Software testing techniques

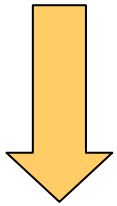
Paolo Tonella
ITC-irst, Centro per la Ricerca Scientifica e Tecnologica
Povo, Trento, Italy
tonella@itc.it

Software correctness

Requirements

```
Given matrix A,
determine its inverse B,
such that B A = A B = I
```

What?



Implementation

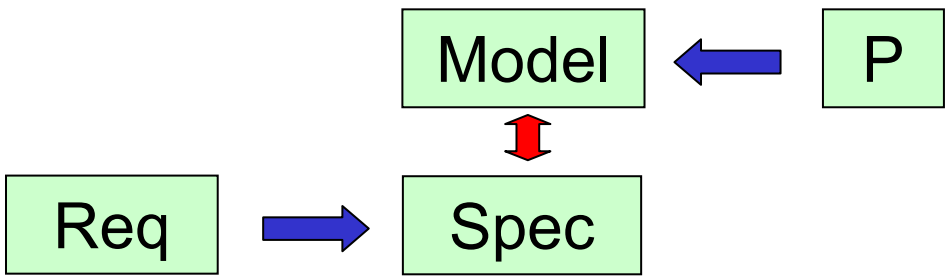
```
class Matrix {
   …
   public Matrix product(Matrix B) {…}
   public Matrix invert() {
      int i;
      for (i = 0; i < repr.size; i++) {
         double tmp = repr[i];
         …
      }
   }
}
```

How?

Does the implementation match the requirements?
Does the program produce the expected output for all inputs?

Correctness proofs

The program (or its model) is shown to be consistent with its formal specifications by means of mathematical arguments.



Support techniques:

- Symbolic execution
- Theorem proving
- Model checking

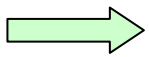
Available technology has a limited scalability
➡ it works on an abstraction of *P*.

1. the semantics (*actual behavior*) of (an abstraction of) *P* is formally defined *and*
2. the specifications (*expected behavior)* of *P* are formally defined:

In general, correctness is undecidable.

Complete testing

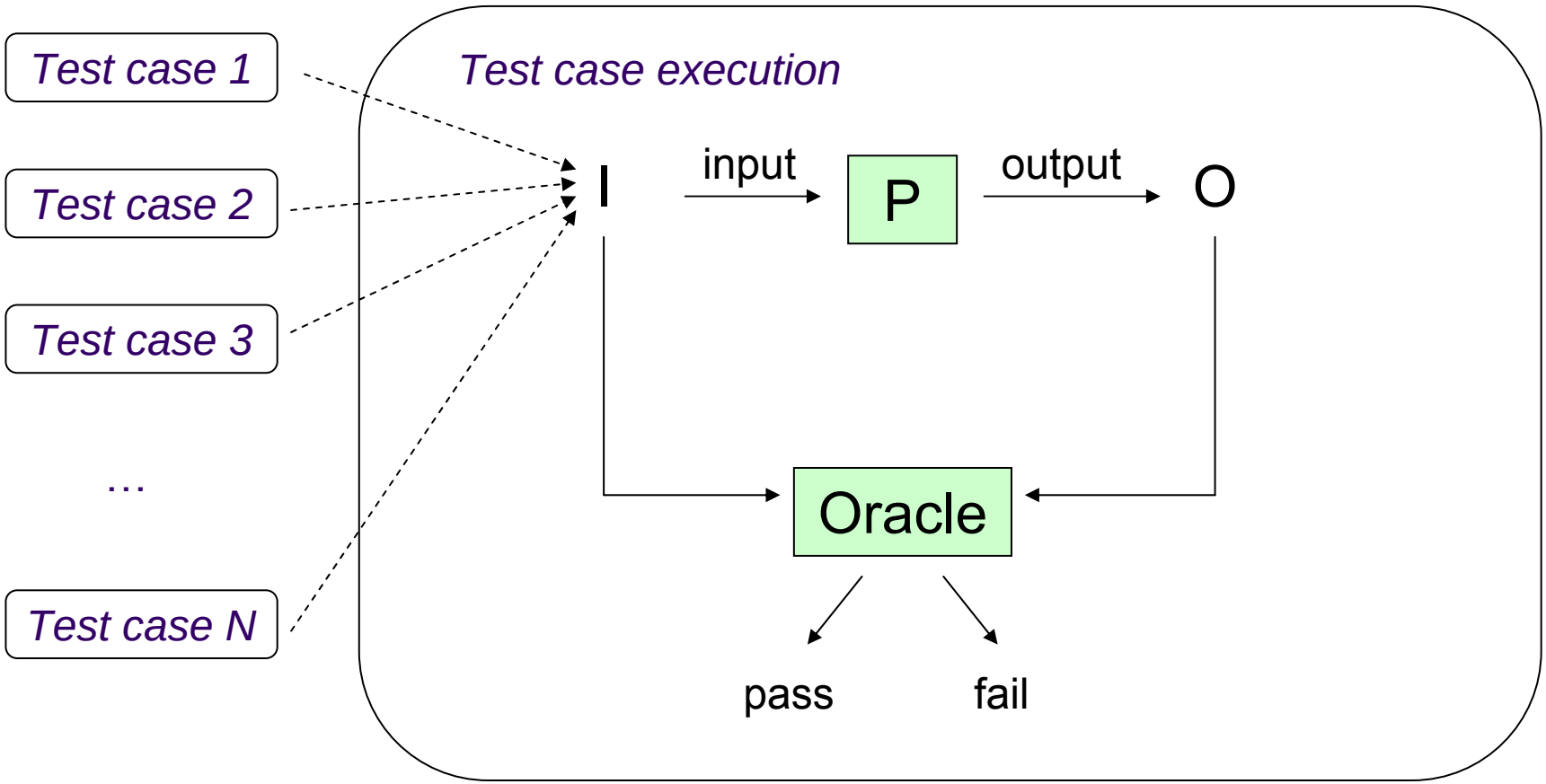
Problem: a program cannot be exercised with all possible input values

