Lecture 2 – Random Testing

AAA705: Software Testing and Quality Assurance

Jihyeok Park

PLRG

2024 Spring

Recall

- Equivalence Partitioning (EP)
- Boundary Value Analysis (BVA)
- Category Partition Method (CPM)
- Combinatorial Testing (CT)
    - Covering Array (CA)
    - Fault Detection Effectiveness
    - Greedy Algorithm – IPOG Strategy
    - Greedy vs. Meta-heuristic

Contents

Contents

Random Testing

- We need to **sample** the test input from the vast and possibly infinite **input space**.

- What happens if we just sample the input **randomly**?

  - Since developers has their own mental model of the software, they often have a **biased** view of the input space.

  - **Random testing** can help to ignore this bias.

Random Testing

- **SUT**: Software Under Test
- S: Set of all possible test inputs for SUT
- F: a subset of S – a set of all failing test inputs

$$\text{Failure Rate } t = \frac{|F|}{|S|}$$

(The probability that a randomly sampled test input is fail when we sample uniformly at random from S)

Random Testing – Example

```
/* C */
int abs(int x) {
  if (x < 0) return x;     // should be -x
  else return x;
}
```

- Failure Rate $t \approx 0.5$
- Oracle
  - `assertEqual(abs(-5), 5)`
  - `assertEqual(abs(5), 5)`

How Random Can We Get?

- **Pseudo-random number generators (PRNGs)**
  - **Middle Square Method** – Initial algorithm by John von Neumann
  - **Linear Congruential Generator** – Most popular
  - **Mersenne Twister** (1997) – `C++ 11` / PHP 7.1 – a bias bug[1][2]
  - **Xorshift** – Fast but fail some tests / variants (xorshift+, xoshiro, etc.)
- **True-random number generators** (TRNGs) – expensive
  - **Atmospheric noise** – https://random.org
  - **Quantum random number generator (QRNG)** – https://qrng.anu.edu.au
  - **Lava lamps** – **Cloudflare**

[1] https://bugs.php.net/bug.php?id=75170
[2] https://github.com/php/php-src/commit/a0724d

How Random Can We Get?

The new Galaxy Quantum 4 is equipped with the world's smallest (width 2.5mm x length 2.5mm) **Quantum Random Number Generator (QRNG)** chipset, enabling trusted authentication and encryption of information.

Probabilistic Analysis

$$\text{Failure Rate } p = \frac{|F|}{|S|}$$

- Given a failure rate p, **how many** test inputs do we need to sample to find the **first failure**?
- Given n random test inputs, what is the **probability** of finding **at least one failure**?

Probabilistic Analysis – Geometric Distribution

- The **geometric distribution** models the first occurrence of a success in a sequence of n independent (Bernoulli) trials with the same probability p.

- The most popular example is the **coin flipping**.

- The **probability mass function (PMF)** of the geometric distribution:

$$Pr(X = k) = (1 - p)^{k-1} p$$

It is the probability that the first success occurs on the n-th trial.

- Given a failure rate p, **how many** test inputs do we need to sample to find the **first failure**?

$$
\begin{aligned}
E(X) &= \sum_{k=1}^{\infty} k \cdot Pr(X = k) \\
&= \sum_{k=1}^{\infty} k \cdot (1-p)^{k-1} p \\
&= p \sum_{k=1}^{\infty} k \cdot (1-p)^{k-1} \\
&= p \left(\sum_{k=1}^{\infty} (1-p)^{k-1} + \sum_{k=2}^{\infty} (1-p)^{k-1} + \cdots \sum_{k=3}^{\infty} (1-p)^{k-1} + \cdots \right) \\
&= p \left(\frac{1}{p} + \frac{1-p}{p} + \frac{(1-p)^2}{p} + \cdots \right) \\
&= 1 + (1-p) + (1-p)^2 + \cdots = \frac{1}{p}
\end{aligned}
$$

