>
<td>7, 3</td>
</tr>
<tr>
<td>8, 6, 7, 1, 5, 3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8, 6, 7, 1, 5, 3, 2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8, 6, 7, 1, 5, 3, 2, 4</td>
<td></td>
<td>1, 4</td>
</tr>
<tr>
<td>8, 6, 7, 4, 5, 3, 2, 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

List to be sorted in ascending order:

\[
[6 \ 5 \ 3 \ 1 \ 8 \ 7 \ 2 \ 4]
\]

Heapified list:

\[
[8 \ 6 \ 7 \ 4 \ 5 \ 3 \ 2 \ 1]
\]
# System 1: HeapSort – Sorting

<table>
<thead>
<tr>
<th>Heap</th>
<th>Swap elements</th>
<th>Delete element</th>
<th>Sorted array</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>8, 6, 7, 4, 5, 3, 2, 1</td>
<td>8, 1</td>
<td></td>
<td></td>
<td>Swap 8 and 1 to delete 8 from heap</td>
</tr>
<tr>
<td>1, 6, 7, 4, 5, 3, 2, 8</td>
<td></td>
<td>8</td>
<td></td>
<td>Delete 8 from heap &amp; add to sorted array</td>
</tr>
<tr>
<td>1, 6, 7, 4, 5, 3, 2</td>
<td>1, 7</td>
<td></td>
<td>8</td>
<td>Swap 1 and 7 as they are not in order</td>
</tr>
<tr>
<td>7, 6, 1, 4, 5, 3, 2</td>
<td>1, 3</td>
<td></td>
<td>8</td>
<td>Swap 1 and 3 as they are not in order</td>
</tr>
<tr>
<td>7, 6, 3, 4, 5, 1, 2</td>
<td>7, 2</td>
<td></td>
<td>8</td>
<td>Swap 7 and 2 to delete 7 from heap</td>
</tr>
<tr>
<td>2, 6, 3, 4, 5, 1, 7</td>
<td></td>
<td>7</td>
<td>8</td>
<td>Delete 7 from heap &amp; add to sorted array</td>
</tr>
<tr>
<td>2, 6, 3, 4, 5, 1</td>
<td>2, 6</td>
<td></td>
<td>7, 8</td>
<td>Swap 2 and 6 as they are not in order</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
# System 1: HeapSort – Sorting /1

<table>
<thead>
<tr>
<th>Heap</th>
<th>Swap elements</th>
<th>Delete element</th>
<th>Sorted array</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>3, 2, 1</td>
<td>3, 1</td>
<td></td>
<td>4, 5, 6, 7, 8</td>
<td><strong>Swap 3 and 1 to delete 3 from heap</strong></td>
</tr>
<tr>
<td>1, 2, 3</td>
<td></td>
<td>3</td>
<td>4, 5, 6, 7, 8</td>
<td><strong>Delete 3 from heap &amp; add to sorted array</strong></td>
</tr>
<tr>
<td>1, 2</td>
<td>1, 2</td>
<td></td>
<td>3, 4, 5, 6, 7, 8</td>
<td><strong>Swap 1 and 2 as they are not in order</strong></td>
</tr>
<tr>
<td>2, 1</td>
<td>2, 1</td>
<td></td>
<td>3, 4, 5, 6, 7, 8</td>
<td><strong>Swap 2 and 1 to delete 2 from heap</strong></td>
</tr>
<tr>
<td>1, 2</td>
<td></td>
<td>2</td>
<td>3, 4, 5, 6, 7, 8</td>
<td><strong>Delete 2 from heap &amp; add to sorted array</strong></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>2, 3, 4, 5, 6, 7, 8</td>
<td><strong>Delete 1 from heap &amp; add to sorted array</strong></td>
</tr>
</tbody>
</table>

*Completed: 1, 2, 3, 4, 5, 6, 7, 8*

https://en.wikipedia.org/wiki/Heapsort
Issue Report 1– Sys_2: 8-Queens Problem

- 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:**
  
  ```java
  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
Issue Report 1 – Sys_2: 8-Queens Problem

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

Expected output:
<table>
<thead>
<tr>
<th>Generation</th>
<th>Current highest fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;?&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;?&gt;</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Found suitable board state on generation <?: <x1,x2,x3>
Here is the found solution as a board where . marks an empty cell.
```
X . .
. . X
. . .
```

Actual output:
<table>
<thead>
<tr>
<th>Generation</th>
<th>Current highest fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>38</td>
</tr>
</tbody>
</table>

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– Sys_2: 8-Queens Problem

* 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:
```java
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);
    }
```
Issue Report 2 – Sys_2: 8-Queens Problem

- 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 in under 100 generations with population size 100

Input code:
```java
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.remove(0);
}
```

Expected output:
- Average generation count < 87
- \( P(\text{generation count} \leq 100) > 75\% \)

Actual output generalized and specific:
- Average generation count > 87
- \( P(\text{generation count} \leq 100) < 75\% \)
- Average generation count: 109.916
- \( P(\text{generation count} < 100) : 57.7\% \)

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)
Representation of Chess Board

Queen

[ 2 4 1 6 7 0 2 3 ]

5th row, 8th position
8th position = index 7
Genetic Algorithm
Genetic Algorithm

Size = 4

Fitness of first individual? +1 in the fitness function per each queen per each clash
Genetic Algorithm

Generate new population:
- Sort by fitness value
- Take upper half
- Then generate new board allocations based on allocation pairs in the upper half

Size = 4
Genetic Algorithm

Generate new population:

Size = 4
To Do & Next Week

• Quiz 1 (in Moodle!):
  – Opens tomorrow morning – closes on Monday before noon!

• Lab 1:
  – Debugging

Next Week! Tue/Wed

• Lecture 2:
  – Basic Black-Box and White-Box Testing Techniques (intro)