

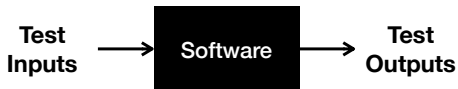
Lecture 1 – Input Space Modeling

AAA705: Software Testing and Quality Assurance

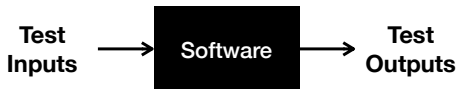
Jihyeok Park



2026 Fall



- **Black-box testing** views the software as a **black box**: test data are derived from the **specification**, not from the code.



- **Black-box testing** views the software as a **black box**: test data are derived from the **specification**, not from the code.
- We already know the input space is far too large to cover. So the real question is **which** inputs to pick – and before we can answer that, we need a **model** of the input space.

1. Modeling by Hand

- Equivalence Partitioning (EP)
- Boundary Value Analysis (BVA)
- Category Partition Method (CPM)

2. Combinatorial Testing (CT)

- Covering Arrays
- Size of Covering Arrays
- Construction and Coverage

3. Inferring the Input Model

1. Modeling by Hand

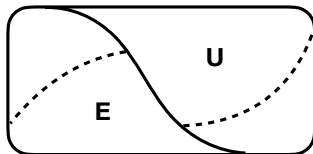
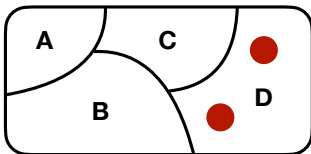
- Equivalence Partitioning (EP)
- Boundary Value Analysis (BVA)
- Category Partition Method (CPM)

2. Combinatorial Testing (CT)

- Covering Arrays
- Size of Covering Arrays
- Construction and Coverage

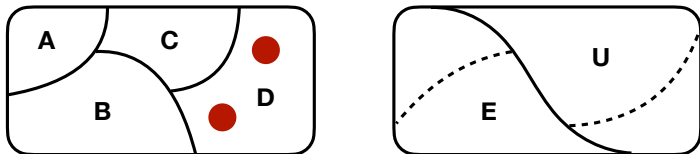
3. Inferring the Input Model

Equivalence partitioning divides the input domain into **equivalence classes**, on the observation that the program should behave the **same way** for all members of a class.



- If one test case in an equivalence class reveals an error, the others in the **same class** likely reveal the **same error**. So we select **one test case** from each class.

Equivalence partitioning divides the input domain into **equivalence classes**, on the observation that the program should behave the **same way** for all members of a class.



- If one test case in an equivalence class reveals an error, the others in the **same class** likely reveal the **same error**. So we select **one test case** from each class.
- Then, how to define the classes? One way is to split the input domain into **expected (E)** (legal) and **unexpected (U)** (illegal) inputs, and then divide the expected inputs further.

Example

Consider a program that takes a **password** as input. The length of the password must be between 6 and 20 characters.

- We can divide the input domain into two equivalence classes:
 - $E = \{ \text{a password } p \mid 6 \leq |p| \leq 20 \}$
 - $U = \{ \text{a password } p \mid |p| < 6 \vee |p| > 20 \}$

Example

Consider a program that takes a **password** as input. The length of the password must be between 6 and 20 characters.

- We can divide the input domain into two equivalence classes:
 - $E = \{ \text{a password } p \mid 6 \leq |p| \leq 20 \}$
 - $U = \{ \text{a password } p \mid |p| < 6 \vee |p| > 20 \}$
- We can divide it more finely:
 - $E_1 = \{ \text{a password } p \mid 6 \leq |p| \leq 10 \}$ for *weak passwords*
 - $E_2 = \{ \text{a password } p \mid 11 \leq |p| \leq 15 \}$ for *medium-strength passwords*
 - $E_3 = \{ \text{a password } p \mid 16 \leq |p| \leq 20 \}$ for *strong passwords*
 - $U_1 = \{ \text{a password } p \mid |p| < 6 \}$ for *too short passwords*
 - $U_2 = \{ \text{a password } p \mid |p| > 20 \}$ for *too long passwords*

Example

Consider a program that takes a **password** as input. The length of the password must be between 6 and 20 characters.

- We can divide the input domain into two equivalence classes:
 - $E = \{ \text{a password } p \mid 6 \leq |p| \leq 20 \}$
 - $U = \{ \text{a password } p \mid |p| < 6 \vee |p| > 20 \}$
- We can divide it more finely:
 - $E_1 = \{ \text{a password } p \mid 6 \leq |p| \leq 10 \}$ for *weak passwords*
 - $E_2 = \{ \text{a password } p \mid 11 \leq |p| \leq 15 \}$ for *medium-strength passwords*
 - $E_3 = \{ \text{a password } p \mid 16 \leq |p| \leq 20 \}$ for *strong passwords*
 - $U_1 = \{ \text{a password } p \mid |p| < 6 \}$ for *too short passwords*
 - $U_2 = \{ \text{a password } p \mid |p| > 20 \}$ for *too long passwords*
- We can select one test case from each equivalence class.

$$I = \{p_1, p_2, p_3, p_4, p_5\}$$

such that $|p_1| = 7$, $|p_2| = 13$, $|p_3| = 18$, $|p_4| = 3$, $|p_5| = 40$

- There are **many ways** to partition the input domain.

¹[TSE'91] E. J. Weyuker, B. Jeng. "*Analyzing Partition Testing Strategies.*"

- There are **many ways** to partition the input domain.
- Even from the same equivalence classes, we can choose **different test cases**.

¹[TSE'91] E. J. Weyuker, B. Jeng. "Analyzing Partition Testing Strategies."

- There are **many ways** to partition the input domain.
- Even from the same equivalence classes, we can choose **different test cases**.
- **Effectiveness** may depend on the tester's **experience** and **intuition**.

¹[TSE'91] E. J. Weyuker, B. Jeng. "Analyzing Partition Testing Strategies."

- There are **many ways** to partition the input domain.
- Even from the same equivalence classes, we can choose **different test cases**.
- **Effectiveness** may depend on the tester's **experience** and **intuition**.

Partition testing can be better, worse, or the same as random testing, depending on how the partitioning is done.¹

¹[TSE'91] E. J. Weyuker, B. Jeng. "Analyzing Partition Testing Strategies."

Logic errors often occur at the **boundaries** of the input domain, usually as **off-by-one** errors from misunderstanding the boundary conditions.

Simple, but **very common**:

```
for (let i = 0; i < 10; i++) {  
    /* body of the loop */  
}
```

CORRECT

```
for (let i = 1; i < 10; i++) {  
    /* body of the loop */  
}
```

INCORRECT

```
for (let i = 0; i <= 10; i++) {  
    /* body of the loop */  
}
```

INCORRECT

```
for (let i = 0; i < 9; i++) {  
    /* body of the loop */  
}
```

INCORRECT

Off-by-one Errors – Fencepost error

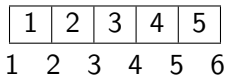
If you build a straight fence 15 meters long with posts spaced 3 meters apart, how many posts do you need?

$$15 / 3 = 5 \text{ posts?}$$

Off-by-one Errors – Fencepost error

If you build a straight fence 15 meters long with posts spaced 3 meters apart, how many posts do you need?