Probabilistic Analysis – Geometric Distribution

- Given a failure rate p, **how many** test inputs do we need to sample to find the **first failure**?
- **Mean** (If p = 0.01, the average test inputs = 100)

$$\frac{1}{p}$$

- **Median** (If p = 0.01, the median test inputs $\approx$ 69)

$$\left\lceil \frac{-1}{\log_2(1-p)} \right\rceil$$

- **Variance** (If p = 0.01, the variance = 9900)

$$\frac{1-p}{p^2}$$

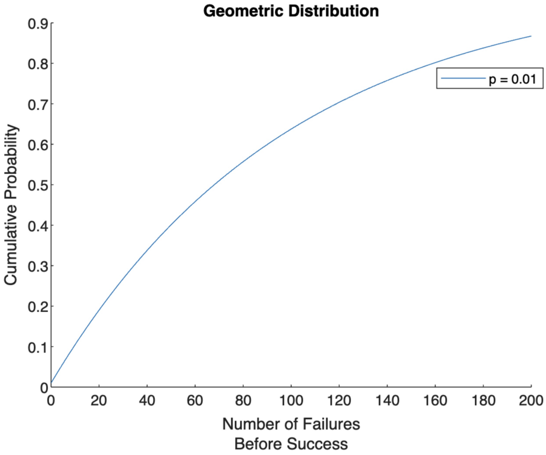
Probabilistic Analysis – Geometric Distribution

- Given n random test inputs, what is the **probability** of finding **at least one failure**?

$$
\begin{aligned}
P(X \leq n) &= \sum_{k=1}^{n} \cdot Pr(X = k) \\
&= \sum_{k=1}^{n} \cdot (1-p)^{k-1} p \\
&= p \frac{1-(1-p)^{n+1}}{1-(1-p)} \\
&= 1-(1-p)^{n+1}
\end{aligned}
$$

Probabilistic Analysis – Geometric Distribution

- If we test $n = 100$ random test inputs, the probability of finding at least one failure is $1 - (1 - 0.01)^{101} = 63.76\%$.
- If we test $n = 200$ random test inputs, the probability of finding at least one failure is $1 - (1 - 0.01)^{201} = 86.74\%$.



Probabilistic Analysis – Geometric Distribution

- Unfortunately, failure rate p is **unknown** in practice.
- But, we can **estimate** p in various ways:
  - Previous versions of the software
  - Similar software
  - Literature

Weaknesses of Random Testing

PLRG

- Random testing provides **no guidance**; it is the **needle in a haystack** problem – the probability of finding a failure is low.

```c
/* C */
void foo(int x) {
  if (x == 0) {
    /* faulty code here */
  }
}
```

```python
# Python
def foo(x):
  # e.g., x = 2840
  if (x * 7919 % 5711 == 42):
    # faulty code here
}
```

- We need **biased** random testing with predefined probability:
  - **Special values** (-0, `null`, π, . . .)
  - **Extracted values** from code (e.g., constants, literals)
  - **Previously successful values**

Examples

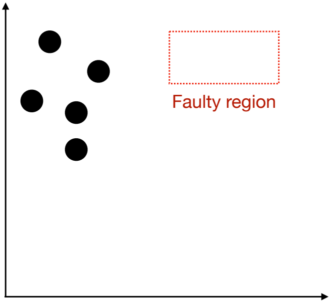
- **Apple** (1983) - “Monkey” for random events (e.g., mouse clicks, key presses, etc.) to test the robustness of the MacWrite and MacPaint applications.
- **Amazon** (2003) - “Game day” for website reliability
- **Google** (2006) - “DiRT”’ for Site Reliability Engineering (SRE)
- **Netflix** (2011) – **“Chaos Monkey”** that randomly terminates AWS instances to test the fault tolerance of the Netflix infrastructure.

Contents

Adaptive Random Testing (ART)

- **Insight** – failing test inputs often **cluster** in the input space.
- Consider the fault under the condition `x >= 0 && x < 100`.
- We call such clustered reasons **faulty regions**.
- **Without knowing** the faulty regions, what is the **best way** to sample the test inputs?

Adaptive Random Testing (ART)



- A more **diverse** set of test inputs is more likely to find a failure.
- Diversity is depending on the **distance** between test inputs.
- If input data is numeric, we can use the **Euclidean distance**.

$$d((x_1, x_2, \ldots, x_n), (y_1, y_2, \ldots, y_n)) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

- Then, how to measure the distance between **complex data types**?

