1 Equivalence Classes & Boundary Values

A local flower shop offers discounts to its most valuable customers. Function discountPercent calculates the percentage of discount that should be applied to a purchase. Purchases of more than 10 flowers are given 10discount. (The discounts are cumulative.) Function discountPercent throws an InvalidOrderException if the number of flowers is zero.

```
public int discountPercent(int flowers, boolean membershipCard);
```

a) What are the partitions for each parameter? How many partitions are there in total?

b) What partitions can be combined?

c) What are the boundary values? Which are the on and off points?

d) Construct test cases for function discountPercent according to your analysis. Give inputs and expected outputs.

1.1 Solution

a) Values for flowers:

- < 0
- $= 0$
- between 1 and 10
- > 10

Values for membershipCard:

- true
- false

Combinations of inputs:

- flowers is < 0 and membershipCard is true
- flowers is < 0 and membershipCard is false
- flowers is $= 0$ and membershipCard is true
- flowers is $= 0$ and membershipCard is false
- flowers is between 1 and 10 and membershipCard is true
- flowers is between 1 and 10 and membershipCard is false
- flowers is > 10 and membershipCard is true
- flowers is > 10 and membershipCard is false

Outputs are:

- InvalidOrderException
- 0%
- 5%
- 10%
- 15%

In total there are ~~8 Partitions~~ 6 Partitions

b) We could combine the cases for flowers being smaller or equal to 0 as there is either no defined behavoir or an Error is thrown.

c) Boundary values would be (depending on the value of flowers):

- 0, On: the function throws an Error; Off: the function returns either 0% or 5%
- 10, On: the function returns either 10% or 15%; Off: the function returns either 0% or 5%

d)

```
discountPercent(-1, true); // Expected: InvalidOrderException
discountPercent(0, true); // Expected: InvalidOrderException
discountPercent(1, true); // Expected: 5
discountPercent(1, false); // Expected: 0
discountPercent(9, true); // Expected: 5
discountPercent(10, true); // Expected: 15
discountPercent(11, false); // Expected: 10
```

2 Specification-Based & Structural Testing

a) Which of these software testing activities correspond to specification-based testing, which correspond to structural testing, and which correspond to neither?

(a) Asking a colleague to check if the tests match the documentation. **Specification-Based Testing** ✓

(b) Measuring which statements are executed by each test case. **Structural Testing** ✓

(c) Doing test-driven development. **Specification-Based Testing** ✓

(d) Testing with random data to find crashes. ~~**Structural Testing**~~ neither

(e) Constructing test cases to cover all branches. **Structural Testing** ✓

b) Which of the following statements are correct?

(a) MC/DC is a stronger property than branch coverage. **True** ✓

(b) Programs that have 100% path coverage do not contain any kind of bugs. **False** ✓

(c) Boundary values are extracted from the source code. **False**

(d) Loop coverage is a stronger property than branch coverage. ~~True~~ False ...

(e) A test suite constructed from boundary values has 100% branch coverage. ~~True~~ False

3 Basic-Block & Branch Coverage

```
public int compute(int[] x) {
    if (x == null) {
        return 0;
    }
    int sum = 0;
    for (int i = 0; i < x.length; i++) {
        if (x[i] % 2 == 0) {
            sum += x[i];
        }
    }
    return sum;
}
```

a) Draw the control flow graph. Count the basic blocks and branches

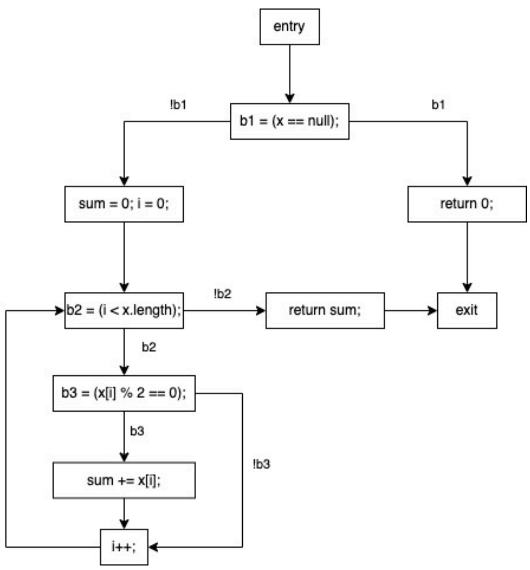


Figure 1: Control Flow Graph with 8 basic blocks, and 6 branches

b) Define test cases that achieve 100% basic-block coverage, but not 100% branch coverage.

x = null

x = { 0, 2, 4 }

c) Define test cases that achieve 100% branch coverage.

x = null

x = { 0, 2, 4 }

x = { 1, 3, 5 }

4 Path & Loop Coverage

a) Consider function max

```
public int max(int a, int b, int c) {
    int max = a;
    if (max < b) {
        max = b;
    }
    if (max < c) {
        max = c;
    }
    return max;
}
```

(a) Count the paths in function max. **4 Paths**

(b) List a set of test cases that achieve 100% path coverage.