$15 / 3 = 5$ posts? **No, you need 6 posts!**



If you build a straight fence 15 meters long with posts spaced 3 meters apart, how many posts do you need?

$15 / 3 = 5$ posts? **No, you need 6 posts!**

1	2	3	4	5	
1	2	3	4	5	6

`linspace(a, b, n)` in MATLAB is a linear interpolation function that generates a row vector of n **points** instead of n **intervals** between a and b .

Off-by-one Errors – Fencepost error

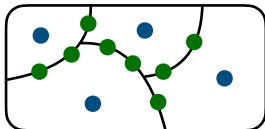
If you build a straight fence 15 meters long with posts spaced 3 meters apart, how many posts do you need?

$15 / 3 = 5$ posts? **No, you need 6 posts!**

1	2	3	4	5	
1	2	3	4	5	6

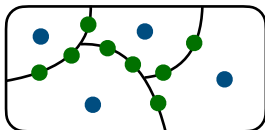
`linspace(a, b, n)` in MATLAB is a linear interpolation function that generates a row vector of n **points** instead of n **intervals** between a and b .

```
linspace(0, 10, 5) == [0, 2.5, 5, 7.5, 10]
                   != [0, 2, 4, 6, 8, 10]
```



- Equivalence Partitioning (EP)
- Boundary Value Analysis (BVA)

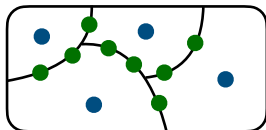
- **Boundary value analysis** selects test cases **at the boundaries** of the equivalence classes, because a program is more likely to fail there. It is used **together with** equivalence partitioning.



- Equivalence Partitioning (EP)
- Boundary Value Analysis (BVA)

- **Boundary value analysis** selects test cases **at the boundaries** of the equivalence classes, because a program is more likely to fail there. It is used **together with** equivalence partitioning.
- For the password classes above (split at lengths 6, 10, 15, 20), the boundaries give

$$\begin{array}{llll} |p_1| = 5 & |p_2| = 6 & |p_3| = 10 & |p_4| = 11 \\ |p_5| = 15 & |p_6| = 16 & |p_7| = 20 & |p_8| = 21 \end{array}$$



- Equivalence Partitioning (EP)
- Boundary Value Analysis (BVA)

- **Boundary value analysis** selects test cases **at the boundaries** of the equivalence classes, because a program is more likely to fail there. It is used **together with** equivalence partitioning.
- For the password classes above (split at lengths 6, 10, 15, 20), the boundaries give

$$\begin{array}{cccc} |p_1| = 5 & |p_2| = 6 & |p_3| = 10 & |p_4| = 11 \\ |p_5| = 15 & |p_6| = 16 & |p_7| = 20 & |p_8| = 21 \end{array}$$

- **both sides** of every boundary, so a comparison written with $<$ instead of $\leq$ cannot survive.

- Most programs behave differently when they receive different **parameters** or are executed under different **environments**.

- Most programs behave differently when they receive different **parameters** or are executed under different **environments**.
- **Category partition method (CPM)** is a black-box testing technique that systematically generates test cases by considering the combinations of the categories of the input domain.
 - ① Analyze **specification**
 - ② Identify **parameters** and **environments**
 - ③ Identify **categories** for each parameter and environment
 - ④ **Partition** categories into equivalence classes
 - ⑤ Identify **constraints**
 - ⑥ **Generate** test cases

Example

Unix command `grep` searches lines matching a pattern:

```
grep <pattern> <filename>
```

`grep park myfile` finds lines containing "park"; `grep "hello world" myfile` needs **quotes** for embedded spaces; `grep " said \"hello \"` myfile needs a **backslash** for an embedded quotation mark.

Example

Unix command `grep` searches lines matching a pattern:

```
grep <pattern> <filename>
```

`grep park myfile` finds lines containing "park"; `grep "hello world" myfile` needs **quotes** for embedded spaces; `grep " said \"hello \" myfile` needs a **backslash** for an embedded quotation mark.

- ① Analyze **specification** ② Identify **parameters** and **environments**
- **Parameters** – (1) <pattern>, (2) <filename>
 - **Environments** – (3) file contents

③ Identify **categories** and ④ **Partition** them into equivalence classes

① **Parameter** – <pattern>

- **Size** – 0 / 1 / ≥ 2
- **Quotation marks** – quoted (Q) / unquoted (U) / improper (I)
- **Embedded spaces** – none (N) / single (S) / multiple (M)
- **Embedded quotation marks** – none (N) / single (S) / multiple (M)

② **Parameter** – <filename>

- **Validity** – exists (E) / not exists (N) / omitted (O)

③ **Environment** – file contents

- **Number of occurrences of the pattern** – 0 / 1 / ≥ 2
- **Number of occurrences of the pattern in a line** – 0 / 1 / ≥ 2

⑤ Identify **constraints** – e.g. an **unquoted** pattern cannot have embedded spaces:

unquoted (U) $\Rightarrow \Leftarrow$ single (S) or multiple (M) **embedded spaces**

- ⑤ Identify **constraints** – e.g. an **unquoted** pattern cannot have embedded spaces:

unquoted (U) $\Rightarrow$ single (S) or multiple (M) **embedded spaces**

- ⑥ **Generate** test cases

- Pick one **test case** from each combination **satisfying** the constraints.

Size	Q	Space	Emb. Q	Valid	Occur	Occur
0	Q	N	N	E	0	0
1	U	S	S	N	1	1
≥ 2	I	M	M	O	≥ 2	≥ 2

```
# myfile has exactly one matching line
$ grep "hello world" myfile
she said hello world once
```

1. Modeling by Hand

Equivalence Partitioning (EP)

Boundary Value Analysis (BVA)

Category Partition Method (CPM)

2. Combinatorial Testing (CT)

Covering Arrays

Size of Covering Arrays

Construction and Coverage

3. Inferring the Input Model

- Testing all combinations is still **too expensive** because of the **combinatorial explosion**!

- Testing all combinations is still **too expensive** because of the **combinatorial explosion!**
 - For example, we need 1,799,736,525 test cases required for the following airport system:

Airline	Destination	Departure Date	Return Date
79	171	365	365

- Testing all combinations is still **too expensive** because of the **combinatorial explosion!**
 - For example, we need 1,799,736,525 test cases required for the following airport system:

Airline	Destination	Departure Date	Return Date
79	171	365	365

- **Combinatorial testing (CT)** or **combinatorial interaction testing (CIT)** constructs test cases by considering the **interactions** between the parameters.

Definition (Interaction)

For k parameters with v values each, a **t -way interaction** is a combination of values for t parameters.

