COM3529 Software Testing and Analysis

Mutation Testing

Dr José Miguel Rojas

Downside of Coverage

```
/**
 * Make sure Double.NaN is returned iff n = 0
 */
@Test
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    assertTrue(Double.isNaN(std.getResult()));
    std.increment(1d);
    assertEquals(0d, std.getResult(), 0);
}
```

Downside of Coverage

```
/**
 * Make sure Double.NaN is returned iff n = 0
 */
@Test
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    Double.isNaN(std.getResult());
    std.increment(1d);
    std.getResult();
}
```

Downside of Coverage

```
/**
 * Make sure Double.NaN is returned iff n = 0
 */
@Test
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    Double.isNaN(std.getResult());
    std.increment(1d);
    std.getResult();
}
```

Coverage does not change!

The Oracle Problem

Executing all the code **is not enough**

We need to **check** the functional behaviour

Does this code **actually do what it is meant to do**?

Automated oracle: specification or model

Else, manual oracles have to be defined

How good are my tests?

Coverage = how much of the code is executed

But how much of it is actually **checked**? 🤔

Unfortunately, we don’t know where the bugs are… 😭

But we know the bugs we have made in the past! 😎

Learning from Mistakes

Learning from Mistakes

Key idea: Learning from earlier mistakes to **prevent** them from happening again

Learning from Mistakes

Key idea: Learning from earlier mistakes to **prevent** them from happening again

Key technique: **Simulate** earlier mistakes and see whether the resulting defects are found

Learning from Mistakes

Key idea: Learning from earlier mistakes to **prevent** them from happening again

Key technique: **Simulate** earlier mistakes and see whether the resulting defects are found

Known as **fault-based testing** or **Mutation Testing**

Mutation Coverage

Make many copies of the source code, where **mutations** have been introduced

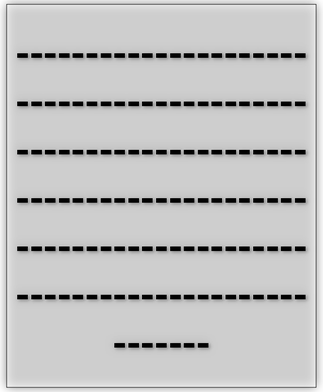
E.g., replacing “**<**” with “**>**”, or “**+**” with “**-**”.

Find the **test (set)** that is able to **kill** the largest number of mutants.

I.e., Executing the test on **both** the original program and the mutated program leads to **different outputs**.

