

# Lecture 0 – Introduction

## AAA705: Software Testing and Quality Assurance

Jihyeok Park



2024 Spring

- **Instructor:** Jihyeok Park (박지혁)
  - **Position:** Assistant Professor in CS, Korea University
  - **Expertise:** Programming Languages, Software Analysis
  - **Office hours:** 14:00–16:00, Tuesdays (appointment by e-mail)
  - **Office:** 609A, Science Library Bldg
  - **Email:** jihyeok\_park@korea.ac.kr

- **Instructor:** Jihyeok Park (박지혁)
  - **Position:** Assistant Professor in CS, Korea University
  - **Expertise:** Programming Languages, Software Analysis
  - **Office hours:** 14:00–16:00, Tuesdays (appointment by e-mail)
  - **Office:** 609A, Science Library Bldg
  - **Email:** jihyeok\_park@korea.ac.kr
- **Class:** AAA705: Software Testing and Quality Assurance

- **Instructor:** Jihyeok Park (박지혁)
  - **Position:** Assistant Professor in CS, Korea University
  - **Expertise:** Programming Languages, Software Analysis
  - **Office hours:** 14:00–16:00, Tuesdays (appointment by e-mail)
  - **Office:** 609A, Science Library Bldg
  - **Email:** jihyeok\_park@korea.ac.kr
- **Class:** AAA705: Software Testing and Quality Assurance
- **Lectures:** 15:00–17:45, Mon. and Wed. @ 107 미래융합기술관

- **Instructor:** Jihyeok Park (박지혁)
  - **Position:** Assistant Professor in CS, Korea University
  - **Expertise:** Programming Languages, Software Analysis
  - **Office hours:** 14:00–16:00, Tuesdays (appointment by e-mail)
  - **Office:** 609A, Science Library Bldg
  - **Email:** jihyeok\_park@korea.ac.kr
- **Class:** AAA705: Software Testing and Quality Assurance
- **Lectures:** 15:00–17:45, Mon. and Wed. @ 107 미래융합기술관
- **Homepage:** <https://plrg.korea.ac.kr/courses/aaa705/>

Weak	Date	Contents
1	03/04	Introduction
	03/06	Combinatorial Testing
2	03/11	Random Testing
	03/13	Coverage Criteria (1)
3	03/18	Coverage Criteria (2)
	03/20	Search Based Software Testing (SBST)
4	03/25	Dynamic Symbolic Execution (DSE)
	03/27	Mutation Testing
5	04/01	Regression Testing
	04/03	Fault Localization
6	04/08	Metamorphic Testing
7	04/15	Differential Testing
	04/17	Course Review
12	05/20	Project Presentation
	05/22	Project Presentation

- **Homework Assignments: 90%**
  - **2 Programming Assignments:**
    - Homework 1: 20% (due on March 27)
    - Homework 2: 20% (due on April 17)
  - Submit your homework on **Blackboard**.

- **Homework Assignments: 90%**
  - **2 Programming Assignments:**
    - Homework 1: 20% (due on March 27)
    - Homework 2: 20% (due on April 17)
  - Submit your homework on **Blackboard**.
- **Project: 50%** (due on May 20)
  - Personal project. No team project.
  - Presentation on May 20 (Mon.) and May 22 (Wed.) 15:00 – 17:45



- **Homework Assignments: 90%**
  - **2 Programming Assignments:**
    - Homework 1: 20% (due on March 27)
    - Homework 2: 20% (due on April 17)
  - Submit your homework on **Blackboard**.
- **Project: 50%** (due on May 20)
  - Personal project. No team project.
  - Presentation on May 20 (Mon.) and May 22 (Wed.) 15:00 – 17:45
- **Attendance: 10%**

- **Self-contained lecture notes.**

<https://plrg.korea.ac.kr/courses/aaa705/>

(Special thanks to Prof. **Shin Yoo** @ KAIST)

- **Self-contained lecture notes.**

<https://plrg.korea.ac.kr/courses/aaa705/>

(Special thanks to Prof. **Shin Yoo** @ KAIST)

- **Reference:** we do not teach these books and these books do not contain answers to this course.
  - **“Introduction to Software Testing (2nd Ed.)”** by Paul Ammann and Jeff Offutt.
  - **“Why Programs Fail (2nd Ed.)”** by Andreas Zeller.