Definition (Interaction)

For k parameters with v values each, a **t -way interaction** is a combination of values for t parameters.

Definition (Covering Array (CA))

A **covering array** $A = CA(N; t, k, v)$ is a $N \times k$ matrix such that every field is an element from the set $[0, v - 1]$, and every t -way interaction is covered at least once by a row of A .

Definition (Interaction)

For k parameters with v values each, a **t -way interaction** is a combination of values for t parameters.

Definition (Covering Array (CA))

A **covering array** $A = CA(N; t, k, v)$ is a $N \times k$ matrix such that every field is an element from the set $[0, v - 1]$, and every t -way interaction is covered at least once by a row of A .

A	B	C
0	0	0
1	1	1



$CA(N = 4; t = 2, k = 3, v = 2)$

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Definition (Mixed Covering Array (MCA))

A **mixed covering array** $A = MCA(N; t, k, v = (v_1, v_2, \dots, v_k))$ is a $N \times k$ matrix such that every field in the i -th column is an element from the set $[0, v_i - 1]$, and every t -way interaction is covered at least once by a row of A .

Definition (Mixed Covering Array (MCA))

A **mixed covering array** $A = MCA(N; t, k, v = (v_1, v_2, \dots, v_k))$ is a $N \times k$ matrix such that every field in the i -th column is an element from the set $[0, v_i - 1]$, and every t -way interaction is covered at least once by a row of A .

$$MCA(N = 6; t = 2, k = 3, v = (2, 2, 3))$$

A	B	C
0	0	0
1	1	1
		2



A	B	C
0	0	0
0	1	1
1	0	2
1	1	0
1	0	1
0	1	2

Definition (Constraint Mixed Covering Array (CMCA))

A **constraint mixed covering array**

$A = CMCA(N; t, k, v = (v_1, v_2, \dots, v_k), P)$ is a $N \times k$ matrix such that every field in the i -th column is an element from the set $[0, v_i - 1]$, and every **valid** t -way interaction is covered at least once by a row of A . We say that a t -way interaction is **valid** if it satisfies the predicate P .

Definition (Constraint Mixed Covering Array (CMCA))

A **constraint mixed covering array**

$A = CMCA(N; t, k, v = (v_1, v_2, \dots, v_k), P)$ is a $N \times k$ matrix such that every field in the i -th column is an element from the set $[0, v_i - 1]$, and every **valid** t -way interaction is covered at least once by a row of A . We say that a t -way interaction is **valid** if it satisfies the predicate P .

A	B	C
0	0	0
1	1	1
		2



$CMCA(N = 5; t = 2, k = 3, v = (2, 2, 3), P)$

A	B	C
0	1	1
1	0	2
1	1	0
1	0	1
0	1	2

$$P(x, y) = x + y > 0$$

Definition (Combinatorial Testing (CT))

Combinatorial testing with strength t builds a test suite from a covering array $CA(N; t, k, v)$: each **row** becomes **one test case**. **Pairwise testing** is the special case $t = 2$.

Definition (Combinatorial Testing (CT))

Combinatorial testing with strength t builds a test suite from a covering array $CA(N; t, k, v)$: each **row** becomes **one test case**. **Pairwise testing** is the special case $t = 2$.

Pairwise testing produces 5 test cases for the following system:

A	B	C
0	0	0
1	1	1
		2



$$CMCA(N = 5; t = 2, k = 3, v = (2, 2, 3), P)$$

A	B	C
0	1	1
1	0	2
1	1	0
1	0	1
0	1	2

$$P(x, y) = x + y > 0$$

Why Does Pairwise Testing Work?

A bug that sat inside `java.util.Arrays.binarySearch` for **nine years**.²

```
int mid = (low + high) / 2;    // int tops out at 2,147,483,647
```

	target in front half	target in back half
small array	ok	ok
huge array ($> 2^{30}$)	ok	crash

Only there do **both** grow large: $\text{low} = 1,073,741,820$,
 $\text{high} = 2,147,483,639$, so

$$\text{low} + \text{high} = 3,221,225,459 \longrightarrow -1,073,741,837 \longrightarrow \text{mid} < 0$$

Either one alone stays in range. It took **both** – which is why nobody hit it for nine years.

²[Google'06] J. Bloch. "Nearly All Binary Searches and Mergesorts Are Broken." (also in *Programming Pearls*, 1986)

Definition (FTFI Number)

The **FTFI number** of a fault is how many parameters must **all** be set to specific values to trigger the failure.

³[TSE'04] D. R. Kuhn, D. R. Wallace, A. M. Gallo. "Software Fault Interactions and Implications for Software Testing."

Definition (FTFI Number)

The **FTFI number** of a fault is how many parameters must **all** be set to specific values to trigger the failure.

binarySearch's fault: **FTFI** = 2 $\implies$ **pairwise finds it**

³[TSE'04] D. R. Kuhn, D. R. Wallace, A. M. Gallo. "Software Fault Interactions and Implications for Software Testing."

Definition (FTFI Number)

The **FTFI number** of a fault is how many parameters must **all** be set to specific values to trigger the failure.

binarySearch's fault: **FTFI** = 2 $\implies$ **pairwise finds it**

Kuhn et al. counted FTFI numbers in **real fault reports**, and the study has since been repeated across many systems.³

³[TSE'04] D. R. Kuhn, D. R. Wallace, A. M. Gallo. "Software Fault Interactions and Implications for Software Testing."

Definition (FTFI Number)

The **FTFI number** of a fault is how many parameters must **all** be set to specific values to trigger the failure.

binarySearch's fault: **FTFI** = 2 $\implies$ **pairwise finds it**

Kuhn et al. counted FTFI numbers in **real fault reports**, and the study has since been repeated across many systems.³

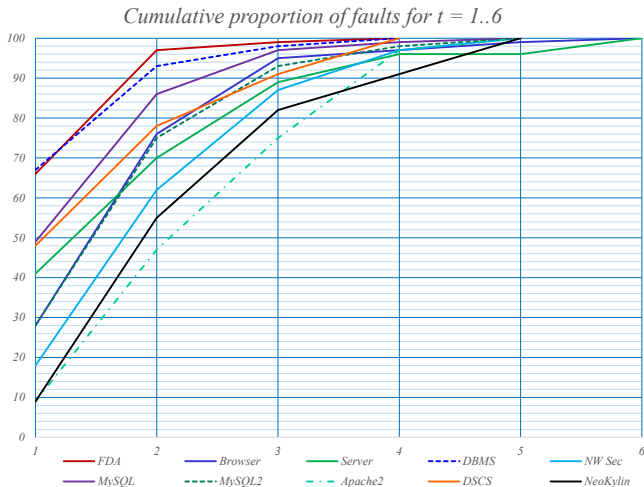
The Interaction Rule. *Almost all faults have a **small FTFI number**, and in every system measured, **none** exceeded 4 ~ 6.*

³[TSE'04] D. R. Kuhn, D. R. Wallace, A. M. Gallo. "Software Fault Interactions and Implications for Software Testing."