Test 1 Test 2 Test 3

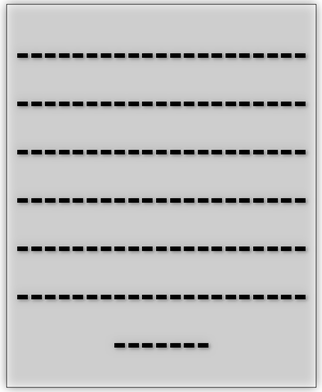
Original Program



Operators

Test 1 | Test 2 | Test 3

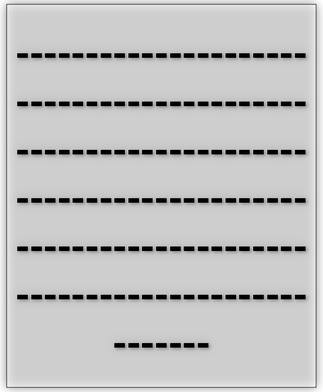
Original Program

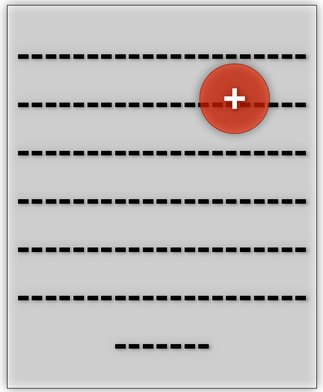


Operators

Test 1 | Test 2 | Test 3

Original Program

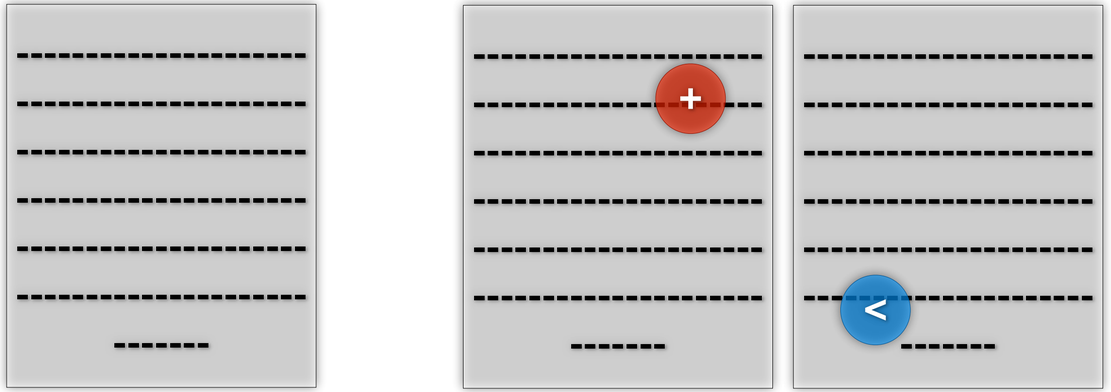




Operators

Test 1 | Test 2 | Test 3

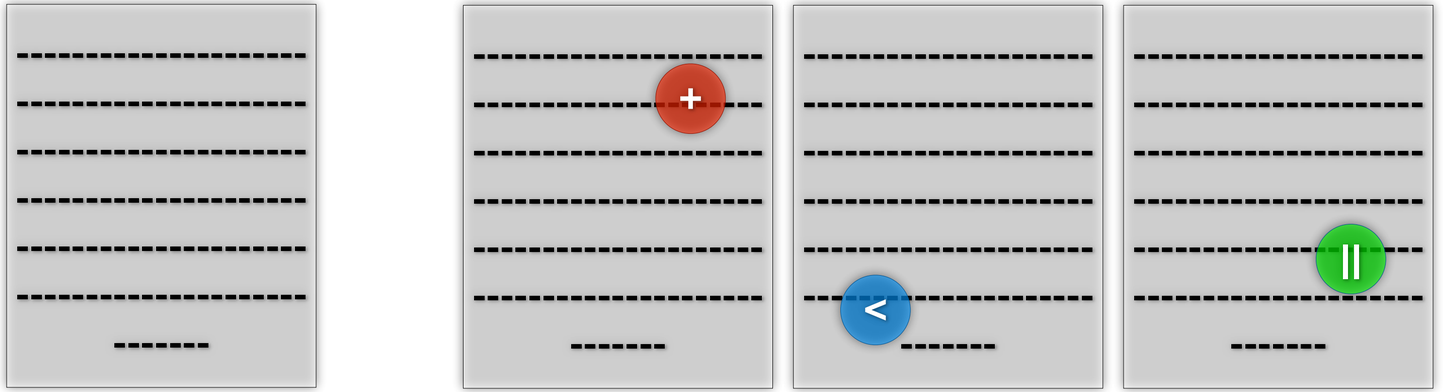
Original Program



Operators

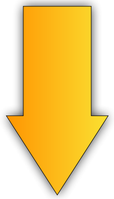
Test 1 | Test 2 | Test 3

Original Program

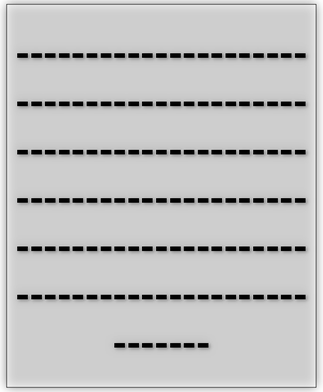


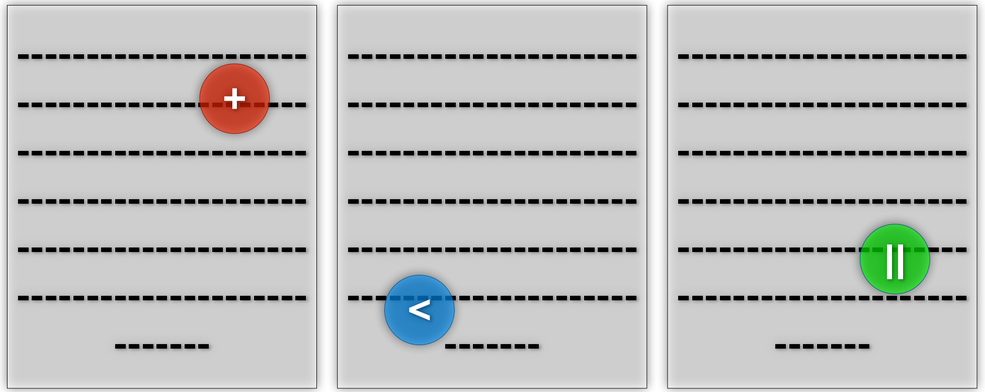
Operators

Test 1 | Test 2 | Test 3



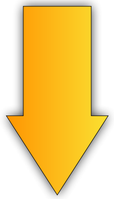
Original Program



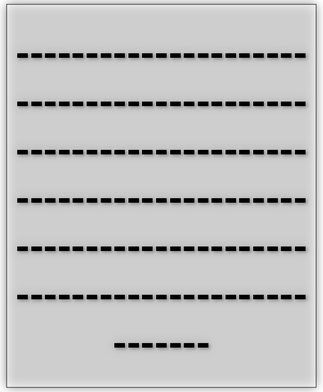