## 1. Why Software Testing?

## 2. Terminologies in Software Testing

Types of Software Quality

Faults vs. Errors vs. Failures

More Terminologies

## 3. Software Testing Techniques

Unexpected faults in **safety-critical software** cause serious problems:

<p><b>June 4, 1996: Ariane-5 explodes after lift off</b></p> <p>Today in History: June 4, 1996: Ariane-5 explodes after lift off</p> <p>Copyright: June 06, 2016 0:03 Abdellatif, David of Archibute</p> 	<p><b>Knight Capital Says Trading Glitch Cost It</b></p> <p>BY NATHANIEL POPPER AUGUST 3, 2013 9:07 AM 398</p> <p>Runaway Trades Spread Turmoil Across Wall St.</p> 	<p><b>Heathrow Airport apologises for IT failure disruption</b></p> <p>3:16 February 2020</p> 	<p><b>Cruise recalls all its driverless cars</b></p> <p>pedestrian hit and dragged</p> <p>In another setback, Cruise updates software on 950 driverless cars to fix its 'Collision Detection'</p> <p>By David Stewart</p> <p>Updated November 6, 2023 at 2:22 p.m. EST Published November 6, 2023 at 2:08 p.m. EST</p> 
<b>Rocket</b>	<b>Financial</b>	<b>Airport</b>	<b>Auto. Vehicle</b>
(1996)	(2012)	(2020)	(2023)

Unexpected faults in **safety-critical software** cause serious problems:

<p><b>June 4, 1996: Ariane-5 explodes after lift off</b></p> <p>Today in History: June 4, 1996: Ariane-5 explodes after lift off</p> <p>Published November 6, 2023 at 2:52 p.m. EST · Published November 6, 2023 at 2:52 p.m. EST</p> 	<p><b>Knight Capital Says Trading Glitch Cost It</b></p> <p>BY NATHANIEL POPPER AUGUST 3, 2013 9:07 AM · 398</p> <p>Runaway Trades Spread Turmoil Across Wall St.</p> 	<p><b>Heathrow Airport apologises for IT failure disruption</b></p> <p>3:16 February 2020</p> 	<p><b>Cruise recalls all its driverless cars</b></p> <p>pedestrian hit and dragged</p> <p>In another setback, Cruise updates software on 950 driverless cars to fix its 'Collision Detection'</p> <p>Published November 6, 2023 at 2:52 p.m. EST · Published November 6, 2023 at 2:52 p.m. EST</p> 
<b>Rocket</b>	<b>Financial</b>	<b>Airport</b>	<b>Auto. Vehicle</b>
(1996)	(2012)	(2020)	(2023)

Then, how can we **prevent** such software faults?

Unexpected faults in **safety-critical software** cause serious problems:

<p><b>June 4, 1996: Ariane-5 explodes after lift off</b></p> <p>Today in History: June 4, 1996: Ariane-5 explodes after lift off</p> <p>Published November 6, 2023 at 2:55 p.m. CST · Published November 6, 2023 at 2:55 p.m. CST</p> 	<p><b>Knight Capital Says Trading Glitch Cost It</b></p> <p>BY NATHANIEL POPPER AUGUST 3, 2012 9:07 AM · 398</p> <p>Runaway Trades Spread Turmoil Across Wall St.</p> 	<p><b>Heathrow Airport apologises for IT failure disruption</b></p> <p>3:16 February 2020</p> 	<p><b>Cruise recalls all its driverless cars</b></p> <p>It's another setback. Cruise updates software on 950 driverless cars to fix its 'Collision Detection' software.</p> <p>Published November 6, 2023 at 2:55 p.m. CST · Published November 6, 2023 at 2:55 p.m. CST</p> 
<b>Rocket</b>	<b>Financial</b>	<b>Airport</b>	<b>Auto. Vehicle</b>
(1996)	(2012)	(2020)	(2023)

Then, how can we **prevent** such software faults?