Fault Detection Effectiveness

x-axis: strength t ($1 \rightarrow 6$)

y-axis: cumulative % of faults found



“Combinatorial Methods in Software Testing” by Rick Kuhn, NIST, 2019.

Lower Bound – $N \geq v^t$

The interaction rule says a **small** t is enough. But what does covering **all** t -way interactions **cost**? Bound N from below first.

Lower Bound – $N \geq v^t$

The interaction rule says a **small** t is enough. But what does covering **all** t -way interactions **cost**? Bound N from below first.

Pick **any** t of the k columns. Together they have

$$\underbrace{v \times v \times \cdots \times v}_{t \text{ columns}} = v^t \text{ value combinations,}$$

and one **row** shows exactly **one** of them. All v^t must appear:

$$N \geq v^t$$

The interaction rule says a **small** t is enough. But what does covering **all** t -way interactions **cost**? Bound N from below first.

Pick **any** t of the k columns. Together they have

$$\underbrace{v \times v \times \cdots \times v}_{t \text{ columns}} = v^t \text{ value combinations,}$$

and one **row** shows exactly **one** of them. All v^t must appear:

$$N \geq v^t$$

For example, in $CA(N=4; t=2, k=3, v=2)$, any two columns need all $2^2 = 4$ pairs, one row each – and $N = 4$ sits exactly on the floor:

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

$(A, B): (0, 0) (0, 1) (1, 0) (1, 1)$

The Strength t Is What Costs

The bound is v^t , so every $+1$ in strength multiplies the floor by v :

strength t	2	3	4	5	6
v^t for $v = 4$	16	64	256	1,024	4,096

The Strength t Is What Costs

The bound is v^t , so every $+1$ in strength multiplies the floor by v :

strength t	2	3	4	5	6
v^t for $v = 4$	16	64	256	1,024	4,096

- Recall the interaction rule: the **returns diminish** beyond $t = 2 \sim 3$, while this **cost multiplies**.

The Strength t Is What Costs

The bound is v^t , so every $+1$ in strength multiplies the floor by v :

strength t	2	3	4	5	6
v^t for $v = 4$	16	64	256	1,024	4,096

- Recall the interaction rule: the **returns diminish** beyond $t = 2 \sim 3$, while this **cost multiplies**.
- That is why tools and standards settle at $t = 2$ or 3 , and $t = 4 \sim 6$ is reserved for **safety-critical** systems where the last few percent matter.

But the Parameter Count k Barely Matters

Look again at the bound: $N \geq v^t$ has **no** k **in** **it**. Adding parameters never raises the floor.

But the Parameter Count k Barely Matters

Look again at the bound: $N \geq v^t$ has **no k in it**. Adding parameters never raises the floor.

Exact minimum pairwise test counts for **binary** parameters ($t = 2, v = 2$) bear that out:

parameters k	3	4	10	15
exhaustive 2^k	8	16	1,024	32,768
pairwise N	4	5	6	7

But the Parameter Count k Barely Matters

Look again at the bound: $N \geq v^t$ has **no k in it**. Adding parameters never raises the floor.

Exact minimum pairwise test counts for **binary** parameters ($t = 2$, $v = 2$) bear that out:

parameters k	3	4	10	15
exhaustive 2^k	8	16	1,024	32,768
pairwise N	4	5	6	7

- k grows **five-fold**; N grows by **three**, while exhaustive testing grows by a factor of **4,096**.

But the Parameter Count k Barely Matters

Look again at the bound: $N \geq v^t$ has **no k in it**. Adding parameters never raises the floor.

Exact minimum pairwise test counts for **binary** parameters ($t = 2$, $v = 2$) bear that out:

parameters k	3	4	10	15
exhaustive 2^k	8	16	1,024	32,768
pairwise N	4	5	6	7

- k grows **five-fold**; N grows by **three**, while exhaustive testing grows by a factor of **4,096**.
- So it stays far below exhaustive – but **how far**? For that we need an **upper** bound.

Fill an $N \times k$ matrix **at random**. Take $k = 10$ parameters with $v = 3$ values each, and $t = 2$.

Fill an $N \times k$ matrix **at random**. Take $k = 10$ parameters with $v = 3$ values each, and $t = 2$.

Worry about **one** interaction – columns A, D must somewhere show $(0, 1)$:

$$\Pr[\text{a row shows it}] = \underbrace{\frac{1}{3} \times \frac{1}{3}}_{t \text{ columns}} = \frac{1}{9} = \frac{1}{v^t}$$
$$\Pr[\text{no row does}] = \left(\frac{8}{9}\right)^N = \left(1 - \frac{1}{v^t}\right)^N$$

Fill an $N \times k$ matrix **at random**. Take $k = 10$ parameters with $v = 3$ values each, and $t = 2$.

Worry about **one** interaction – columns A, D must somewhere show $(0, 1)$:

$$\Pr[\text{a row shows it}] = \underbrace{\frac{1}{3} \times \frac{1}{3}}_{t \text{ columns}} = \frac{1}{9} = \frac{1}{v^t}$$

$$\Pr[\text{no row does}] = \left(\frac{8}{9}\right)^N = \left(1 - \frac{1}{v^t}\right)^N$$

There are $\binom{10}{2} \times 3^2 = 45 \times 9 = 405$ interactions $\left(= \binom{k}{t} v^t\right)$. Let X count how many end up **uncovered**; adding the 405 chances gives

$$\mathbb{E}[X] = 405 \times \left(\frac{8}{9}\right)^N \quad \left(= \binom{k}{t} v^t \left(1 - \frac{1}{v^t}\right)^N\right)$$

Upper Bound – How Many Rows Suffice?

Raise N until the **average** drops below 1:

N	50	51	52	53
$\mathbb{E}[X]$	1.122	0.997	0.886	0.788
	≥ 1	< 1	< 1	< 1

Upper Bound – How Many Rows Suffice?

Raise N until the **average** drops below 1:

N	50	51	52	53
$\mathbb{E}[X]$	1.122	0.997	0.886	0.788
	≥ 1	< 1	< 1	< 1

The moment $\mathbb{E}[X] < 1$, **not every** matrix can have an uncovered interaction – so at least one has **zero**.

Upper Bound – How Many Rows Suffice?

Raise N until the **average** drops below 1:

N	50	51	52	53
$\mathbb{E}[X]$	1.122	0.997	0.886	0.788
	≥ 1	< 1	< 1	< 1

The moment $\mathbb{E}[X] < 1$, **not every** matrix can have an uncovered interaction – so at least one has **zero**.

$\implies N = 51$ already **guarantees** a covering array exists

Upper Bound – How Many Rows Suffice?

Raise N until the **average** drops below 1:

N	50	51	52	53
$\mathbb{E}[X]$	1.122	0.997	0.886	0.788
	≥ 1	< 1	< 1	< 1

The moment $\mathbb{E}[X] < 1$, **not every** matrix can have an uncovered interaction – so at least one has **zero**.