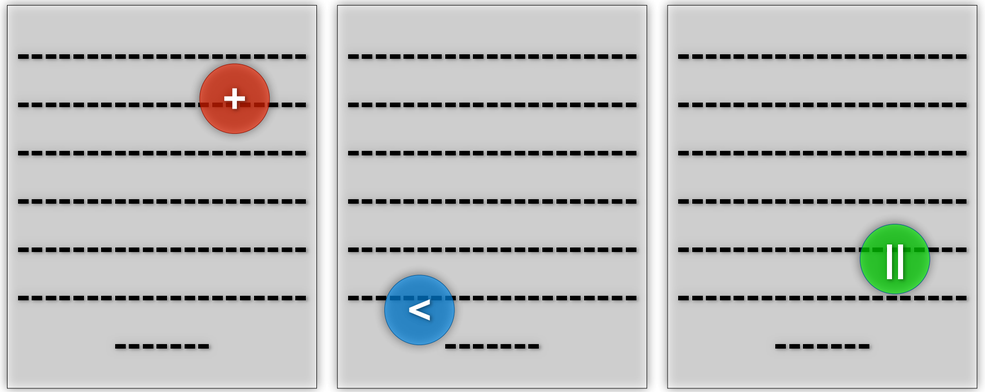
Operators

Test 1 | Test 2 | Test 3



Original Program

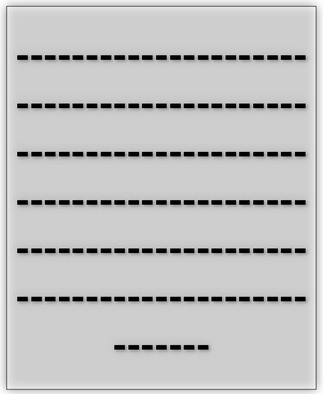


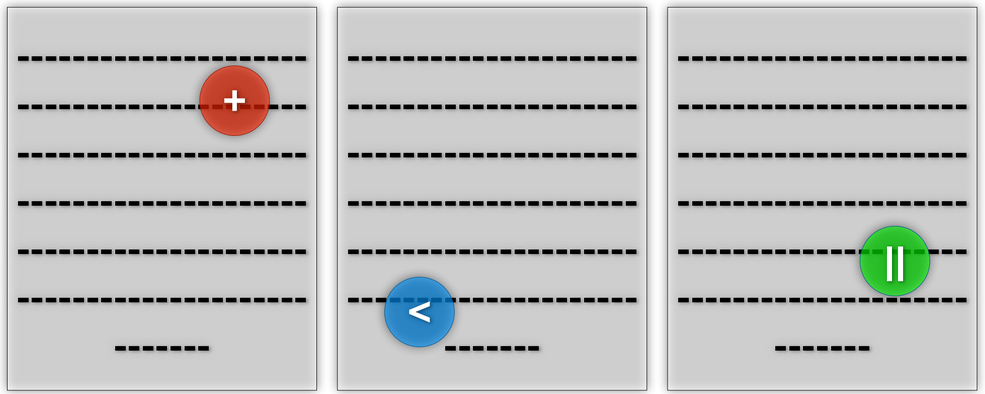


Operators

Test 1 | Test 2 | Test 3

Original Program

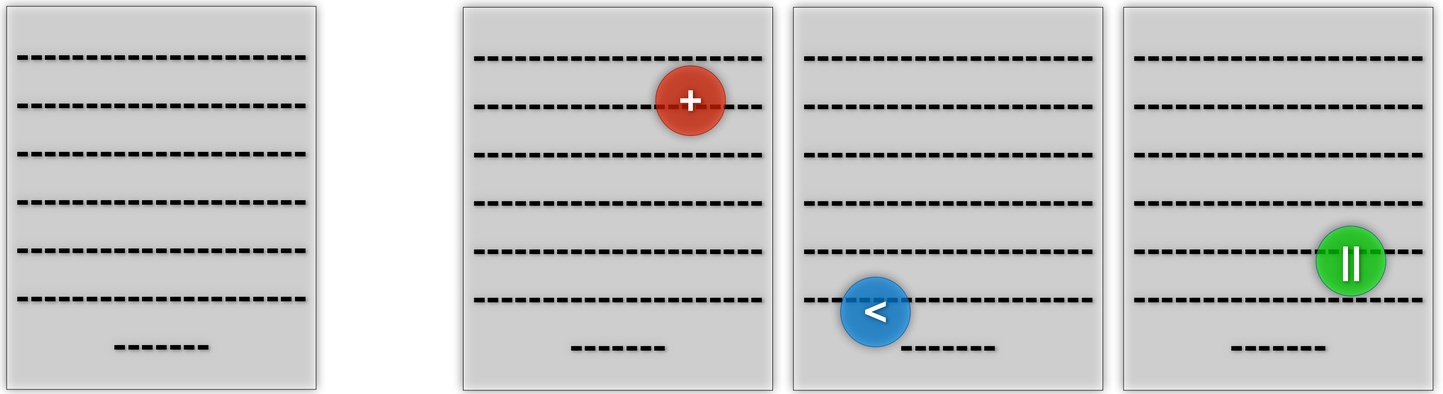




Operators

Test 1 | Test 2 | Test 3

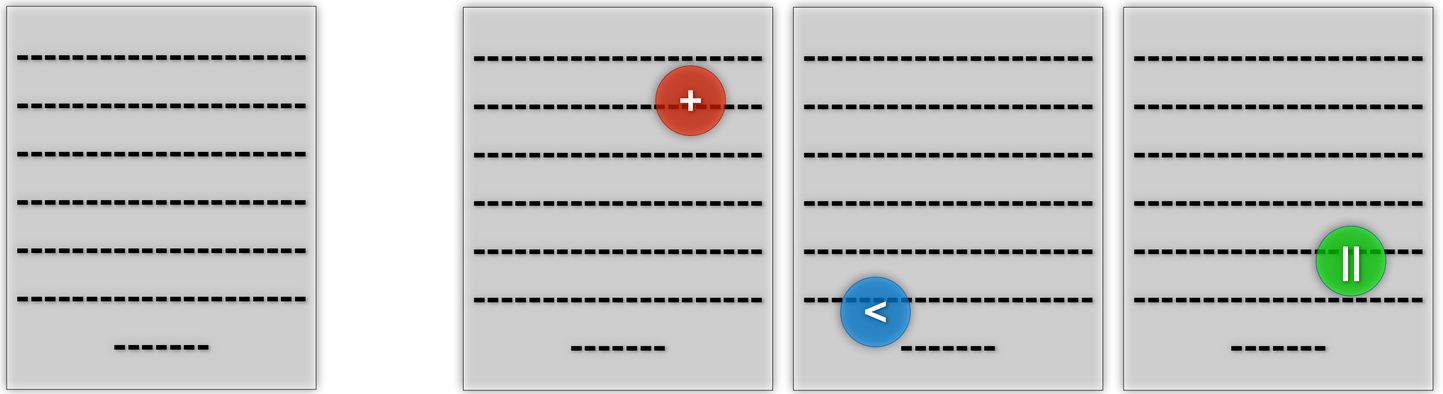
Original Program

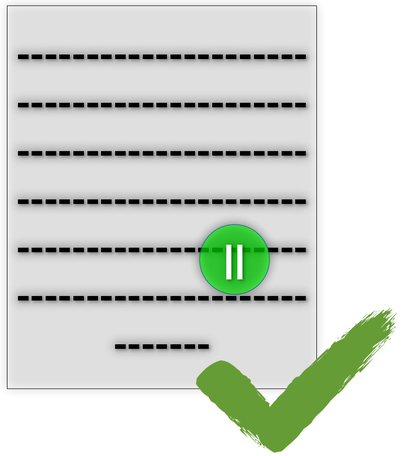


Operators

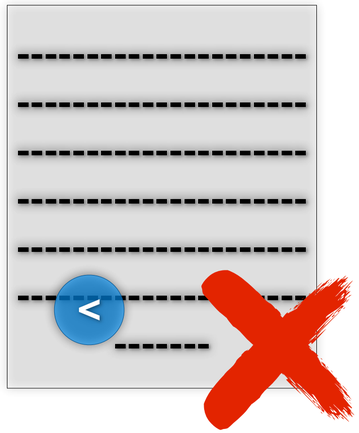
Test 1 | Test 2 | Test 3

Original Program

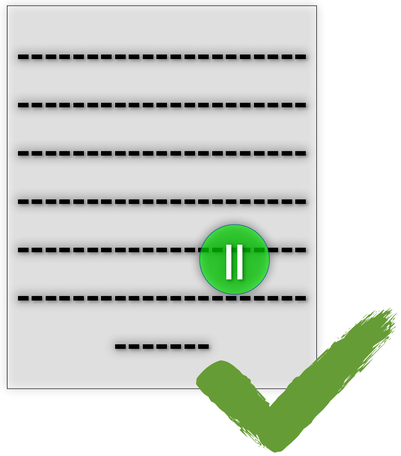


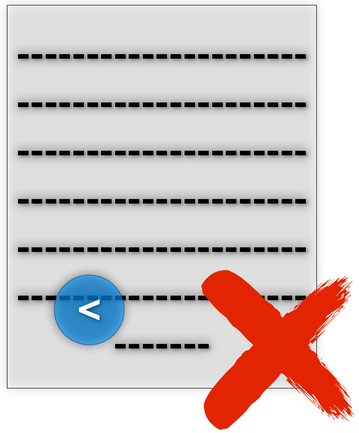


Live mutant - we need more tests



Dead mutant - of no further use





Mutation Score

$$\frac{\text{Killed Mutants}}{\text{Total Mutants}}$$

```
int do_something(int x, int y)
{
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int do_something(int x, int y)
{
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int y)
{
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int y)
{
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int do_something(int x, int y)
{
  if(x < y)
    return x-y;
  else
    return x*y;
}
```

Mutant

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int y)
{
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int y)
{
  if(x < y)
    return x-y;
  else
    return x*y;
}
```

Mutant

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int y)
{
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Program

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