`input: char[10]`  10e24 test cases

Solution: input values are properly selected:

- Equivalence classes/boundary values
- Coverage
- Data flow
- Ability to kill mutants
- Statistical ranking
- State transition coverage

Software testing



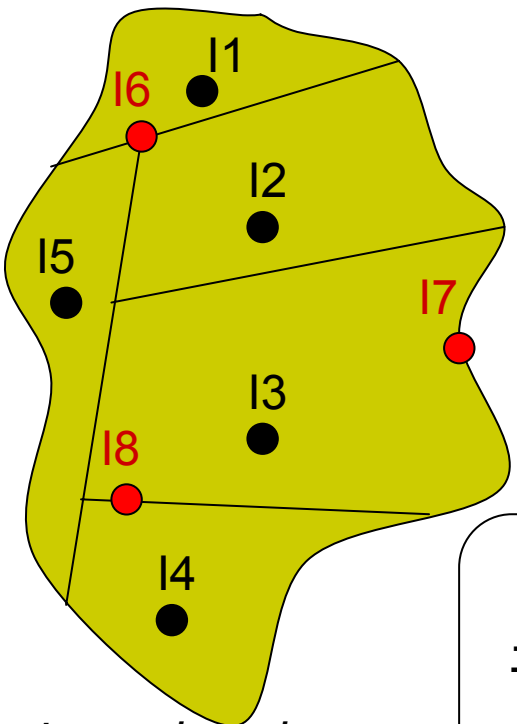
Testing process

The **purpose of testing** is to show that a program has bugs, not that it is bug free.

- *Programmers* build a possibly bug-free system, focusing on construction.
- *Testers* assume that bugs are there to be removed, focusing on the “destructive” side.

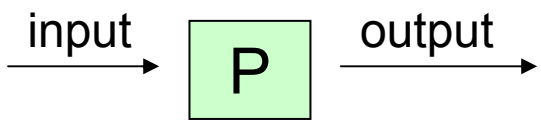
Programmer and tester must be different roles.

Black box testing



Input domain

A program can be delivered only when at least an input for each equivalence class and all boundary values have been tested.



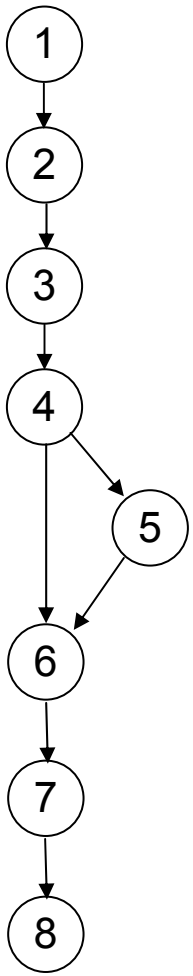

```
input: char[10]
```

```
input = "aa bb cc d"
input = "aaaaa:bbbb"
input = "12/12/2002"
input = ""
input = 0
```

- Equivalence partitioning
- Boundary values

Coverage testing

```
1   program P
2   begin
3     input(x);
4     if (x > 0)
5         x++;
6     end if
7     print(x);
8   end program
```



A program can be delivered only when all statements (branches, paths, etc.) have been traversed in some test case.

1. Statement coverage
2. Branch coverage
3. Condition coverage
4. Path coverage

```
x = 1
```

⇒ Coverage: 1

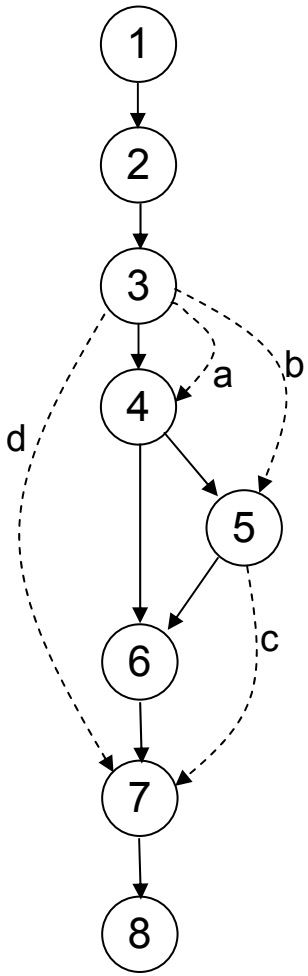
```
x = 0
x = 1
```

⇒ Coverage: 1, 2, 3, 4

Problem: infeasible paths.

Data flow testing

```
1   program P
2   begin
3     input(x);
4     if (x > 0)
5         x++;
6     end if
7     print(x);
8   end program
```



A program can be delivered only when all data flows have been covered in some test case.