$\implies N = 51$ already **guarantees** a covering array exists

- At $N = 50$ the argument says **nothing** – not that none exists, only that it cannot decide.

Raise N until the **average** drops below 1:

N	50	51	52	53
$\mathbb{E}[X]$	1.122	0.997	0.886	0.788
	≥ 1	< 1	< 1	< 1

The moment $\mathbb{E}[X] < 1$, **not every** matrix can have an uncovered interaction – so at least one has **zero**.

$\implies N = 51$ already **guarantees** a covering array exists

- At $N = 50$ the argument says **nothing** – not that none exists, only that it cannot decide.
- We proved existence by **averaging**, never by constructing. This is the **probabilistic method**.

$$\mathbb{E}[X] = \binom{k}{t} v^t \left(1 - \frac{1}{v^t}\right)^N \leq k^t v^t e^{-N/v^t}$$

because $\binom{k}{t} \leq k^t$ and $1 - x \leq e^{-x}$ at $x = 1/v^t$. Both make it **larger**, so it still bounds.

$$\mathbb{E}[X] = \binom{k}{t} v^t \left(1 - \frac{1}{v^t}\right)^N \leq k^t v^t e^{-N/v^t}$$

because $\binom{k}{t} \leq k^t$ and $1 - x \leq e^{-x}$ at $x = 1/v^t$. Both make it **larger**, so it still bounds.

$$k^t v^t e^{-N/v^t} < 1 \quad \Longleftrightarrow \quad \boxed{N > t v^t \ln(kv)}$$

$$\text{here } t v^t \ln(kv) = 2 \cdot 9 \cdot \ln 30 \approx 62$$

$$\mathbb{E}[X] = \binom{k}{t} v^t \left(1 - \frac{1}{v^t}\right)^N \leq k^t v^t e^{-N/v^t}$$

because $\binom{k}{t} \leq k^t$ and $1 - x \leq e^{-x}$ at $x = 1/v^t$. Both make it **larger**, so it still bounds.

$$k^t v^t e^{-N/v^t} < 1 \quad \Longleftrightarrow \quad \boxed{N > t v^t \ln(kv)}$$

$$\text{here } t v^t \ln(kv) = 2 \cdot 9 \cdot \ln 30 \approx 62$$

$$\underbrace{16}_{\text{greedy finds}} < \underbrace{51}_{\mathbb{E}[X] < 1} < \underbrace{62}_{\text{after both relaxations}}$$

$$\mathbb{E}[X] = \binom{k}{t} v^t \left(1 - \frac{1}{v^t}\right)^N \leq k^t v^t e^{-N/v^t}$$

because $\binom{k}{t} \leq k^t$ and $1 - x \leq e^{-x}$ at $x = 1/v^t$. Both make it **larger**, so it still bounds.

$$k^t v^t e^{-N/v^t} < 1 \quad \Longleftrightarrow \quad \boxed{N > t v^t \ln(kv)}$$

$$\text{here } t v^t \ln(kv) = 2 \cdot 9 \cdot \ln 30 \approx 62$$

$$\underbrace{16}_{\text{greedy finds}} < \underbrace{51}_{\mathbb{E}[X] < 1} < \underbrace{62}_{\text{after both relaxations}}$$

The argument only proves 51 is **enough** – it never claimed to be tight. Each relaxation then gave away a little more.

$$\underbrace{v^t}_{\text{unavoidable}} \leq N \leq \underbrace{O(t v^t \log(kv))}_{k \text{ enters only here}}$$

$$\underbrace{v^t}_{\text{unavoidable}} \leq N \leq \underbrace{O(t v^t \log(kv))}_{k \text{ enters only here}}$$

For $k = 20$ parameters with $v = 15$ values each:

$$\underbrace{225}_{v^t} \leq \underbrace{441}_{\text{greedy finds}} \leq \underbrace{2,567}_{t v^t \ln(kv)} \quad \left(15^{20} \approx 3.3 \times 10^{23}\right)$$

$$\underbrace{v^t}_{\text{unavoidable}} \leq N \leq \underbrace{O(t v^t \log(kv))}_{k \text{ enters only here}}$$

For $k = 20$ parameters with $v = 15$ values each:

$$\underbrace{225}_{v^t} \leq \underbrace{441}_{\text{greedy finds}} \leq \underbrace{2,567}_{t v^t \ln(kv)} \quad \left(15^{20} \approx 3.3 \times 10^{23}\right)$$

- Add configuration options **almost for free**.
- Raise the strength **never** for free.

Existence is not construction. The bound says a small array is out there; it does not hand us one.

⁴[IEEE Trans. Inf. Theory'88] G. Seroussi, N. H. Bshouty. "Vector Sets for Exhaustive Testing of Logic Circuits."

Existence is not construction. The bound says a small array is out there; it does not hand us one.

- Deciding whether a covering array of a given size exists is **NP-complete**, shown for the binary case by Seroussi and Bshouty.⁴
- So we give up on **optimal** and settle for two practical routes:
 - **Greedy** – fast, any parameters, not optimal. It is **set cover**: keep taking the test that covers the most still-uncovered interactions.
 - **Meta-heuristic** – smaller arrays, much slower.

⁴[IEEE Trans. Inf. Theory'88] G. Seroussi, N. H. Bshouty. "Vector Sets for Exhaustive Testing of Logic Circuits."

- Since finding a **minimum** covering array is NP-complete, we settle for a **polynomial time** greedy algorithm.

⁵[ECBS'07] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, J. Lawrence. "*IPOG: A General Strategy for T-Way Software Testing.*"

- Since finding a **minimum** covering array is NP-complete, we settle for a **polynomial time** greedy algorithm.
- Let's learn the greedy algorithm called **IPOG (In-Parameter-Order-General)**⁵ that generates a covering array with a strength t . It is not optimal but practical.

⁵[ECBS'07] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, J. Lawrence. "IPOG: A General Strategy for T-Way Software Testing."

- ① Initialize **test set** ts to be an empty set.
- ② **Parameters** are $P_1, P_2, \dots, P_k$.
- ③ Add a test into ts for **all interactions of the first t parameters**.
- ④ **for** ($i = t + 1; i \leq n; i++$)
 - ① Let π be the **set of t -way interactions** involving **parameter P_i** and $t - 1$ parameters among the first $i - 1$ parameters.
 - ② (**Horizontal Growth**) **for** (test $\gamma = (v_1, v_2, \dots, v_{i-1}) \in ts$)
 - ① **Choose** a value v_i of P_i
 - ② **Replace** γ with $\gamma' = (v_1, v_2, \dots, v_{i-1}, v_i)$ such that γ' covers the most number of interactions in π .
 - ③ **Remove** from π the interaction covered by γ' .
 - ③ (**Vertical Growth**) **for** (interaction $\alpha \in \pi$)
 - ① **if** ($\exists$ a test covers α) **Remove** α from π .
 - ② **else if** (possible) **Change** an existing test
 - ③ **else Add** a new test to cover α and **Remove** it from π .