Levenshtein (Edit) Distance

- The **Levenshtein distance** is a measure of the similarity between two strings.
- It is the minimum number of single-character edits (**insertions**, **deletions**, or **substitutions**) required to change one word into the other.
- For example, the distance between "kitten" and "sitting" is 3:

$$\text{“kitten”} \xrightarrow[k \to s]{\text{substitute}} \text{“sitten”} \xrightarrow[e \to i]{\text{substitute}} \text{“sittin”} \xrightarrow[i]{\text{insert}} \text{“sitting”}$$

- and the distance between "uninformed" and "uniform" is 3:

$$\text{“uninformed”} \xrightarrow[n]{\text{delete}} \text{“uniformed”} \xrightarrow[e]{\text{delete}} \text{“uniformd”} \xrightarrow[d]{\text{delete}} \text{“uniform”}$$

Levenshtein (Edit) Distance

- The formal definition of the **Levenshtein distance** is as follows:

$$lev(a,b) = \begin{cases} |a| & \text{if } |b| = 0 \\ |b| & \text{if } |a| = 0 \\ lev(tail(a), tail(b)) & \text{if } head(a) = head(b) \\ 1 + \min \begin{cases} lev(tail(a), b) & \text{(insert)} \\ lev(a, tail(b)) & \text{(delete)} \\ lev(tail(a), tail(b)) & \text{(substitute)} \end{cases} & \text{otherwise} \end{cases}$$

- It is usually extended into a parameterized version with a set of allowed **edit operations** (e.g., transposition) with different **costs**.
- **Wagner-Fischer algorithm** (1967) – $O(mn)$ time complexity
- Indyk and Bačkurs (2015) proved that the problem of finding the edit distance **cannot be solved in less than quadratic time**. (We cannot do better than the Wagner-Fischer algorithm.)

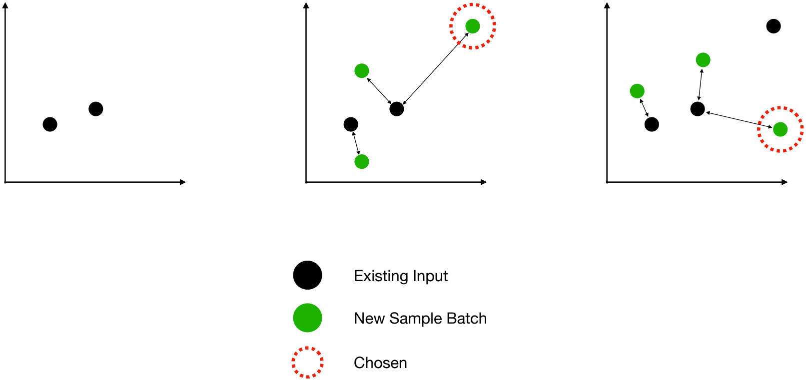
Adaptive Random Testing (ART)

- The **diversity** of a test suite is defined as the **sum of distances** between all pairs of test inputs.

$$diversity(T) = \sum_{(t_1, t_2) \in T \times T} d(t_1, t_2)$$

- We will sample **multiple** Z test inputs and measure the **distance** between **existing** test inputs and the **new** test input.
- Choose the test input that has the **maximum** distance from the existing test inputs.
- Add the **chosen new test input** to the set of existing test inputs.
- Iterate the process until the **stopping criterion** is met.

Adaptive Random Testing (ART)



- It **samples** $Z = 3$ new test inputs and **chooses** the one with the **maximum distance** from the existing test inputs.

Distance Comparison Target

- For each **new test case** t, we need to choose the **target for comparison** in the existing test suite T.[3]
- Minimum-Distance

$$fitness(t, T) = \min_{t' \in T} d(t, t')$$

- Average-Distance

$$fitness(t, T) = \frac{1}{|T|} \sum_{t' \in T} d(t, t')$$

- Maximum-Distance

$$fitness(t, T) = \max_{t' \in T} d(t, t')$$

- Centroid-Distance

$$fitness(t, T) = d(t, 1/|T| \sum_{t' \in T} t')$$

[3] **[CSUR'19]** R. Huang et al. *"A survey on adaptive random testing."*

Complexity

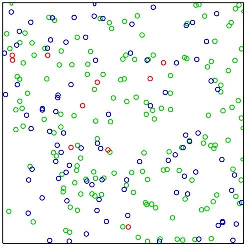
- If we use Z sample points and get ART test suite of k test cases, how many distance calculations do we need?

$$0 + Z + 2Z + 3Z + \cdots + (k-1)Z = \frac{k(k-1)}{2} Z$$

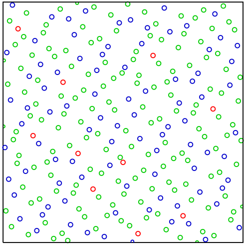
- $\mathbf{O}(k^2 Z)$ **time complexity** – this could be **expensive**.