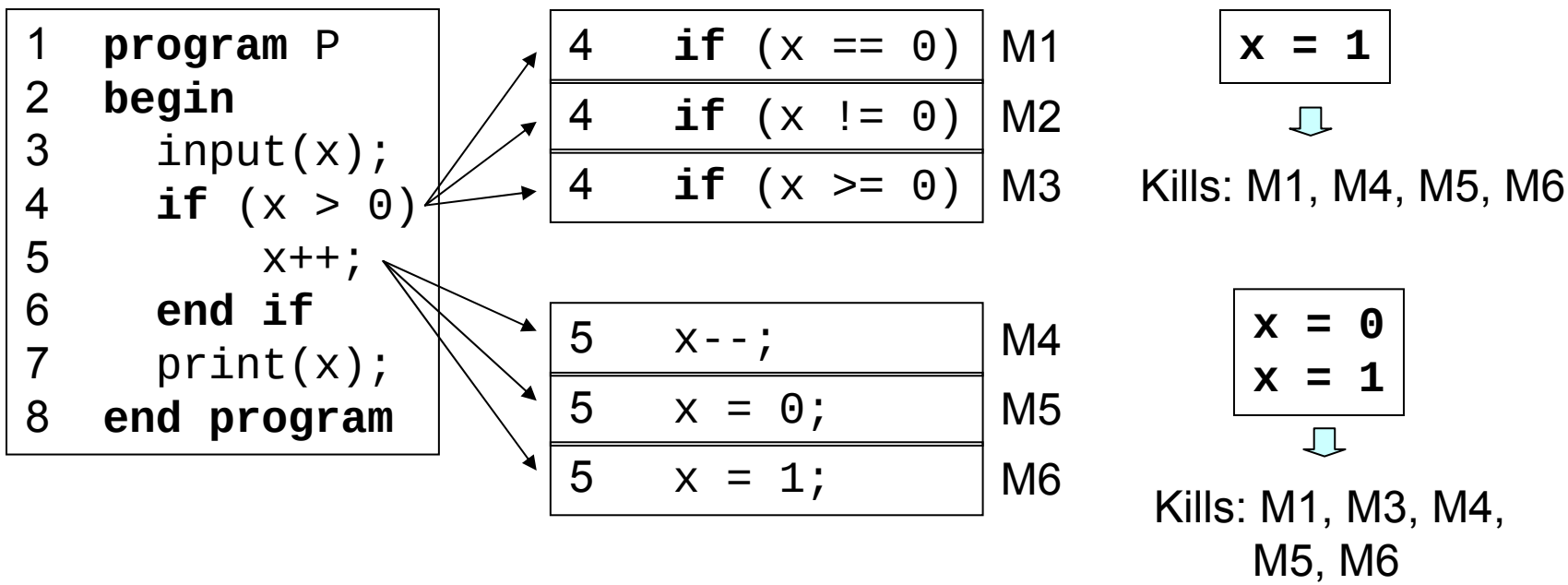
`x = 1` ⇒ Coverage: a, b, c

```
x = 0
X = 1
```
⇒ Coverage: a, b, c, d

Problem: infeasible paths.

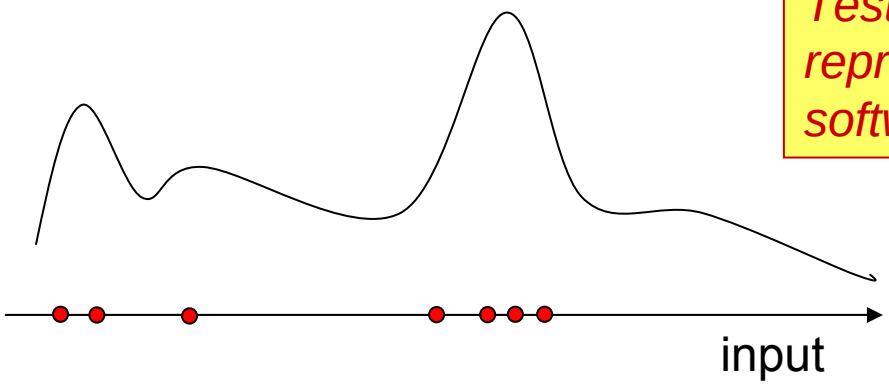
Mutation testing

A program can be delivered only when test cases succeed revealing a set of known bugs (killing all mutants).



Both test suites are not satisfactory because they cannot reveal the defect in **M2**.

Statistical testing

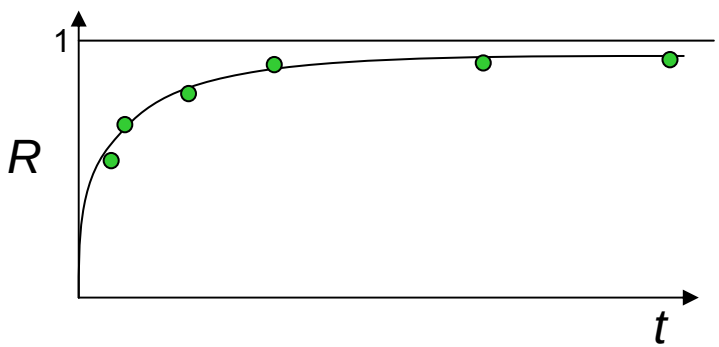


Test cases are generated so as to reproduce the actual usage of the software.

Usage model: probability distribution of input values.