Can we **automatically check** whether a program does not have any software faults?

# Detecting Software Faults

How do we know whether a software is correct?



How do we know whether a software is correct?



**Empiricists – Francis Bacon**

*It is correct because I **TESTED**  
several times but no error was found!*

**VS.**



**Rationalists – René Descartes**

*It is correct because I formally  
**PROVED** that no error exists!*

We can use various **analysis** techniques to detect software faults.



An **analyzer** is a program that takes a **program** and a **property** as inputs and determines whether the program **satisfies** the property.

We can use various **analysis** techniques to detect software faults.



An **analyzer** is a program that takes a **program** and a **property** as inputs and determines whether the program **satisfies** the property.

We can categorize them into two groups:

- **Dynamic analyzers** analyze programs by **executing** them.
- **Static analyzers** analyze programs **without executing** them.

# Dynamic Analysis vs. Static Analysis

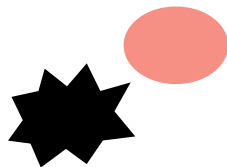
★ : Possible States    ● : Error States    ⋮ : Dynamic Analysis    ⬡ : Static Analysis



P<sub>1</sub>



P<sub>2</sub>

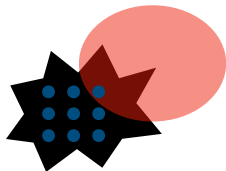


P<sub>3</sub>

Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

# Dynamic Analysis vs. Static Analysis

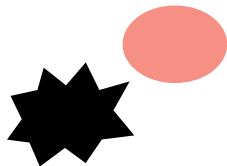
★ : Possible States    ● : Error States    ⋮ : Dynamic Analysis    ◈ : Static Analysis



P<sub>1</sub>



P<sub>2</sub>

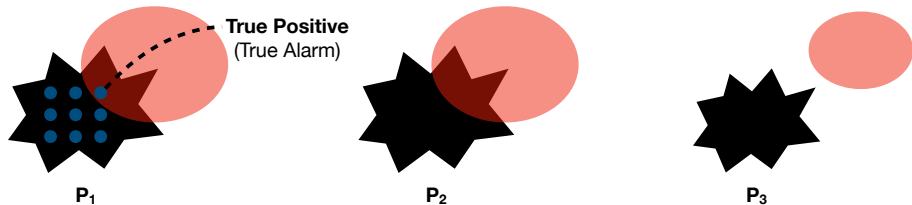


P<sub>3</sub>

Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

# Dynamic Analysis vs. Static Analysis

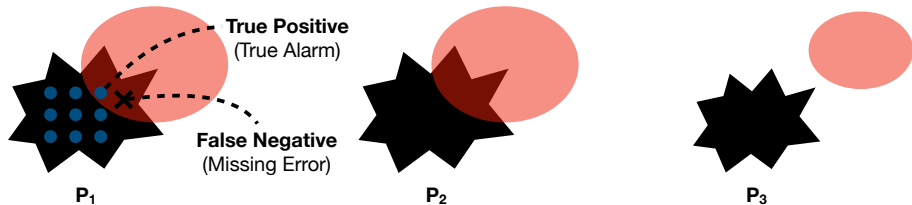
★ : Possible States    ● : Error States    ⋮ : Dynamic Analysis    ◈ : Static Analysis



Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

# Dynamic Analysis vs. Static Analysis

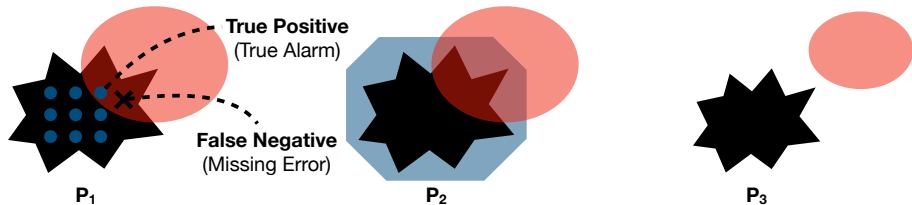
★ : Possible States    ● : Error States    ⋮ : Dynamic Analysis    ◈ : Static Analysis



Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

# Dynamic Analysis vs. Static Analysis

★ : Possible States    ● : Error States    ⋮ : Dynamic Analysis    ⬡ : Static Analysis

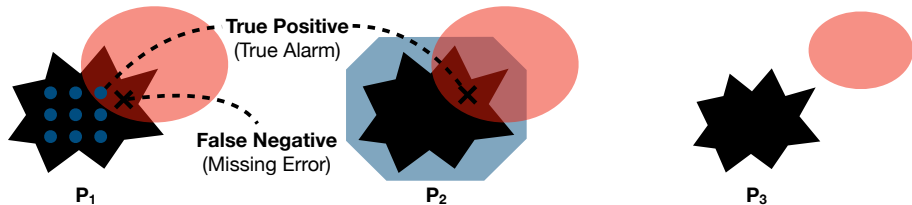


Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)



# Dynamic Analysis vs. Static Analysis

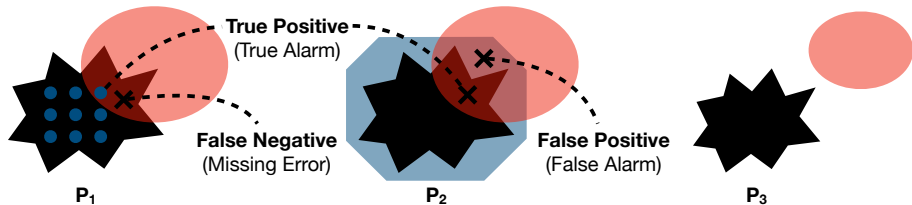
★ : Possible States    ● : Error States    ⋮ : Dynamic Analysis    ⬡ : Static Analysis



Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

# Dynamic Analysis vs. Static Analysis

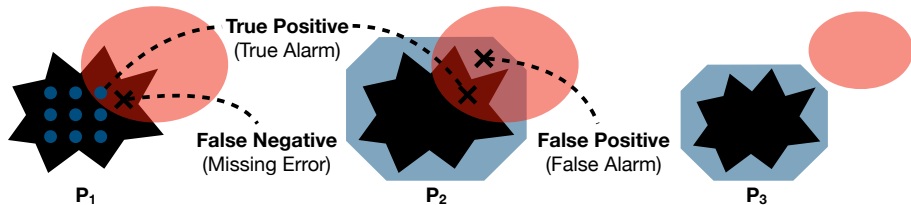
★ : Possible States    ● : Error States    ⋮ : Dynamic Analysis    ⬡ : Static Analysis



Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

# Dynamic Analysis vs. Static Analysis

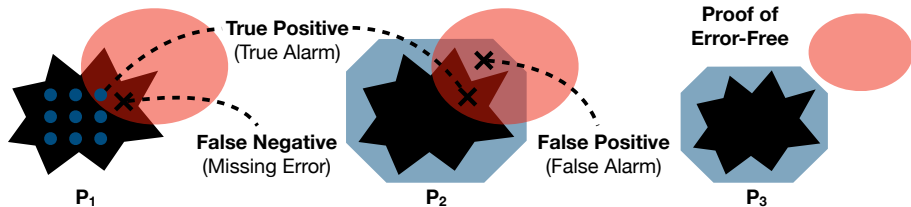
★ : Possible States    ● : Error States    ⋮ : Dynamic Analysis    ⬡ : Static Analysis



Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)

# Dynamic Analysis vs. Static Analysis

★ : Possible States    ● : Error States    ⋮ : Dynamic Analysis    ⬡ : Static Analysis



Dynamic Analysis	Static Analysis
Software Testing	Formal Verification
Empiricists	Rationalists
Under-approximation	Over-approximation
False Negatives (Missed Errors)	False Positives (False Alarms)



- Imagine you have two choices when boarding a airplane:
  - While an airplane A has **never been proven** to have any run-time errors, it has been **tested** with a finite number of test flights.
  - While an airplane B has been **formally verified** to have no run-time errors, it has **never been tested** in the real world.