```
int do_something(int x, int y)
{
  if(x < y)
    return x-y;
  else
    return x*y;
}
```

Mutant

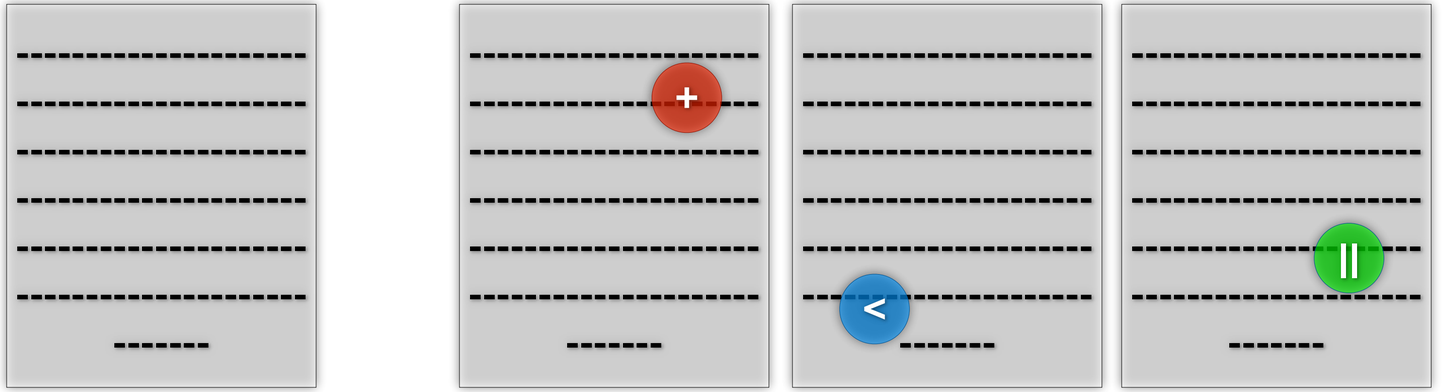
```
int a = do_something(5, 10);
assertEquals(a, 15);
```

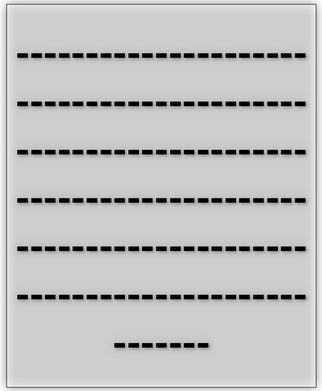
Test

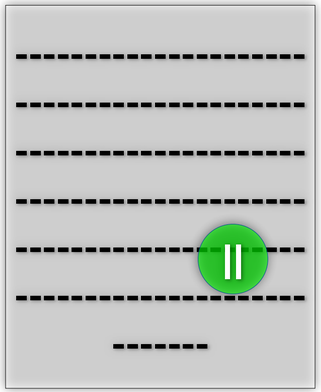
Operators

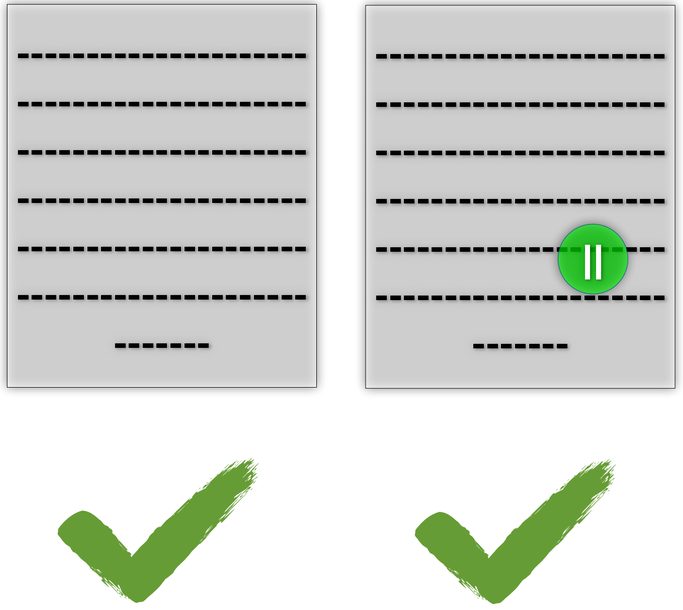
Test 1 | Test 2 | Test 3

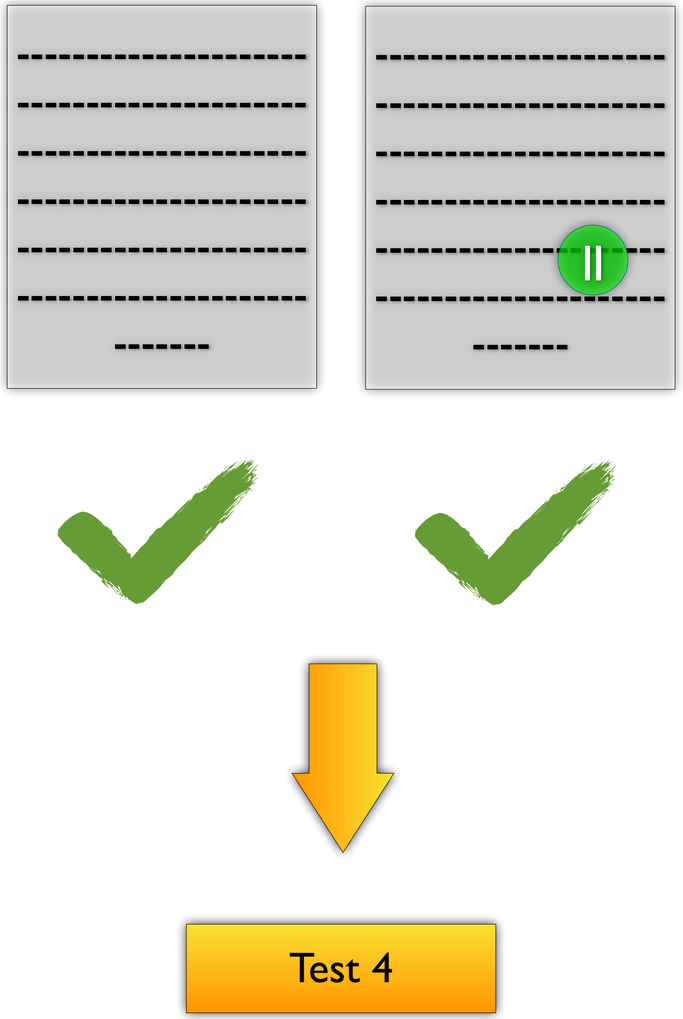
Original Program

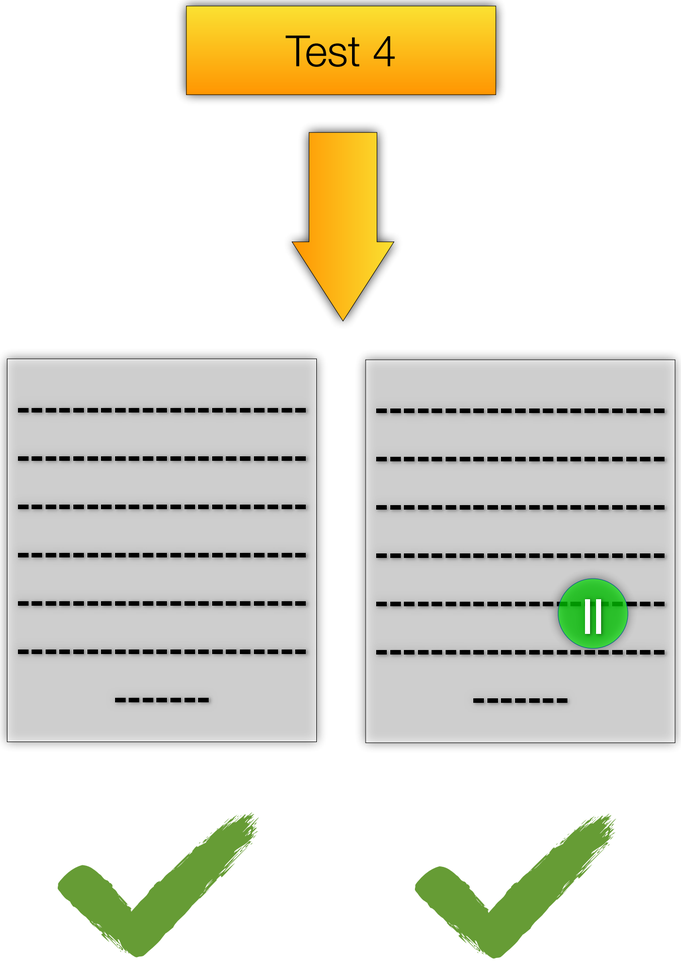


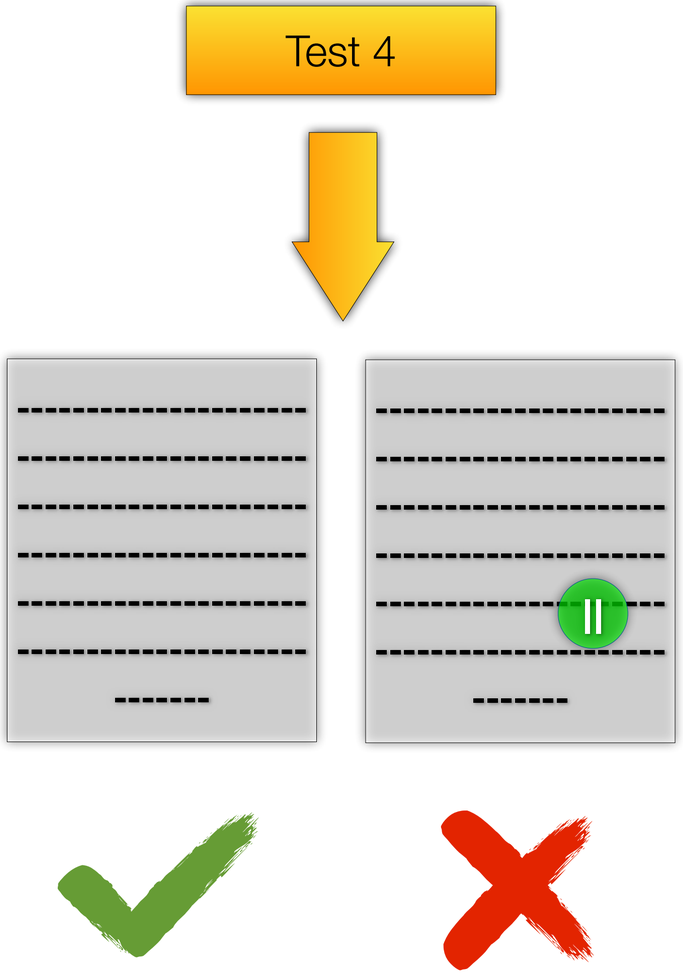







Test 4

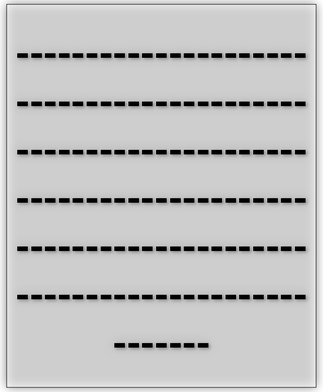

Test 4

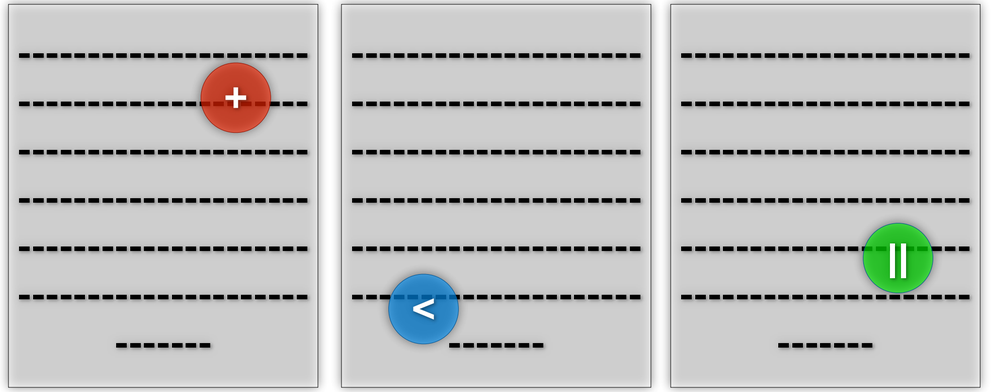

Test 4

Operators

Test 1 | Test 2 | Test 3 | Test 4

Original Program

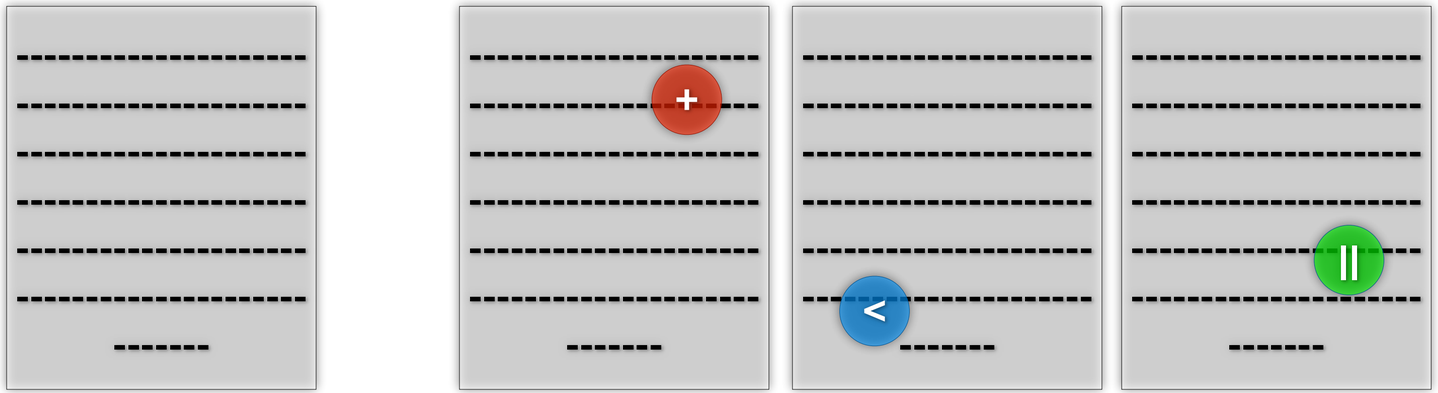




Operators

Test 1 | Test 2 | Test 3 | Test 4

Original Program



Mutants

Slightly changed version of original program

Syntactic change, i.e., valid, compilable code

Simple, resembling a programming glitch

Based on faults from a fault hierarchy

Generating Mutants

Mutation operators
Rule to derive mutants from a program

Mutations based on real faults
Mutation operators represent typical errors

Dedicated mutation operators have been defined for most languages

For example, 100+ operators for the C programming language

Some example operators…

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = abs(y);
    y = tmp;
  }

  return x;
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = abs(y);
    y = tmp;
  }

  return x;
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return -abs(x);
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return -abs(x);
}
```