- It may be difficult to choose the meaningful **distance metric** for complex data types.

Quasi-Random Strategy for ART



$\Rightarrow$



- What if we can randomly sample the test inputs having **diversity** (i.e., **low discrepancy**)?
- **Quasi-random** sequences could be a good choice.
- Let's learn **Halton sequence**, one of the representative quasi-random sequences.

Quasi-Random Strategy for ART – Halton Sequence

- The **halton sequence** is constructed in a **deterministic** way using **co-prime numbers**.
- For example, generate the sequence of numbers in the range $[0, 1]$ by recursively splitting the range into **2** or **3** subintervals.

$$\begin{array}{llll} \frac{1}{2} & & & \\ \frac{1}{4} & \frac{3}{4} & & \\ \frac{1}{8} & \frac{5}{8} & \frac{3}{8} & \frac{7}{8} \\ \frac{1}{16} & \cdots & & \end{array} \qquad \begin{array}{llllll} \frac{1}{3} & \frac{2}{3} & & & & \\ \frac{1}{9} & \frac{4}{9} & \frac{7}{9} & \frac{2}{9} & \frac{5}{9} & \frac{8}{9} \\ \cdots & & & & & \end{array}$$

- Generate a sequence of pairs of numbers (x, y) by combining above sequences.

$$(\frac{1}{2}, \frac{1}{3}), (\frac{1}{4}, \frac{2}{3}), (\frac{3}{4}, \frac{1}{9}), (\frac{1}{8}, \frac{4}{9}), (\frac{5}{8}, \frac{7}{9}), (\frac{3}{8}, \frac{2}{9}), (\frac{7}{8}, \frac{5}{9}), (\frac{1}{16}, \frac{8}{9}), \cdots$$

Quasi-Random Strategy for ART

We can utilize other quasi-random sequences for ART:[4]

- Halton Sequence

$$\phi_b(i) = \sum_{j=0}^{\omega} i_j b^{-j-1}$$

- Sobol Sequence

$$Sobol(i) = XOR_{j=1,2,\cdots,\omega}(i_j \delta_j)$$

where

$$\delta_j = XOR_{k=1,2,\cdots,r}\left(\frac{\beta_k \delta_{j-k}}{2^j}\right) \oplus \frac{\delta_{j-k}}{2^{j+r}}$$

- Niederreiter Sequence

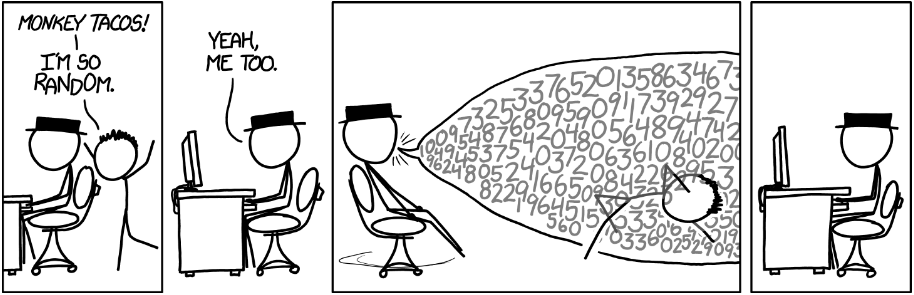
[4] **[CSUR'19]** R. Huang et al. *"A survey on adaptive random testing."*

Adaptive Random Testing (ART) – Summary

- **Application Domains**
  - Numeric Programs
  - Object-Oriented Programs
  - Configurable Systems
  - Web Services and Applications
  - Embedded Systems
  - Simulations and Models
- **Faulty regions** may not apply to all types of faults.
- **ART** is still mostly an academic idea, with debates going on:
  - **[ISSTA'11]** A. Arcuri et al. *"Adaptive random testing: an illusion of effectiveness?"*
  - **[CSUR'19]** R. Huang et al. *"A survey on adaptive random testing."*