- Imagine you have two choices when boarding a airplane:
  - While an airplane A has **never been proven** to have any run-time errors, it has been **tested** with a finite number of test flights.
  - While an airplane B has been **formally verified** to have no run-time errors, it has **never been tested** in the real world.
- Some people may choose A, while others may choose B.



- Imagine you have two choices when boarding a airplane:
  - While an airplane A has **never been proven** to have any run-time errors, it has been **tested** with a finite number of test flights.
  - While an airplane B has been **formally verified** to have no run-time errors, it has **never been tested** in the real world.
- Some people may choose A, while others may choose B.
- In addition, some properties only can be **tested** but not **verified** (e.g., energy consumption, usability, etc.).

## 1. Why Software Testing?

## 2. Terminologies in Software Testing

- Types of Software Quality

- Faults vs. Errors vs. Failures

- More Terminologies

## 3. Software Testing Techniques



**Software testing** *is an **investigation** conducted to provide stakeholders with information about the **quality** of the product or service under test.*

The software should be **dependable**: **correct**, **reliable**, **safe**, and **robust**.

The software should be **dependable**: **correct**, **reliable**, **safe**, and **robust**.

- **Correctness**: the software should exactly **conform** to its **formal specification**.

The software should be **dependable**: **correct**, **reliable**, **safe**, and **robust**.

- **Correctness**: the software should exactly **conform** to its **formal specification**.
- **Reliability**: the software should have a **high probability** of being **correct** for period of time.

The software should be **dependable**: **correct**, **reliable**, **safe**, and **robust**.

- **Correctness**: the software should exactly **conform** to its **formal specification**.
- **Reliability**: the software should have a **high probability** of being **correct** for period of time.
- **Safety**: the software should be **no risk** of any kind of **hazard** (loss of life, injury, etc.).

The software should be **dependable**: **correct**, **reliable**, **safe**, and **robust**.

- **Correctness**: the software should exactly **conform** to its **formal specification**.
- **Reliability**: the software should have a **high probability** of being **correct** for period of time.
- **Safety**: the software should be **no risk** of any kind of **hazard** (loss of life, injury, etc.).
- **Robustness**: the software should reasonably **remain dependable** even if surrounding **environment changes**.

Apart from dependability, the software should meet certain **performance** expectations.

Apart from dependability, the software should meet certain **performance** expectations.

- For example, execution time, network throughput, memory usage, number of simultaneous users, etc.



Apart from dependability, the software should meet certain **performance** expectations.

- For example, execution time, network throughput, memory usage, number of simultaneous users, etc.
- **Hard to thoroughly test** due to the heavy reliance on the **execution environment** and **usage patterns**.

# Types of Software Quality: Usability

The software should be **usable**.

The software should be **usable**.

- In general, there is no universally accepted criterion for **usability**.

The software should be **usable**.

- In general, there is no universally accepted criterion for **usability**.
- Usability testing usually involves **user studies**, such as **focus groups**, **beta-testing**, **A/B testing**, etc.

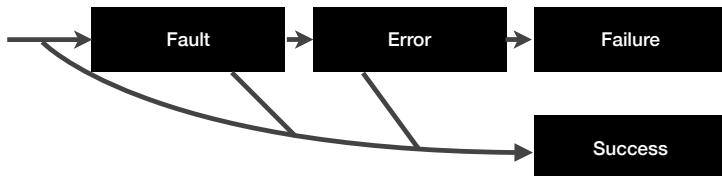
# Types of Software Quality: Ethics

The software should be **ethical**.

The software should be **ethical**.

- Typically, this is applied to AI/ML based systems.
- **[FSE'17]** S. Galhotra, Y. Brun, and A. Meliou. "*Fairness testing: testing software for discrimination.*"
- **[ASE'18]** S. Udeshi, P. Arora, and S. Chattopadhyay. "*Automated directed fairness testing.*"
- **[ICSE'20]** P. Zhang, J. Wang, J. Sun, G. Dong, X. Wang, X. Wang, J. S. Dong, and T. Dai. "*White-box fairness testing through adversarial sampling.*"

The purpose of testing is to **detect** and **remove** faults, errors, and failures.