A measure of software reliability can be derived from the number of failures observed (*f*):

$$R = 1 - \frac{f}{n} \qquad MTBF = \frac{n}{f}$$



A *reliability growth model* can be estimated over time:

$$R = 1 - e^{-bt}$$

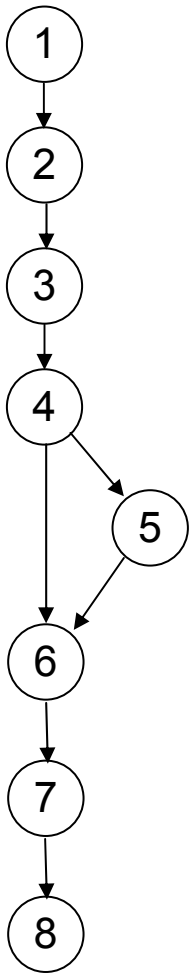
Regression testing

A changed program can be delivered only when all existing test cases are passed.

- **Test case selection**: given the changed statements, the subset of test cases that need be re-executed is determined.
- **Test suite minimization**: a minimal subset of test cases is determined maintaining the same coverage level.
- **Test case prioritization**: test cases are ranked (for example, by coverage, or by rate of fault detection).

Coverage testing

```
1   program P
2   begin
3     input(x);
4     if (x > 0)
5         x++;
6     end if
7     print(x);
8   end program
```