Contents

Fuzz Testing



https://xkcd.com/1210/

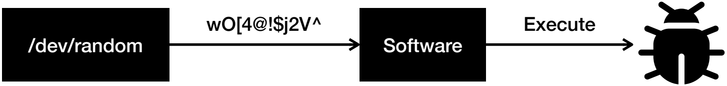
Fuzz Testing

- **[CACM'90]** B. P. Miller et al. *"An empirical study of the reliability of UNIX utilities."*[5]

"On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the **rain** *had affected the phone lines; there were* **frequent spurious characters** *on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these* **spurious characters were causing programs to crash***."*

[5] https://alastairreid.github.io/RelatedWork/papers/miller:cacm:1990/

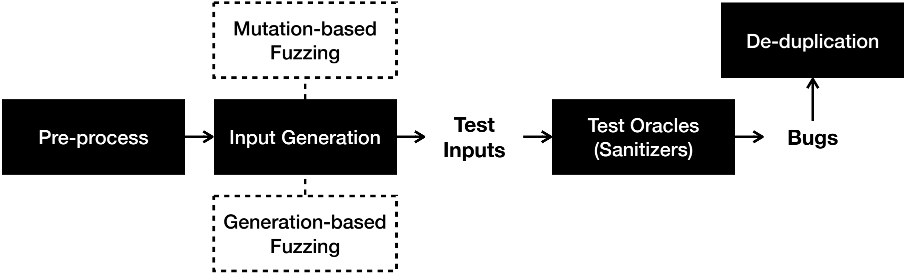
Fuzz Testing



- **Fuzz testing** is a random testing technique to find **exceptional outcomes** (e.g., crashes, exceptions, freezes, etc.) of a software system.

- 1990 study found crashes in: `adb`, `as`, `bc`, `cb`, `col`, `diction`, `emacs`, `eqn`, `ftp`, `indent`, `lex`, `look`, `m4`, `make`, `nroff`, `plot`, `prolog`, `ptx`, `refer!`, `spell`, `style`, `tsort`, `uniq`, `vgrind`, `vi`

Fuzz Testing – Overview

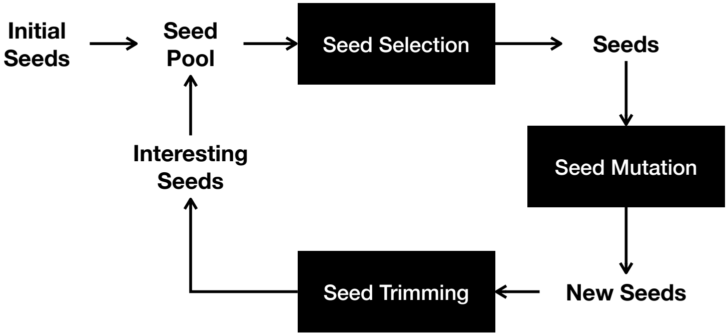


- **Pre-process** – prepare the SUT for fuzz testing
- **Input Generation** – generate test inputs
  - **Mutation-Based Fuzzing** – modify existing test inputs
  - **Generation-Based Fuzzing** – generate new test inputs
- **Test Oracles (Sanitizers)** – detect exceptional outcomes
- **De-duplication** – remove duplicate test inputs

Pre-process

- **Instrumentation** – **source-level** or **binary-level** modification of the SUT to collect information about the execution in compile time (**static**) or runtime (**dynamic**).
  - **Execution Feedback** – collect execution information including **node/branch coverage**.
  - **Thread Scheduling** – **control how threads are scheduled** to to trigger different non-deterministic behaviors.
  - **In-Memory Fuzzing** – take a **memory snapshot** and restore it before writing the new new test case directly into memory and executing it. It can skip over **unnecessary startup costs**.
- **Preparing a Driver Application** – we need to prepare for a driver program when it is difficult to directly fuzz the SUT.
  - **Libraries** – a driver program that calls functions in the library
  - **Kernels** – may fuzz user-land applications to test kernels
  - **IoT devices** – a driver communicate with the corresponding smartphone application.

Input Generation – Mutation-Based Fuzzing



- In the mutation-based fuzzing, a **seed** is a test input that is used to generate new test inputs.
- **Mutation-Based Fuzzing** first initializes **seed pool** with the initial seeds, and then **mutates** them to generate new test inputs and **updates** the seed pool when a new test input is interesting.