From **IEEE Standard 729-1983**, IEEE Standard Glossary of Software Engineering Terminology<sup>1</sup>

- **Fault**: an anomaly in the software that may lead to an error.
- **Error**: a runtime effect of executing a fault, which may cause a failure.
- **Failure**: a manifestation of an error external to the software.

<sup>1</sup><https://ieeexplore.ieee.org/document/7435207>

We want to implement a JavaScript function that computes the sum of elements in a given array.

```
function sum(arr) {  
  let result = 0;  
  for (let i = 0; i < arr.length; i++) {  
    // fault: `i` should be fixed to `arr[i]`  
    result += i;  
  }  
  return result;  
}
```

It is a **fault** but **not an error** until the function is executed.

```
// the faulty statement is not reached at runtime (no error)  
assert(sum([]) === 0);
```



We want to implement a JavaScript function that computes the sum of elements in a given array.

```
function sum(arr) {  
  let result = 0;  
  for (let i = 0; i < arr.length; i++) {  
    // fault: `i` should be fixed to `arr[i]`  
    result += i;  
  }  
  return result;  
}
```

It is an **error** with the following input but **not a failure** because the output is **coincidentally correct**.

```
// the faulty statement is reachable at runtime (error)  
// the output is coincidentally correct (no failure)  
assert(sum([4, -2, 1]) === 3);
```

We want to implement a JavaScript function that computes the sum of elements in a given array.

```
function sum(arr) {  
  let result = 0;  
  for (let i = 0; i < arr.length; i++) {  
    // fault: `i` should be fixed to `arr[i]`  
    result += i;  
  }  
  return result;  
}
```

It is a **failure** with the following input because the output is **incorrect**.

```
// the output is incorrect (failure)  
assert(sum([3, 7, 4]) === 14);
```

- **Test Input:** a set of inputs that are used to test a program.

- **Test Input:** a set of inputs that are used to test a program.
- **Test Oracle:** a mechanism to determine whether the program behaves correctly.

- **Test Input:** a set of inputs that are used to test a program.
- **Test Oracle:** a mechanism to determine whether the program behaves correctly.
- **Test Case:** a pair of a test input and a test oracle.

- **Test Input:** a set of inputs that are used to test a program.
- **Test Oracle:** a mechanism to determine whether the program behaves correctly.
- **Test Case:** a pair of a test input and a test oracle.
- **Test Suite:** a set of test cases.

- **Test Input:** a set of inputs that are used to test a program.
- **Test Oracle:** a mechanism to determine whether the program behaves correctly.
- **Test Case:** a pair of a test input and a test oracle.
- **Test Suite:** a set of test cases.
- **Test Effectiveness:** the ability of a test suite to detect faults or achieve other testing objectives.

- **Test Input:** a set of inputs that are used to test a program.
- **Test Oracle:** a mechanism to determine whether the program behaves correctly.
- **Test Case:** a pair of a test input and a test oracle.
- **Test Suite:** a set of test cases.
- **Test Effectiveness:** the ability of a test suite to detect faults or achieve other testing objectives.
- **Testing vs. Debugging:** testing is the process of detecting faults, while debugging is the process of fixing faults.



## 1. Why Software Testing?

## 2. Terminologies in Software Testing

Types of Software Quality

Faults vs. Errors vs. Failures

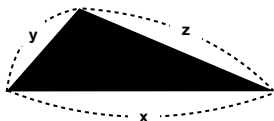
More Terminologies

## 3. Software Testing Techniques

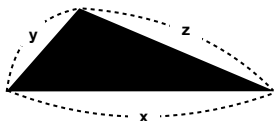
- **Exhaustive Testing:** Can we test a program with all possible inputs?

- **Exhaustive Testing:** Can we test a program with all possible inputs?  
In theory, **Yes!**

- **Exhaustive Testing:** Can we test a program with all possible inputs?  
In theory, **Yes!**
- However, it is infeasible for most programs.
- For example, consider a program that takes three 32-bit integers as inputs and returns they can form a **triangle** and **its type**.