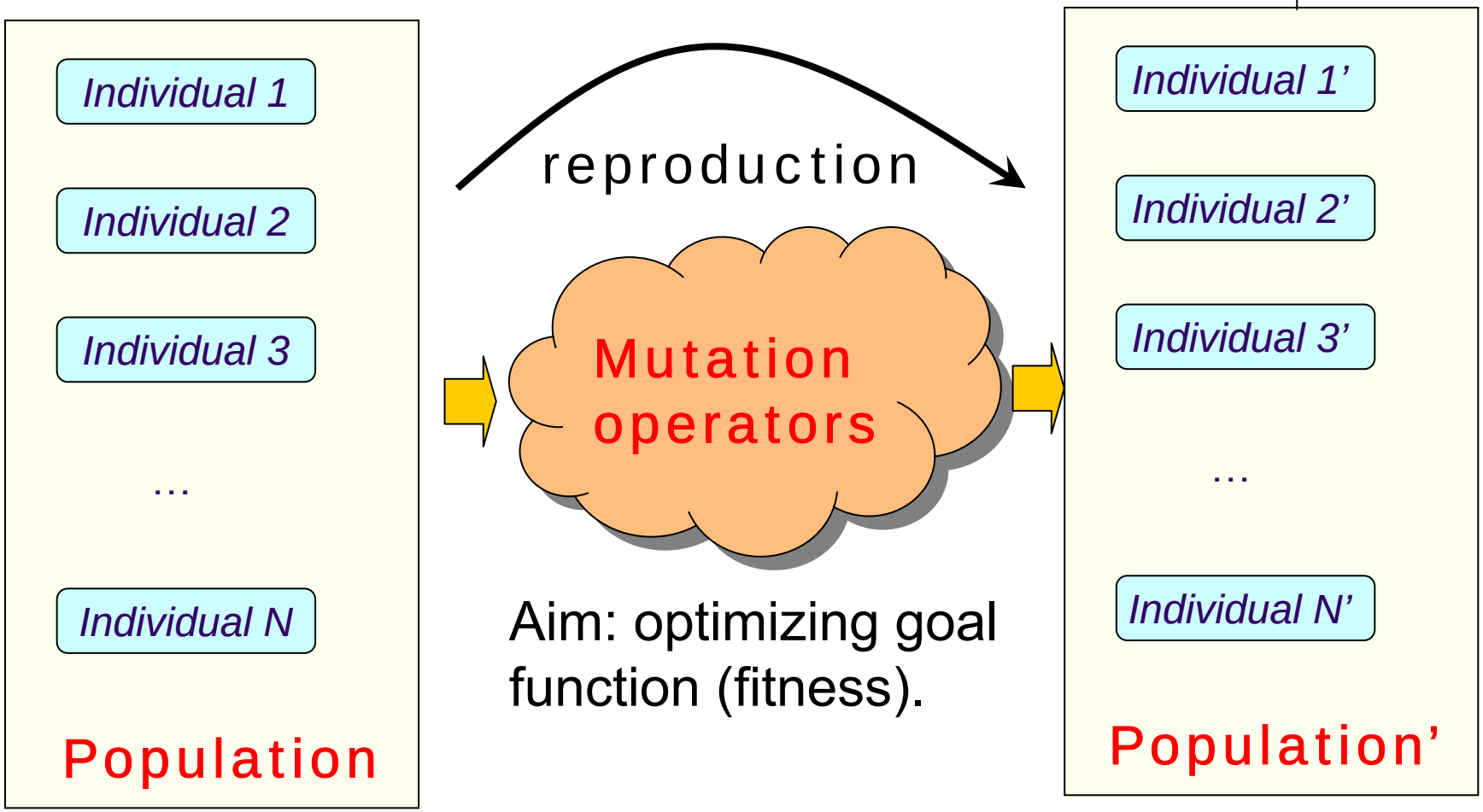


***Problem**: which input values allow reaching a given level of coverage?*

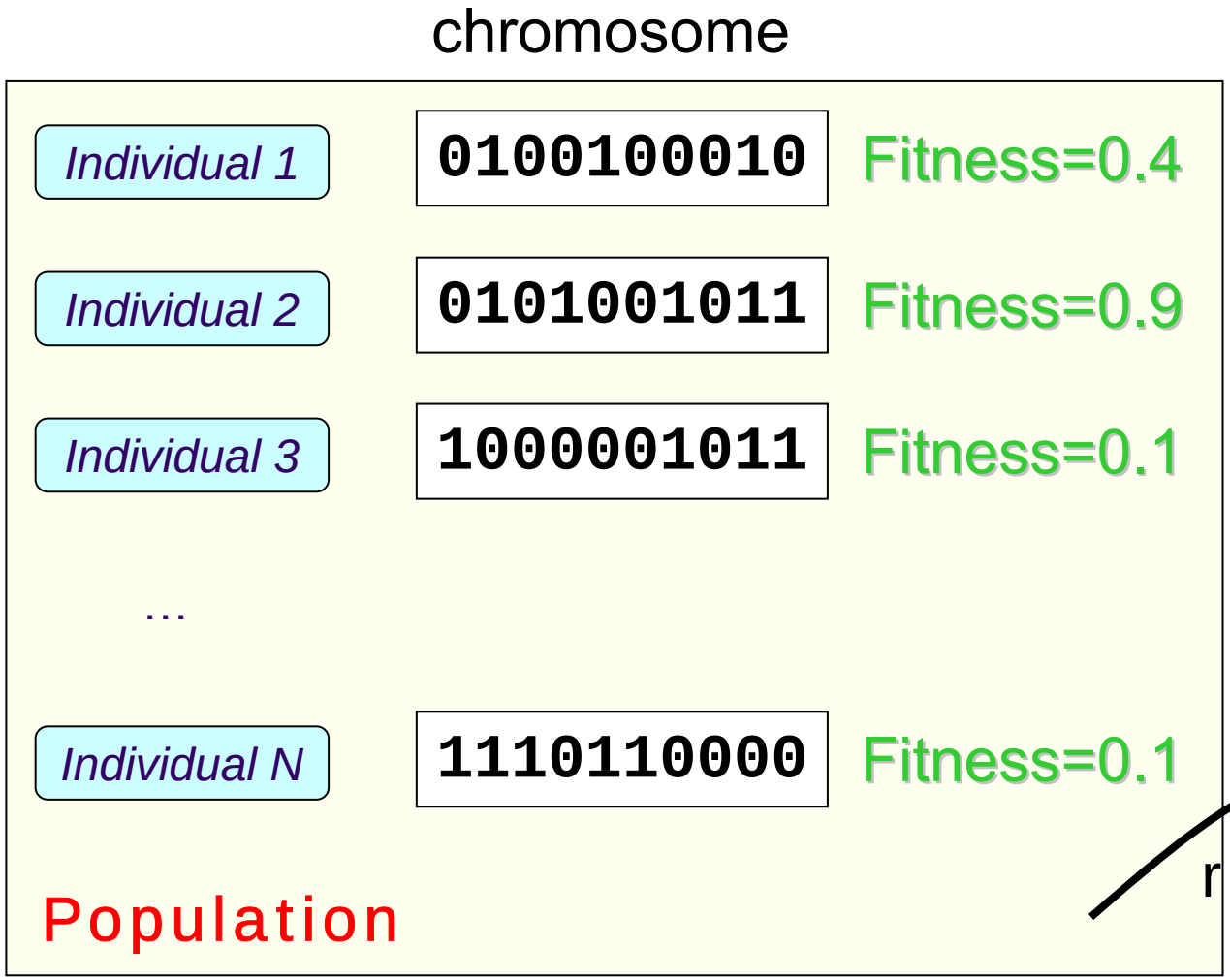
Approaches:

- manual selection
- symbolic execution
- genetic algorithms

Genetic algorithms



Fitness



Probability of reproduction is proportional to fitness.

reproduction

Fitness measures distance of behavior from goal.

Example program

```
1   class Triangle {
2       int a, b, c;
3       public void checkRightAngle() {
4          if (a*a + b*b == c*c)
5             kind = RIGHT_ANGLE;
6          else if (b*b + c*c == a*a)
7             kind = RIGHT_ANGLE;
8          else if (a*a + c*c == b*b)
9             kind = RIGHT_ANGLE;
10         else kind = TRIANGLE;
11      }
12      public void computeTriangleKind() {
13         if (a == b)
14           if (b == c) kind = EQUILATERAL;
15           else kind = ISOSCELE;
16         else if (a == c) kind = ISOSCELE;
17         else if (b == c) kind = ISOSCELE;
18         else checkRightAngle();
19      }
20  }
```

Note: check on triples that are not sides of triangle (method **`isTriangle`**) was omitted

Test cases

(a, b, c)

(2, 2, 2)
(2, 2, 1)
(2, 1, 2)
(3, 4, 2)
(4, 3, 3)
(4, 5, 6)
(3, 5, 2)
(3, 2, 2)

Population

```
1   class Triangle {
2       int a, b, c;
3       public void checkRightAngle() {
4           if (a*a + b*b == c*c)
5               kind = RIGHT_ANGLE;
6           else if (b*b + c*c == a*a)
7               kind = RIGHT_ANGLE;
8           else if (a*a + c*c == b*b)
9               kind = RIGHT_ANGLE;
10          else kind = TRIANGLE;
11      }
12      public void computeTriangleKind() {
13          if (a == b)
14            if (b == c) kind = EQUILATERAL;
15            else kind = ISOSCELE;
16          else if (a == c) kind = ISOSCELE;
17          else if (b == c) kind = ISOSCELE;
18          else checkRightAngle();
19      }
20  }
```

Note: check on triples that are not sides of triangle (method **`isTriangle`**) was omitted