Input Generation – Mutation-Based Fuzzing

- **Initial Seeds** – from the **existing test suite**, **manually crafted**, **inferred** from the SUT or specification.
- **Seed Selection** – **random** or **guided** selection (e.g., coverage-based, distance-based, etc.) of the seed from the seed pool.
- **Seed Mutation** – mutate the seed to generate new test inputs.
  - **Bit-Flip** – flip a random bit in the seed
  - **Arithmetic Mutation** – add, subtract, multiply, divide, etc.
  - **Block-based Mutation** – mutate a block of bits
  - **Dictionary-Based Mutation** – replace a value with a predefined value
  - **Semantic-aware Mutation**[6] – mutate seeds using spec. of SUT
- **Seed Trimming** – filter out the **uninteresting** test inputs (e.g., **no coverage** increase).

[6] **[ICSE'21]** J. Park et al. *"JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification."*

Input Generation – Generation-Based Fuzzing

Generation-Based Fuzzing generates new test inputs from a **model** that represents the **input space** of the SUT.

- **Predefined Model** – a model that is **manually crafted**
  - **Simple Specification** – e.g., a range of values, a set of values, etc.
  - **Grammar-Based Model** – inputs are generated from a input grammar
- **Inferred Model** – a model that is **inferred** from previous executions of the SUT or existing test suite.
  - **Probabilistic Grammar**
  - **Call Sequence Model**
  - **Code Bricks**
  - **State Machines**
- **Encoder Model** – generates test inputs for **decoder programs** (e.g., image decoders, audio decoders, etc.) using the corresponding **encoder programs**.

Test Oracles (Sanitizers)

- **Test Oracles (Sanitizers)** – a mechanism to detect **exceptional outcomes** (e.g., crashes, exceptions, freezes, etc.) of the SUT.
    - **ASAN (Address Sanitizer)** – finds **memory corruption** bugs (e.g., buffer overflows, use-after-free, etc.)
    - **MSAN (Memory Sanitizer)** – finds **uninitialized memory** bugs
    - **UBSAN (Undefined Behavior Sanitizer)** – finds **undefined behavior** bugs
    - **CFISAN (Control Flow Integrity Sanitizer)** – finds **control flow integrity** bugs
    - **TSAN (Thread Sanitizer)** – finds thread **race conditions**
    - **LSAN (Leak Sanitizer)** – finds **memory leaks**
- They are usually **instrumented** into the SUT to collect information about the execution in compile time (**static**) or runtime (**dynamic**) with **runtime overhead**.

De-duplication

- **De-duplication** removes **duplicate** test inputs triggering the exceptional outcomes depending on the their **equivalence criteria**.

  - **Stack Backtrace Hashing** – hash the (limited) **stack backtrace** of the exceptional outcome and compare the **hash values**

foo	x
bar	y
g	g
h	h
foo (crashed)	foo (crashed)

    (e.g., both are the same with the stack backtrace hashing with $n = 3$)

  - **Coverage-based De-duplication** – compare the **coverage** of the test inputs (e.g., node, branch, grammar, semantics, etc.)

  - **Semantic-aware De-duplication** – compare the **semantics** of the test inputs (e.g., **backward data-flow analysis** for blaming)

Fuzz Testing

If you are interested in further more details about fuzz testing, please refer to the following resources:

- **[TSE'19]** V. Manès et al. *"The Art, Science, and Engineering of Fuzzing: A Survey"*
- **[CSUR'22]** X. Zhu et al. *"Fuzzing: a survey for roadmap"*
- **The Fuzzing Book** by **Andreas Zeller** et al.

  `https://www.fuzzingbook.org/`

- **AFL++ (American Fuzzy Lop Plus Plus)**

  `https://aflplus.plus/`

- **ClusterFuzz** developed by Google

  `https://google.github.io/clusterfuzz`

Summary

1. Random Testing (RT)
   - Probabilistic Analysis
   - Weaknesses of Random Testing
   - Examples
2. Adaptive Random Testing (ART)
   - Levenshtein (Edit) Distance
   - Distance Comparison Target
   - Complexity of ART
   - Quasi-Random Strategy for ART
3. Fuzz Testing
   - Pre-process
   - Input Generation – Mutation-Based Fuzzing
   - Input Generation – Generation-Based Fuzzing
   - Test Oracles (Sanitizers)
   - De-duplication

Next Lecture

- Coverage Criteria

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr