Recap: structural code coverage

Entry point

- $a == \text{null} || a\text{.length} == 0$
- $i = 0$
- $\text{sum} = 0$
- $\text{num} = a[i]$
- $\text{sum} += \text{num}$
- $++i$

Exceptional exit

Normal exit

Recap: statement coverage

Entry point

- $a == \text{null} || a\text{.length} == 0$
- $i = 0$
- $\text{sum} = 0$
- $\text{num} = a[i]$
- $\text{num} < 0$
- $\text{sum} -= \text{num}$
- $++i$

Exceptional exit

Normal exit
Recap: decision coverage

```
Recap: condition coverage

Recap: subsumption relationships
```

Recap: subsumption relationships

Given two coverage criteria A and B, A subsumes B iff satisfying A implies satisfying B

- Subsumption relationships:
  - **Statement** coverage does not subsume **decision** coverage
  - **Decision** coverage subsumes **statement** coverage
  - **Decision** coverage does not subsume **condition** coverage
  - **Condition** coverage does not subsume **decision** coverage

Does decision coverage subsume statement coverage in a sequential method (i.e., one without any decisions)?
**Code coverage: advantages**

- Code coverage is easy to compute and visualize.
- Code coverage is a simple objective function.
- Code coverage has an intuitive interpretation.

But, does coverage ensure effective testing?

**Code coverage: drawbacks**

- Code coverage does not require test assertions.
- Not all statements etc. are equally important.

Are there any alternatives?

**Mutation testing: overview**

- Code coverage is easy to compute and visualize.
- Code coverage is a simple objective function.
- Code coverage has an intuitive interpretation.

But, does coverage ensure effective testing?
Mutation testing: overview

```
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```

Each mutant contains one small syntactic change

```
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```

```
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```

```
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```
Mutation testing: overview

Assumption: Mutant detection rate is a useful proxy for software testing. (https://homes.cs.washington.edu/~rjust/publ/mutants_real_faults_fse_2014.pdf)

Mutation testing: example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

Original program:
```java
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant:
```java
public int min(int a, int b) {
    return a;
}
```

Assumption: Mutant detection rate is a useful proxy for software testing. (https://homes.cs.washington.edu/~rjust/publ/mutants_real_faults_fse_2014.pdf)

Mutation testing: example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

Original program:
```java
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant:
```java
public int min(int a, int b) {
    return a;
}
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Original</th>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```
Mutation testing: another example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

Original program:
```java
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant:
```java
public int min(int a, int b) {
    return b;
}
```

---

Mutation testing: yet another example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

Original program:
```java
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant:
```java
public int min(int a, int b) {
    return a >= b ? a : b;
}
```
Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

Original program:
```java
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant:
```java
public int min(int a, int b) {
    return a <= b ? a : b;
}
```

There is no such test that can detect the mutant...
The mutant is undetectable: it's semantically equivalent to the original program!

Mutation testing: summary

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Original</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Mutation testing: challenges

- Redundant mutants
  - Inflate the mutant detection ratio
  - Hard to assess progress and remaining effort
- Equivalent mutants
  - Max mutant detection ratio != 100%
  - Wastes resources (CPU and human time)

```
a  b  Original  M1  M2  M3  M4
1  2  1  2  2  1
1  1  1  1  1  1
2  1  2  1  2  1
```

Mutation testing: live example

Let’s start with a coverage-adequate test suite

```
package avg;

public class Avg {

    /**
     * Compute the average of the absolute values of an array of doubles
     */
    public double absavg(double... numbers) {
        // We expect the array to be non-null and non-empty
        if (null == numbers || numbers.length == 0) {
            throw new IllegalArgumentException("Array numbers must not be null or empty!");
        }
        double sum = 0;
        for (int i = 0; i < numbers.length; ++i) {
            double d = numbers[i];
            if ( i > 0) {
                sum -= d;
            } else {
                sum += d;
            }
        }
        return sum/numbers.length;
    }
}
```