Test case execution

(a, b, c)	Trace
(2, 2, 2)	<13, 14>
(2, 2, 1)	<13, (14), 15>
(2, 1, 2)	<13, 16>
(3, 4, 2)	<13, (16), (17), 18, 4, 6, 8, 10>
(4, 3, 3)	<13, 16, 17>
(4, 5, 6)	<13, 16, 17, 18, 4, 6, 8, 10>
(3, 5, 2)	<>
(3, 2, 2)	<13, (16), 17>

```
1   class Triangle {
2       int a, b, c;
3       public void checkRightAngle() {
4           if (a*a + b*b == c*c)
5               kind = RIGHT_ANGLE;
6           else if (b*b + c*c == a*a)
7               kind = RIGHT_ANGLE;
8           else if (a*a + c*c == b*b)
9               kind = RIGHT_ANGLE;
10          else kind = TRIANGLE;
11      }
12      public void computeTriangleKind() {
13          if (a == b)
14            if (b == c) kind = EQUILATERAL;
15            else kind = ISOSCELE;
16          else if (a == c) kind = ISOSCELE;
17          else if (b == c) kind = ISOSCELE;
18          else checkRightAngle();
19      }
20  }
```

Note: check on triples that are not sides of triangle (method **`isTriangle`**) was omitted

Target

(2, 2, 2)
(2, 2, 1)
(2, 1, 2)
(3, 4, 2)
(4, 3, 3)
(4, 5, 6)
(3, 5, 2)
(3, 2, 2)

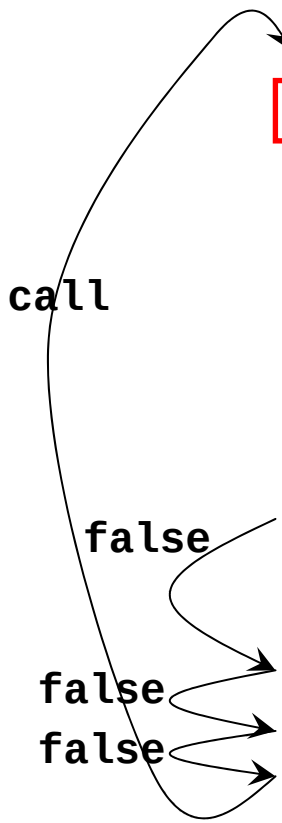
Covered statements
Not yet covered statements

```
1   class Triangle {
2       int a, b, c;
3       public void checkRightAngle() {
4           if (a*a + b*b == c*c)
5               kind = RIGHT_ANGLE;    TARGET
6           else if (b*b + c*c == a*a)
7               kind = RIGHT_ANGLE;
8           else if (a*a + c*c == b*b)
9               kind = RIGHT_ANGLE;
10          else kind = TRIANGLE;
11      }
12      public void computeTriangleKind() {
13          if (a == b)
14            if (b == c) kind = EQUILATERAL;
15            else kind = ISOSCELE;
16          else if (a == c) kind = ISOSCELE;
17          else if (b == c) kind = ISOSCELE;
18          else checkRightAngle();
19      }
20  }
```

Control/call dependences

[13, 16, 17, 18, 4]

(2, 2, 2)
(2, 2, 1)
(2, 1, 2)
(3, 4, 2)
(4, 3, 3)
(4, 5, 6)
(3, 5, 2)
(3, 2, 2)




```
1   class Triangle {
2       int a, b, c;
3       public void checkRightAngle() {
4           if (a*a + b*b == c*c)
5               kind = RIGHT_ANGLE;    TARGET
6           else if (b*b + c*c == a*a)
7               kind = RIGHT_ANGLE;
8           else if (a*a + c*c == b*b)
9               kind = RIGHT_ANGLE;
10          else kind = TRIANGLE;
11      }
12      public void computeTriangleKind() {
13          if (a == b)
14            if (b == c) kind = EQUILATERAL;
15            else kind = ISOSCELE;
16          else if (a == c) kind = ISOSCELE;
17          else if (b == c) kind = ISOSCELE;
18          else checkRightAngle();
19      }
20  }
```

Fitness

	Fitness
(2, 2, 2)	[13] = 1
(2, 2, 1)	[13] = 1
(2, 1, 2)	[13, 16] = 2
(3, 4, 2)	[13, 16, 17, 18, 4] = 5
(4, 3, 3)	[13, 16, 17] = 3
(4, 5, 6)	[13, 16, 17, 18, 4] = 5
(3, 5, 2)	[] = 0
(3, 2, 2)	[13, 16, 17] = 3

[13, 16, 17, 18, 4]

```
1   class Triangle {
2       int a, b, c;
3       public void checkRightAngle() {
4           if (a*a + b*b == c*c)
5               kind = RIGHT_ANGLE;    TARGET
6           else if (b*b + c*c == a*a)
7               kind = RIGHT_ANGLE;
8           else if (a*a + c*c == b*b)
9               kind = RIGHT_ANGLE;
10          else kind = TRIANGLE;
11      }
12      public void computeTriangleKind() {
13          if (a == b)
14            if (b == c) kind = EQUILATERAL;
15            else kind = ISOSCELE;
16          else if (a == c) kind = ISOSCELE;
17          else if (b == c) kind = ISOSCELE;
18          else checkRightAngle();
19      }
20  }
```

Selection

	Fitness	Probab.
(2, 2, 2)	1	0.05
(2, 2, 1)	1	0.05
(2, 1, 2)	2	0.1
(3, 4, 2)	5	0.25
(4, 3, 3)	3	0.15
(4, 5, 6)	5	0.25
(3, 5, 2)	0	0
(3, 2, 2)	3	0.15