AOR - Arithmetic Operator

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

AOR - Arithmetic Operator

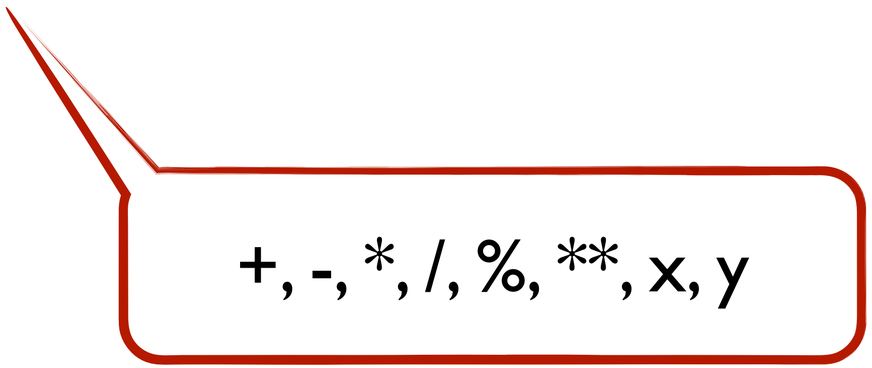
```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```

AOR - Arithmetic Operator

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```

AOR - Arithmetic Operator

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```



ROR - Relational Operator

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

ROR - Relational Operator

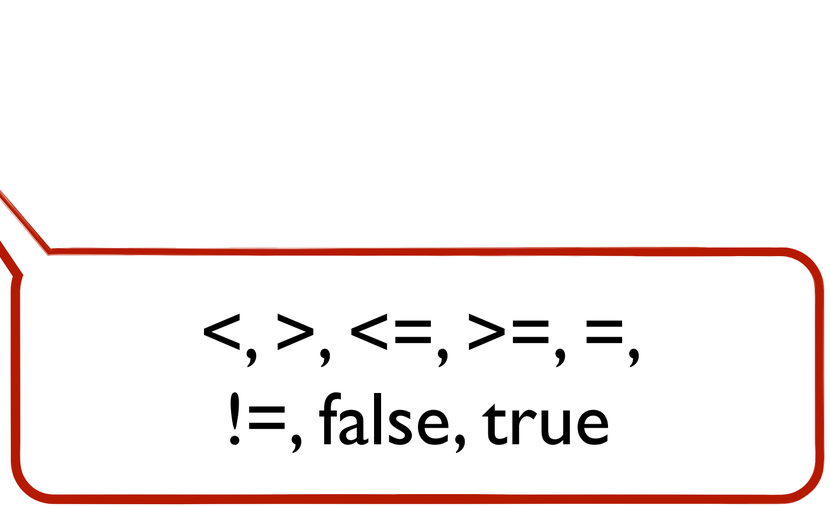
```
int gcd(int x, int y) {
  int tmp;

  while(y > 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

ROR - Relational Operator

```
int gcd(int x, int y) {
  int tmp;

  while(y > 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

ROR - Relational Operator

```
int gcd(int x, int y) {
  int tmp;

  while(y > 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```



COR - Conditional Operator

```
if(a && b)
```

COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
```

COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
if(a & b)
```

COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
```

COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
```

COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
if(false)
```

COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
if(false)
if(true)
```

COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
if(false)
if(true)
if(a)
```

COR - Conditional Operator

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
if(false)
if(true)
if(a)
if(b)
```

UOI - Unary Operator Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

UOI - Unary Operator Insertion

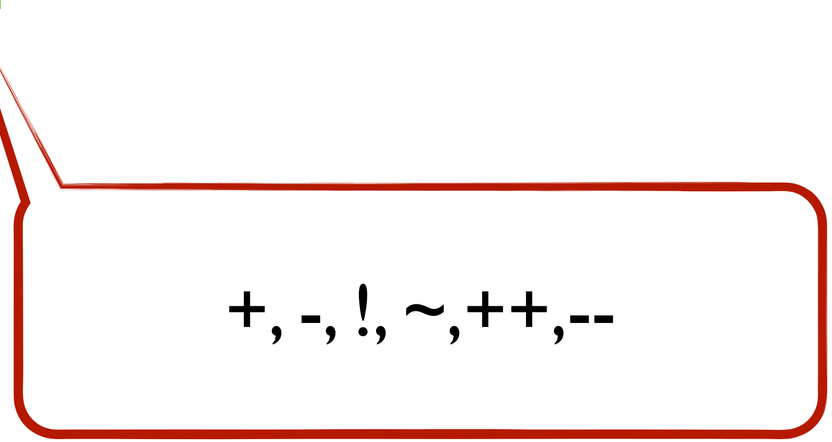
```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % +y;
    x = y;
    y = tmp;
  }

  return x;
}
```

UOI - Unary Operator Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % +y;
    x = y;
    y = tmp;
  }

  return x;
}
```

UOI - Unary Operator Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % +y;
    x = y;
    y = tmp;
  }

  return x;
}
```



SVR - Scalar Variable

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

SVR - Scalar Variable

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = y % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

SVR - Scalar Variable

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = y % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

SVR - Scalar Variable

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = y % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

tmp = x % y
tmp = x % x
tmp = y % y
x = x % y
y = y % x
tmp = tmp % y
tmp = x % tmp

OO Mutation

So far, operators only considered method bodies

Class level elements can be mutated as well:

OO Mutation

So far, operators only considered method bodies

Class level elements can be mutated as well:

```
public class test {
  // ..
  protected void do() {
    // ...
  }
}
```

OO Mutation

So far, operators only considered method bodies

Class level elements can be mutated as well:

```
public class test {
  // ..
  protected void do() {
    // ...
  }
}
```

```
public class test {
  // ..
  private void do() {
    // ...
  }
}
```

OO Mutation

AMC - Access Modifier Change

HVD - Hiding Variable Deletion

HVI - Hiding Variable Insertion

OMD - Overriding Method Deletion

OMM - Overridden Method Moving

OMR - Overridden Method Rename

SKR - Super Keyword Deletion

PCD - Parent Constructor Deletion

ATC - Actual Type Change

Essential Hypotheses

Competent Programmer Hypothesis

Programmers tend to write programs that are in the general neighbourhood of the set of correct programs, i.e., mostly correct

Coupling Effect

Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.

Therefore, mutation testing focuses on **first order mutants**

One mutation at a time; as opposed to Higher Order Mutant (HOM), i.e., mutant of mutant.

$$a + b > c$$

a + b > c

```
a - b > c
a * b > c
a / b > c
a % b > c
a > c
b > c
abs(a) - b > c
a - abs(b) > c
a - b > abs(c)
abs(a - b) > c
-abs(a) - b > c
a - -abs(b) > c
a - b > -abs(c)
-abs(a - b) > c
```

```
a - b >= c
a - b < c
a - b <= c
a - b = c
a - b != c
b - b > c
a - a > c
c - b > c
a - c > c
a - b > a
a - b > b
a - b > c
0 - b > c
a - 0 > c
```

```
a - b > 0
++a - b > c
a - ++b > c
a - b > ++c
--a - b > c
a - --b > c
a - b > --c
++(a - b) > c
--(a - b) > c
-a - b > c
a - -b > c
a - b > -c
-(a - b) > c
0 > c
```

$$a + b > c$$

a + b > c

```
a - b > c
a * b > c
a / b > c
a % b > c
a > c
b > c
abs(a) - b > c
a - abs(b) > c
a - b > abs(c)
abs(a - b) > c
-abs(a) - b > c
a - -abs(b) > c
a - b > -abs(c)
-abs(a - b) > c
```

```
a - b >= c
a - b < c
a - b <= c
a - b = c
a - b != c
b - b > c
a - a > c
c - b > c
a - c > c
a - b > a
a - b > b
a - b > c
0 - b > c
a - 0 > c
```

```
a - b > 0
++a - b > c
a - ++b > c
a - b > ++c
--a - b > c
a - --b > c
a - b > --c
++(a - b) > c
--(a - b) > c
-a - b > c
a - -b > c
a - b > -c
-(a - b) > c
0 > c
```

Performance Problems

Performance Problems

Many mutation operators possible

Proteum - 103 Mutation Operators for C

MuJava - Adds 24 Class level Mutation Operators

Performance Problems

Many mutation operators possible
Proteum - 103 Mutation Operators for C
MuJava - Adds 24 Class level Mutation Operators

Each mutation operator results in **many mutants**
Depending on program under test

Performance Problems

Many mutation operators possible
Proteum - 103 Mutation Operators for C
MuJava - Adds 24 Class level Mutation Operators

Each mutation operator results in **many mutants**
Depending on program under test

Each mutant **needs to be compiled**

Performance Problems

Many mutation operators possible
Proteum - 103 Mutation Operators for C
MuJava - Adds 24 Class level Mutation Operators

Each mutation operator results in **many mutants**
Depending on program under test

Each mutant **needs to be compiled**

Each test case needs to be
executed **against every mutant**

Equivalent Mutants

Mutation = **syntactic** change

The change might leave
the **semantics unchanged**

Equivalent mutants are hard
to detect (undecidable problem)

Might be **reached**, but no **infection**

Might **infect**, but without **propagation**

Example

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[i] > values[r])
      r = i;
  }

  return values[r];
}
```

Example

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 0; i<values.length; i++) {
    if (values[i] > values[r])
      r = i;
  }

  return values[r];
}
```

Example

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 0; i<values.length; i++) {
    if (values[i] > values[r])
      r = i;
  }

  return values[r];
}
```

Example

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 0; i<values.length; i++) {
    if (values[i] > values[r])
      r = i;
  }

  return values[r];
}
```

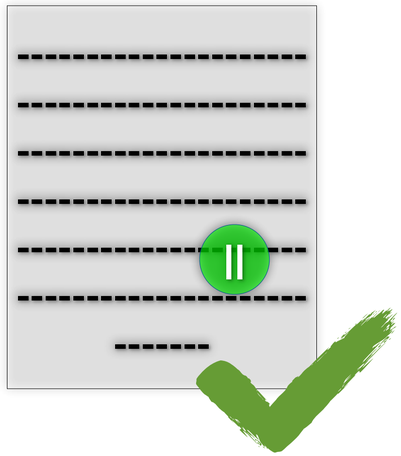
This mutant is equivalent because it only introduces an additional comparison of the first element to itself - **this cannot change the functional behaviour**

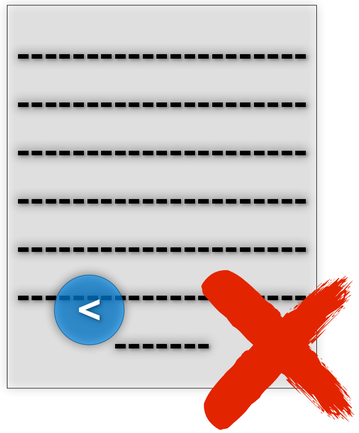
Example

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[i] >= values[r])
      r = i;
  }

  return values[r];
}
```

Example

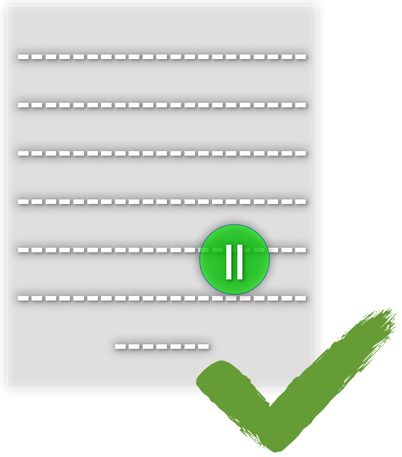
```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[i] >= values[r])
      r = i;
  }

  return values[r];
}
```

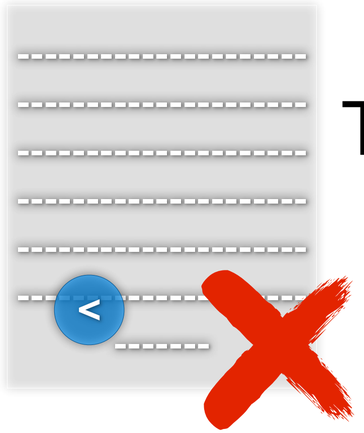




Mutation Score

$$\frac{\text{Killed Mutants}}{\text{Total Mutants}}$$





Mutation Score

$$\frac{\text{Killed Mutants}}{\text{Total Mutants} - \text{Equivalent mutants}}$$

Summary

Mutation Testing uses **mutants** to estimate the **effectiveness** of a test suite

Allows **simulating** other **coverage criteria**

Some **well-known limitations**

- Scalability issues, e.g., high number of mutants
- **Equivalent mutants**

But **increasingly popular** in industry