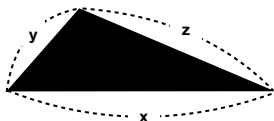


- **Exhaustive Testing:** Can we test a program with all possible inputs?  
In theory, **Yes!**
- However, it is infeasible for most programs.
- For example, consider a program that takes three 32-bit integers as inputs and returns they can form a **triangle** and **its type**.



- How many possible inputs are there?

- **Exhaustive Testing:** Can we test a program with all possible inputs?  
In theory, **Yes!**
- However, it is infeasible for most programs.
- For example, consider a program that takes three 32-bit integers as inputs and returns they can form a **triangle** and **its type**.

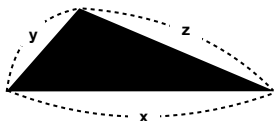


- How many possible inputs are there?

$$2^{32} \times 2^{32} \times 2^{32} = 2^{96} \approx 7.9 \times 10^{28}$$

- Approximated number of stars in the universe:  $10^{24}$

- **Exhaustive Testing:** Can we test a program with all possible inputs?  
In theory, **Yes!**
- However, it is infeasible for most programs.
- For example, consider a program that takes three 32-bit integers as inputs and returns they can form a **triangle** and **its type**.



- How many possible inputs are there?

$$2^{32} \times 2^{32} \times 2^{32} = 2^{96} \approx 7.9 \times 10^{28}$$

- Approximated number of stars in the universe:  $10^{24}$
- Testing allows only a **sampling** of an enormous **input space**.  
The difficulty lies in how to come up with **effective sampling**.

- For every test input, we need to know the **expected behavior** of the program. (i.e., the **oracle**).



- For every test input, we need to know the **expected behavior** of the program. (i.e., the **oracle**).
- How to define the **oracle**?

- For every test input, we need to know the **expected behavior** of the program. (i.e., the **oracle**).
- How to define the **oracle**?
- Without an explicit oracle, we can only small subset of faults. (e.g., **crash, unintended infinite loop, division by zero**, etc.)

- For every test input, we need to know the **expected behavior** of the program. (i.e., the **oracle**).
- How to define the **oracle**?
- Without an explicit oracle, we can only small subset of faults. (e.g., **crash, unintended infinite loop, division by zero**, etc.)
- We need to **define** or **infer** the oracle for testing.

- There is no fixed recipe for software testing.

- There is no fixed recipe for software testing.
- We need to understand the pros and cons of each testing technique.

- There is no fixed recipe for software testing.
- We need to understand the pros and cons of each testing technique.
- There are two major categories of testing techniques:
  - **Black-box Testing**: testing **without** knowing the internal structure of the program.
  - **White-box Testing**: testing **with** the knowledge of the internal structure of the program.

- **Combinatorial Testing**

- Tester utilizes **input specifications** to generate test cases.

- **Random Testing**

- Tester **randomly** selects test cases from the input space.
- It can be used for **white-box testing** as well.

Sometimes called **structural testing** because it uses the **internal structure** of the program to derive test cases.

- **Coverage Criteria**

- The adequacy of a test suite is measured in terms of the **coverage** of the program's internal structure.

- **Search Based Software Testing (SBST)**

- A technique that uses **meta-heuristic search** algorithms to maximize/minimize a certain **fitness function**.

- **Dynamic Symbolic Execution (DSE)**

- A technique that systematically explores the input space using **symbolic execution** with **dynamic analysis**.



- **Mutation Testing**

- A technique that evaluates the quality of a test suite by introducing **artificial faults** to the program.

- **Regression Testing**

- A technique that ensures that a change in the program does not introduce new faults.

- **Fault Localization**

- A technique that identifies the **location** of a fault in the program.

- **Metamorphic Testing**

- A technique that tests a program using **metamorphic relations**.

- **Differential Testing**

- A technique that tests a program by comparing the outputs of **multiple implementations**.

- Combinatorial Testing

Jihyeok Park

[jihyeok\\_park@korea.ac.kr](mailto:jihyeok_park@korea.ac.kr)

<https://plrg.korea.ac.kr>