Selection

		Fitness	Probab.
(2, 2, 2)	×	1	0.05
(2, 2, 1)	×	1	0.05
(2, 1, 2)	√	2	0.1
(3, 4, 2)	√	5	0.25
(4, 3, 3)	√	3	0.15
(4, 5, 6)	√	5	0.25
(3, 5, 2)	×	0	0
(3, 2, 2)	√	3	0.15

Population

(3, 4, 2)
(4, 3, 3)
(4, 5, 6)
(4, 5, 6)
(3, 2, 2)
(2, 1, 2)
(3, 4, 2)
(4, 3, 3)

Population'

Mutation operators

Change value

0100100010

(3, 4, 2)

0100110010

(3, 2, 2)

Crossover

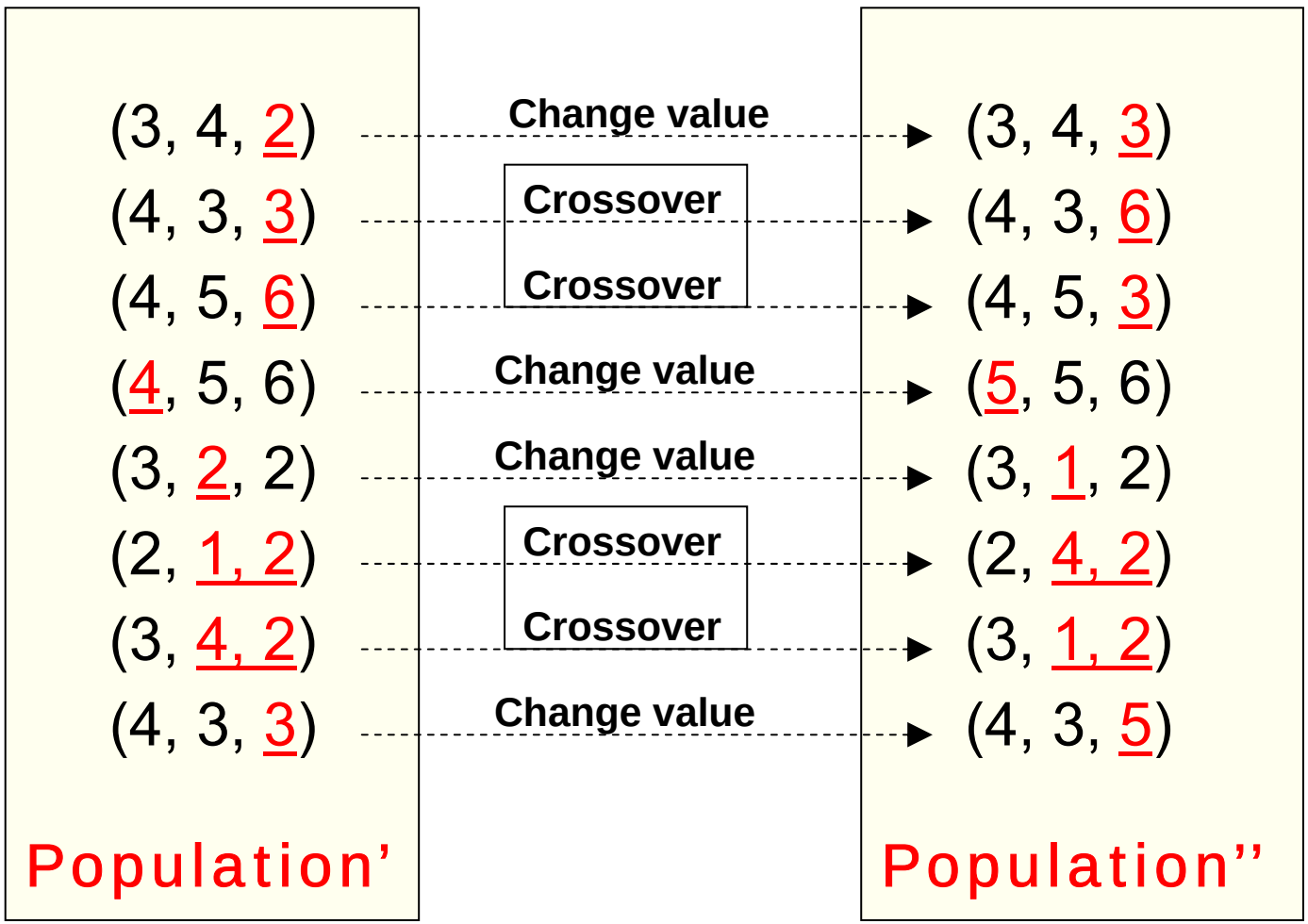
0100100010
1010010111

(3, 4, 2)
(4, 5, 6)

0100100111
1010010010

(3, 5, 6)
(4, 4, 2)

Mutation


(3, 4, 2)
(4, 3, 3)
(4, 5, 6)
(4, 5, 6)
(3, 2, 2)
(2, 1, 2)
(3, 4, 2)
(4, 3, 3)
Population'
Change value
Crossover
Crossover
Change value
Change value
Crossover
Crossover
Change value
(3, 4, 3)
(4, 3, 6)
(4, 5, 3)
(5, 5, 6)
(3, 1, 2)
(2, 4, 2)
(3, 1, 2)
(4, 3, 5)
Population''

Execution

	Trace
(3, 4, 3)	<13, 16>
(4, 3, 6)	<13, (16), (17), 18, 4, 6, 8, 10>
(4, 5, 3)	<13, (16), (17), 18, 4, 6, 7>
(5, 5, 6)	<13, (14), 15>
(3, 1, 2)	<>
(2, 4, 2)	<>
(3, 1, 2)	<>
(4, 3, 5)	<13, (16), (17), 18, 4, 5>