- ① Initialize **test set** ts to be an empty set.
- ② **Parameters** are $P_1, P_2, \dots, P_k$.
- ③ Add a test into ts for **all interactions of the first t parameters**.

Example

We want to **pairwise testing** for the following system:

P_1	P_2	P_3	P_4
0	0	0	0
1	1	1	1
		2	2

Adding all combinations of values between the first 2 parameters:

$ts =$	P_1	P_2
	0	0
	0	1
	1	0
	1	1

- ① Initialize **test set** ts to be an empty set.
- ② **Parameters** are $P_1, P_2, \dots, P_k$.
- ③ Add a test into ts for **all interactions of the first t parameters**.
- ④ **for** ($i = t + 1; i \leq n; i++$) (**Horizontal Growth**)
 - ① Let π be the **set of t -way interactions** involving **parameter P_i** and $t - 1$ parameters among the first $i - 1$ parameters.

Set π as pairs to cover involving P_3 :

$$\pi =$$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

- ① Initialize **test set** ts to be an empty set.
- ② **Parameters** are $P_1, P_2, \dots, P_k$.
- ③ Add a test into ts for **all interactions of the first t parameters**.
- ④ **for** ($i = t + 1; i \leq n; i++$) (**Horizontal Growth**)
 - ① Let π be the **set of t -way interactions** involving **parameter P_i** and $t - 1$ parameters among the first $i - 1$ parameters.
 - ② **for** (test $\gamma = (v_1, v_2, \dots, v_{i-1}) \in ts$)
 - ① **Choose** a value v_i of P_i
 - ② **Replace** γ with $\gamma' = (v_1, v_2, \dots, v_{i-1}, v_i)$ such that γ' covers the most number of interactions in π .
 - ③ **Remove** from π the interaction covered by γ' .

Adding values for P_3 in ts :

$$ts =$$

P_1	P_2	P_3
0	0	
0	1	
1	0	
1	1	

$$\pi =$$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Adding values for P_3 in ts :

$$ts =$$

P_1	P_2	P_3
0	0	0
0	1	
1	0	
1	1	

$$\pi =$$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Adding values for P_3 in ts :

$$ts =$$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	
1	1	

$$\pi =$$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Adding values for P_3 in ts :

$$ts =$$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	

$$\pi =$$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Adding values for P_3 in ts :

$$ts =$$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0

$$\pi =$$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

- ① Initialize **test set** ts to be an empty set.
- ② **Parameters** are $P_1, P_2, \dots, P_k$.
- ③ Add a test into ts for **all interactions of the first t parameters**.
- ④ **for** ($i = t + 1; i \leq n; i++$)
 - ① Let π be the **set of t -way interactions** involving **parameter P_i** and $t - 1$ parameters among the first $i - 1$ parameters.
 - ② (**Horizontal Growth**) **for** (test $\gamma = (v_1, v_2, \dots, v_{i-1}) \in ts$)
 - ① **Choose** a value v_i of P_i
 - ② **Replace** γ with $\gamma' = (v_1, v_2, \dots, v_{i-1}, v_i)$ such that γ' covers the most number of interactions in π .
 - ③ **Remove** from π the interaction covered by γ' .
 - ③ (**Vertical Growth**) **for** (interaction $\alpha \in \pi$)
 - ① **if** ($\exists$ a test covers α) **Remove** α from π .
 - ② **else if** (possible) **Change** an existing test
 - ③ **else Add** a new test to cover α and **Remove** it from π .

Extending ts :

$$ts =$$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0

$$\pi =$$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Extending ts :

$$ts =$$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0
0	0	2

$$\pi =$$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Extending ts :

$$ts =$$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0
0	0	2
1	1	2

$$\pi =$$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

The loop **continues** with P_4 . **Horizontal growth** extends all six rows; of the $6 + 6 + 9 = 21$ new interactions, 6 are left over.

$$ts =$$

P_1	P_2	P_3	P_4
0	0	0	0
0	1	1	1
1	0	1	2
1	1	0	0
0	0	2	1
1	1	2	2

$$\pi =$$

P_1	P_2	P_3	P_4
0			2
1			1
		0	1
		0	2
		1	0
		2	0

Vertical growth. A new row fixes only the two positions its interaction names; the rest are **don't cares**.

P_1	P_2	P_3	P_4
0	—	—	2
1	—	—	1
—	—	1	0
—	—	2	0

no new row for two

		0	2
		0	1

fill the don't cares →

P_1	P_2	P_3	P_4
0	0	0	2
1	0	0	1
0	0	1	0
0	0	2	0

Vertical growth. A new row fixes only the two positions its interaction names; the rest are **don't cares**.

P_1	P_2	P_3	P_4
0	—	—	2
1	—	—	1
—	—	1	0
—	—	2	0

no new row for two

		0	2
		0	1

fill the don't cares →

P_1	P_2	P_3	P_4
0	0	0	2
1	0	0	1
0	0	1	0
0	0	2	0

Six interactions, **four** rows. Ten in total, against a lower bound of $3 \times 3 = 9$.

IPOG is an algorithm *for covering arrays*. A **meta-heuristic** is tied to no problem at all: represent the candidates, score them, and it searches. Hence *meta*.

Here that is **simulated annealing**, one of the search methods used in **Search-Based Software Testing (SBST)**.⁶

Size comparison

Subject	Greedy	Meta-heuristic
SPIN-S	27	19
SPIN-V	42	36
GCC	24	21
Apache	42	32
Bugzilla	21	16

Time (sec.) comparison

Subject	Greedy	Meta-heuristic
SPIN-S	0.2	8.6
SPIN-V	11.3	102.1
GCC	204	1902.0
Apache	76.4	109.1
Bugzilla	1.9	9.1

⁶[SSBSE'09] B. J. Garvin, M. B. Cohen, M. B. Dwyer. "An Improved Meta-heuristic Search for Constrained Interaction Testing."

The **reverse** question: for a suite **not** built by combinatorial testing – inherited tests, fuzzer output, a conformance suite – how much t -way coverage does it **already** have? NIST's **CCM** tool measures exactly this.⁷

⁷[ICSTW'13] D. R. Kuhn, R. N. Kacker, Y. Lei. "Combinatorial Coverage Measurement Concepts and Applications."

The **reverse** question: for a suite **not** built by combinatorial testing – inherited tests, fuzzer output, a conformance suite – how much t -way coverage does it **already** have? NIST's **CCM** tool measures exactly this.⁷

total — fraction of **all** t -way interactions covered

simple — fraction of parameter t -subsets covered **completely**

⁷[ICSTW'13] D. R. Kuhn, R. N. Kacker, Y. Lei. "Combinatorial Coverage Measurement Concepts and Applications."