TARGET COVERED

```
1   class Triangle {
2       int a, b, c;
3       public void checkRightAngle() {
4           if (a*a + b*b == c*c)
5               kind = RIGHT_ANGLE;      TARGET
6           else if (b*b + c*c == a*a)
7               kind = RIGHT_ANGLE;
8           else if (a*a + c*c == b*b)
9               kind = RIGHT_ANGLE;
10          else kind = TRIANGLE;
11      }
12      public void computeTriangleKind() {
13          if (a == b)
14            if (b == c) kind = EQUILATERAL;
15            else kind = ISOSCELE;
16          else if (a == c) kind = ISOSCELE;
17          else if (b == c) kind = ISOSCELE;
18          else checkRightAngle();
19      }
20  }
```

Execution

	Trace
(3, 4, 3)	<13, 16>
(4, 3, 6)	<13, (16), (17), 18, 4, 6, 8, 10>
(4, 5, 3)	<13, (16), (17), 18, 4, 6, 8, 9>
(5, 5, 6)	<13, (14), 15>
(3, 1, 2)	<>
(2, 4, 2)	<>
(3, 1, 2)	<>
(4, 3, 5)	<13, (16), (17), 18, 4, 5>

PREVIOUSLY UNCOVERED

```
1   class Triangle {
2       int a, b, c;
3       public void checkRightAngle() {
4           if (a*a + b*b == c*c)
5               kind = RIGHT_ANGLE;
6           else if (b*b + c*c == a*a)
7               kind = RIGHT_ANGLE;
8           else if (a*a + c*c == b*b)
9               kind = RIGHT_ANGLE;
10          else kind = TRIANGLE;
11      }
12      public void computeTriangleKind() {
13          if (a == b)
14            if (b == c) kind = EQUILATERAL;
15            else kind = ISOSCELE;
16          else if (a == c) kind = ISOSCELE;
17          else if (b == c) kind = ISOSCELE;
18          else checkRightAngle();
19      }
20  }
```

Next evolution

	Trace
(3, 4, 3)	<13, 16>
(4, 3, 6)	<13, (16), (17), 18, 4, 6, 8, 10>
(4, 5, 3)	<13, (16), (17), 18, 4, 6, 7>
(5, 5, 6)	<13, (14), 15>
(3, 1, 2)	<>
(2, 4, 2)	<>
(3, 1, 2)	<>
(4, 3, 5)	<13, (16), (17), 18, 4, 5>

```
1   class Triangle {
2       int a, b, c;
3       public void checkRightAngle() {
4           if (a*a + b*b == c*c)
5               kind = RIGHT_ANGLE;
6           else if (b*b + c*c == a*a)
7               kind = RIGHT_ANGLE;   NEW TARGET
8           else if (a*a + c*c == b*b)
9               kind = RIGHT_ANGLE;
10          else kind = TRIANGLE;
11      }
12      public void computeTriangleKind() {
13          if (a == b)
14            if (b == c) kind = EQUILATERAL;
15            else kind = ISOSCELE;
16          else if (a == c) kind = ISOSCELE;
17          else if (b == c) kind = ISOSCELE;
18          else checkRightAngle();
19      }
20  }
```

Final test suite

Include all test cases that allow covering a previously uncovered statement.

(2, 2, 2)<13,14>

(2, 2, 1)<13,(14),15>

(2, 1, 2)<13,16>

(3, 2, 2)<13,(16),17>

(3, 4, 2)<13,(16),(17),18,4,6,8,10>

(4, 3, 5)<13,(16),(17),18,4,5>

(4, 5, 3)<13,(16),(17),18,4,6,8,9>

(10, 8, 6)<13,(16),(17),18,4,6,7>

```
1   class Triangle {
2       int a, b, c;
3       public void checkRightAngle() {
4           if (a*a + b*b == c*c)
5               kind = RIGHT_ANGLE;
6           else if (b*b + c*c == a*a)
7               kind = RIGHT_ANGLE;
8           else if (a*a + c*c == b*b)
9               kind = RIGHT_ANGLE;
10          else kind = TRIANGLE;
11      }
12      public void computeTriangleKind() {
13          if (a == b)
14            if (b == c) kind = EQUILATERAL;
15            else kind = ISOSCELE;
16          else if (a == c) kind = ISOSCELE;
17          else if (b == c) kind = ISOSCELE;
18          else checkRightAngle();
19      }
20  }
```

Automatic execution

```
class TestTriangle extends TestCase {
    public void testCase1() {
        Triangle t = new Triangle(2, 2, 2);
       assertTrue(t.computeTriangleKind() == Triangle.EQUILATERAL);
    }
    public void testCase2() {
        Triangle t = new Triangle(2, 2, 1);
        assertTrue(t.computeTriangleKind() == Triangle.ISOSCELE);
    }
    …
    public void testCase8() {
        Triangle t = new Triangle(10, 8, 6);
        assertTrue(t.computeTriangleKind() == Triangle.RIGHT_ANGLE);
    }
}
```

Junit

```
> java junit.textui.TestRunner TestTriangle
........
Time: 0.016
OK (8 tests)
```

100% statement coverage

Conclusions

- Our life depends on software systems in several respects
- We expect the software to be reliable, but in practice bug-free software cannot be produced
- *Testing* mitigates the risks associated with software bugs, by means of techniques that aggressively spot their presence in the code