The **reverse** question: for a suite **not** built by combinatorial testing – inherited tests, fuzzer output, a conformance suite – how much t -way coverage does it **already** have? NIST's **CCM** tool measures exactly this.⁷

total – fraction of **all** t -way interactions covered

simple – fraction of parameter t -subsets covered **completely**

Take our $CA(N=4; t=2, k=3, v=2)$ and **drop one row**:

A	B	C
0	0	0
0	1	1
1	0	1

(A, B) : missing (1, 1)

(A, C) : missing (1, 0)

(B, C) : missing (1, 0)

total = $9/12 = 75\%$

simple = $0/3 = 0\%$

⁷[ICSTW'13] D. R. Kuhn, R. N. Kacker, Y. Lei. "Combinatorial Coverage Measurement Concepts and Applications."

The **reverse** question: for a suite **not** built by combinatorial testing – inherited tests, fuzzer output, a conformance suite – how much t -way coverage does it **already** have? NIST's **CCM** tool measures exactly this.⁷

total – fraction of **all** t -way interactions covered

simple – fraction of parameter t -subsets covered **completely**

Take our $CA(N=4; t=2, k=3, v=2)$ and **drop one row**:

A	B	C
0	0	0
0	1	1
1	0	1

(A, B) : missing (1, 1)

(A, C) : missing (1, 0)

(B, C) : missing (1, 0)

total = $9/12 = 75\%$

simple = $0/3 = 0\%$

75% sounds healthy, yet **not one** parameter pair is fully covered – report only the first number and the suite looks far better than it is.

⁷[ICSTW'13] D. R. Kuhn, R. N. Kacker, Y. Lei. "Combinatorial Coverage Measurement Concepts and Applications."

1. Modeling by Hand

Equivalence Partitioning (EP)

Boundary Value Analysis (BVA)

Category Partition Method (CPM)

2. Combinatorial Testing (CT)

Covering Arrays

Size of Covering Arrays

Construction and Coverage

3. Inferring the Input Model

parameters and values $\longrightarrow$ covering array $\longrightarrow$ tests

- But **who wrote** the parameters and values? So far: a **human**, using *EP*, *BVA*, and *CPM*.
- Can we **infer** the model instead?

parameters and values $\longrightarrow$ covering array $\longrightarrow$ tests

- But **who wrote** the parameters and values? So far: a **human**, using *EP*, *BVA*, and *CPM*.
- Can we **infer** the model instead?

inferred from	technique (paper)	what you get (model)
– (written by hand)	<i>EP</i> / <i>BVA</i> / <i>CPM</i>	parameters, values
source code (<code>#ifdef</code>)	Nadi et al. ICSE'14	configuration constraints
documentation	DocTer ISSTA'22	input constraints
examples + an oracle	GLADE PLDI'17 / Arvada ASE'21	a grammar
a seed corpus	Learn&Fuzz ASE'17	a statistical model

`#ifdef X` asks “**was option X selected?**” So the configuration options **are** our parameters, and on/off their values.

```
#ifndef Y
void foo() { ... }
#endif
#ifdef X
void bar() { foo(); }
#endif
```

⁸[ICSE'14] S. Nadi, T. Berger, C. Kästner, K. Czarnecki. “*Mining Configuration Constraints: Static Analyses and Empirical Results.*”

`#ifdef X` asks “**was option X selected?**” So the configuration options **are** our parameters, and on/off their values.

```
#ifndef Y
void foo() { ... }
#endif
#ifdef X
void bar() { foo(); }
#endif
```

- Turn on both and `foo` is **called but never declared** – it does not compile. So every valid configuration satisfies

$$X \rightarrow \neg Y$$

⁸[ICSE'14] S. Nadi, T. Berger, C. Kästner, K. Czarnecki. “Mining Configuration Constraints: Static Analyses and Empirical Results.”

`#ifdef X` asks “**was option X selected?**” So the configuration options **are** our parameters, and on/off their values.

```
#ifndef Y
void foo() { ... }
#endif
#ifdef X
void bar() { foo(); }
#endif
```

- Turn on both and `foo` is **called but never declared** – it does not compile. So every valid configuration satisfies

$$X \rightarrow \neg Y$$

- **Nadi et al.**⁸ reads them out of the code – the **constraints** of a CMCA, written by nobody.

⁸[ICSE'14] S. Nadi, T. Berger, C. Kästner, K. Czarnecki. “*Mining Configuration Constraints: Static Analyses and Empirical Results.*”

An API's documentation already **is** an input model written in English.

```
tf.nn.max_pool3d(input, ksize, strides, padding, ...)
```

input: A **5-D Tensor** of the format specified by data_format.

padding: A **string**, either '**VALID**' or '**SAME**'.

⁹[\[ISSTA'22\]](#) D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, M. W. Godfrey. “DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions.”

An API's documentation already **is** an input model written in English.

```
tf.nn.max_pool3d(input, ksize, strides, padding, ...)
```

input: A **5-D Tensor** of the format specified by data_format.

padding: A **string**, either '**VALID**' or '**SAME**'.

parameter	extracted constraint
input	structure = tensor, ndim = 5
padding	dtype = string, values = {'VALID', 'SAME'}

⁹[ISSTA'22] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, M. W. Godfrey. "DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions."

An API's documentation already **is** an input model written in English.

```
tf.nn.max_pool3d(input, ksize, strides, padding, ...)
```

input: A **5-D Tensor** of the format specified by `data_format`.

padding: A **string**, either '**VALID**' or '**SAME**'.

parameter	extracted constraint
input	structure = tensor, ndim = 5
padding	dtype = string, values = {'VALID', 'SAME'}

The second row is a **parameter with two values** – exactly what *CPM* gave us by hand. **DocTer**⁹ reads it off the prose instead.

⁹[\[ISSTA'22\]](#) D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, M. W. Godfrey. “*DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions.*”

Fully **black-box**. All we have is **one** valid input and an oracle answering “**is this valid?**”:

seed = `<a>hi</a>`

$A \rightarrow (a + \dots + z + \text{<a>A})^*$
what the oracle accepts – **hidden from us**

Fully **black-box**. All we have is **one** valid input and an oracle answering “is this valid?”:

$$\text{seed} = \langle a \rangle \text{hi} \langle /a \rangle \quad \underbrace{A \rightarrow (a + \dots + z + \langle a \rangle A \langle /a \rangle)^*}_{\text{what the oracle accepts – hidden from us}}$$

Guess a generalization, then **ask the oracle** about the strings it newly allows. Keep it only if they all pass.

generalize	candidate	ask the oracle
repeat hi	$(\langle a \rangle (\text{hi})^* \langle /a \rangle)^*$	$\langle a \rangle \langle /a \rangle, \langle a \rangle \text{hihi} \langle /a \rangle$ ✓
split hi	$(\langle a \rangle (\text{h} + \text{i})^* \langle /a \rangle)^*$	$\langle a \rangle \text{h} \langle /a \rangle, \langle a \rangle \text{i} \langle /a \rangle$ ✓

Fully **black-box**. All we have is **one** valid input and an oracle answering “is this valid?”:

$$\text{seed} = \langle a \rangle \text{hi} \langle /a \rangle \quad \underbrace{A \rightarrow (a + \dots + z + \langle a \rangle A \langle /a \rangle)^*}_{\text{what the oracle accepts – hidden from us}}$$

Guess a generalization, then **ask the oracle** about the strings it newly allows. Keep it only if they all pass.

generalize	candidate	ask the oracle
repeat hi	$(\langle a \rangle (\text{hi})^* \langle /a \rangle)^*$	$\langle a \rangle \langle /a \rangle$, $\langle a \rangle \text{hihi} \langle /a \rangle$ ✓
split hi	$(\langle a \rangle (\text{h} + \text{i})^* \langle /a \rangle)^*$	$\langle a \rangle \text{h} \langle /a \rangle$, $\langle a \rangle \text{i} \langle /a \rangle$ ✓

Repeating this until nothing more survives gives a **regular expression**.

That regex puts `<a>...</a>` **next to** each other, but never **inside** one another. Name each `( )`*:

$$A_1 \rightarrow (<a> A_2)^* \qquad A_2 \rightarrow (h + i)^*$$

¹⁰[PLDI'17] O. Bastani, R. Sharma, A. Aiken, P. Liang. "Synthesizing Program Input Grammars."

¹¹[ASE'21] N. Kulkarni, C. Lemieux, K. Sen. "Learning Highly Recursive Input Grammars."

That regex puts `<a>...</a>` **next to** each other, but never **inside** one another. Name each ()^{*}:

$$A_1 \rightarrow (<a> A_2)^* \qquad A_2 \rightarrow (h + i)^*$$

Is A_2 the same as A_1 ? ask `<a><a>hi</a><a>hi</a></a>` ✓

¹⁰[PLDI'17] O. Bastani, R. Sharma, A. Aiken, P. Liang. "Synthesizing Program Input Grammars."

¹¹[ASE'21] N. Kulkarni, C. Lemieux, K. Sen. "Learning Highly Recursive Input Grammars."

That regex puts `<a>...</a>` **next to** each other, but never **inside** one another. Name each ()^{*}:

$$A_1 \rightarrow (<a> A_2)^* \qquad A_2 \rightarrow (h + i)^*$$

Is A_2 the same as A_1 ? ask `<a><a>hi</a><a>hi</a></a>` ✓

Yes – the text slot can hold a whole `<a>...</a>`. One name, one grammar:

$$A \rightarrow (<a> A)^* \mid (h + i)^*$$

¹⁰[PLDI'17] O. Bastani, R. Sharma, A. Aiken, P. Liang. "Synthesizing Program Input Grammars."

¹¹[ASE'21] N. Kulkarni, C. Lemieux, K. Sen. "Learning Highly Recursive Input Grammars."

That regex puts `<a>...</a>` **next to** each other, but never **inside** one another. Name each `( )*`:

$$A_1 \rightarrow (<a> A_2)^* \qquad A_2 \rightarrow (h + i)^*$$

Is A_2 the same as A_1 ? ask `<a><a>hi</a><a>hi</a></a>` ✓

Yes – the text slot can hold a whole `<a>...</a>`. One name, one grammar:

$$A \rightarrow (<a> A)^* \mid (h + i)^*$$

Unbounded nesting, out of one nine-character example. A last pass widens `h+i` to `a+...+z` – and that is exactly the hidden grammar.

¹⁰[PLDI'17] O. Bastani, R. Sharma, A. Aiken, P. Liang. "Synthesizing Program Input Grammars."

¹¹[ASE'21] N. Kulkarni, C. Lemieux, K. Sen. "Learning Highly Recursive Input Grammars."

That regex puts `<a>...</a>` **next to** each other, but never **inside** one another. Name each `( )*`:

$$A_1 \rightarrow (<a> A_2)^* \qquad A_2 \rightarrow (h + i)^*$$

Is A_2 the same as A_1 ? ask `<a><a>hi</a><a>hi</a></a>` ✓

Yes – the text slot can hold a whole `<a>...</a>`. One name, one grammar:

$$A \rightarrow (<a> A)^* \mid (h + i)^*$$

Unbounded nesting, out of one nine-character example. A last pass widens `h+i` to `a+...+z` – and that is exactly the hidden grammar.

GLADE¹⁰ is the above; **Arvada**¹¹ pushes it to deeply recursive grammars.

¹⁰[PLDI'17] O. Bastani, R. Sharma, A. Aiken, P. Liang. "Synthesizing Program Input Grammars."

¹¹[ASE'21] N. Kulkarni, C. Lemieux, K. Sen. "Learning Highly Recursive Input Grammars."

Learn&Fuzz¹²: no oracle, no grammar – just **examples**. 63,000 PDF objects, from a format specified in **1,300 pages**.

```
2 0 obj
<< /Type /Pages /Kids [ 3 0 R ] /Count 1 >>
endobj
```

¹²[\[ASE'17\]](#) P. Godefroid, H. Peleg, R. Singh. “*Learn&Fuzz: Machine Learning for Input Fuzzing.*”

Learn&Fuzz¹²: no oracle, no grammar – just **examples**. 63,000 PDF objects, from a format specified in **1,300 pages**.

```
2 0 obj
<< /Type /Pages /Kids [ 3 0 R ] /Count 1 >>
endobj
```

Train a network to predict the **next character**, then sample from obj until endobj:

$$2\ 0\ \text{obj}\ \ll\ /\text{Ty}\ \longrightarrow\ \text{Pr}[\text{next character}]\ \longrightarrow\ p$$

That is the whole model: **training an RNN on a corpus already is input space modeling** – no rule is ever written down.

¹²[[ASE'17](#)] P. Godefroid, H. Peleg, R. Singh. “*Learn&Fuzz: Machine Learning for Input Fuzzing.*”

Learn&Fuzz¹²: no oracle, no grammar – just **examples**. 63,000 PDF objects, from a format specified in **1,300 pages**.

```
2 0 obj
<< /Type /Pages /Kids [ 3 0 R ] /Count 1 >>
endobj
```

Train a network to predict the **next character**, then sample from obj until endobj:

$$2\ 0\ \text{obj}\ \ll\ /\text{Ty}\ \longrightarrow\ \text{Pr}[\text{next character}]\ \longrightarrow\ p$$

That is the whole model: **training an RNN on a corpus already is input space modeling** – no rule is ever written down.

A perfect model makes a poor fuzzer, so **SampleFuzz** emits the **least** likely character exactly where the model is **most confident**.

¹²[[ASE'17](#)] P. Godefroid, H. Peleg, R. Singh. “*Learn&Fuzz: Machine Learning for Input Fuzzing*.”

1. Modeling by Hand

- Equivalence Partitioning (EP)
- Boundary Value Analysis (BVA)
- Category Partition Method (CPM)

2. Combinatorial Testing (CT)

- Covering Arrays
- Size of Covering Arrays
- Construction and Coverage

3. Inferring the Input Model

- Random Testing and Fuzzing

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