(a=1, b=0, c=0)

(a=0, b=1, c=0)

(a=0, b=0, c=1)

(a=0, b=1, c=2)

(c) Which of these test cases are sufficient for 100% branch coverage?

(a=1, b=0, c=0) (a=0, b=1, c=2)

(d) Which of these test cases are sufficient for 100% basic block coverage? **(a=0, b=1, c=2)**

(e) How many paths does function max4 have? How many test cases are necessary to reach 100% path coverage?

```
public int max(int a, int b, int c, int d) {
    int max = a;
    if (max < b) {
        max = b;
    }
```

```
        if (max < c) {
            max = c;
        }
        if (max < d) {
            max = d;
        }
        return max;
    }
```

8 Paths, so 8 tests are necessary

b) Consider function sumRange.

```
public int sumRange(int[] array, int l, int r) {
    if (array == null || array.length != 4 || l < 0 || 4 <= r) {
        throw new IllegalArgumentException();
    }
    int sum = 0;
    while (l < r) {
        sum += array[l];
        l++;
    }
    return sum;
}
```

(a) Construct a minimal set of test cases that achieve 100% loop coverage. **(array = null, l = 0, r = 0)**
(array = {1, 2, 3, 4}, l = 0, r = 1)
(array = {1, 2, 3, 4}, l = 0, r = 3)

(b) Do these tests reach 100
Yes, but if we didn't use array = null in the first test, the first branch wouldn't be fully covered.

5 Condition + Branch Coverage

```
public String triangle(int a, int b, int c) {
    if (a + b < c || a + c < b || b + c < a) {
        return "invalid";
    }

    if (a * a + b * b == c * c || a * a + c * c == b * b
```

```
        || b * b + c * c == a * a) {
            return "right_angled";
        }
        return "other";
    }
```

a) Count the number of condition values + branches.

b) How much branch coverage does the test (a=1, b=1, c=1) reach?

c) How much C+B coverage does the test (a=1, b=1, c=1) reach?

d) Construct test cases that reach 100% C+B coverage.

5.1 Solution

a) $A = a+b < c$, $B = a+c < b$, $C = b+c < a$, $D = a^2+b^2 = c^2$, $E = a^2+c^2 = b^2$, $F = b^2+c^2 = a^2$ So in total there are 12 condition values and 4 branches.

b) $\frac{2}{4} = 50\%$ Branch coverage

c) $\frac{2+6}{4+12} = 50\%$

Branch 1 Branch 2

d) (a=0, b=0, c=1): **A=true, B=false, C=false, D=false, E=false, F=false**

(a=0, b=1, c=0): **A=false, B=true, C=false, D=false, E=false, F=false**

(a=1, b=0, c=0): **A=false, B=false, C=true, D=false, E=false, F=false**

(a=0, b=1, c=1): **A=false, B=false, C=false, D=true, E=false, F=false**

(a=1, b=0, c=1): **A=false, B=false, C=false, D=false, E=false, F=true**

(a=1, b=1, c=0): **A=false, B=false, C=false, D=false, E=true, F=false**

(a=1, b=2, c=3): **A=false, B=false, C=false, D=false, E=false, F=false**

6 MC/DC

```
public int compute(int a, int b) {
    if ((a * b == 20 || a + b == 12) && a < 10) {
        return a;
    } else {
        return b;
    }
}
```

a) Construct test cases that reach 100% MC/DC. List for each test case which conditions are true and which are false.

b) List the independence pair for each condition.

6.1 Solution

a) Construct test cases that reach 100% MC/DC. List for each test case which conditions are true and which are false.

We can define Condition Variables, $A := a < 10,\ B := a \cdot b = 20,\ C := a + b = 12$, such that the condition we want to evaluate is: $A \wedge (B \vee C)$

Test	A	B	C	Result
$a = 2, b = 10$	T	T	T	a
$a = 5, b = 4$	T	T	F	a
$a = 6, b = 6$	T	F	T	a
$a = 1, b = 0$	T	F	F	b
$a = 10, b = 2$	F	T	T	b
$a = 20, b = 1$	F	T	F	b
$a = 11, b = 1$	F	F	T	b
$a = 10, b = 0$	F	F	F	b

Using MC/DC we can define minimal test cases $\{(a = 5, b = 4), (a = 6, b = 6), (a = 1, b = 0), (a = 20, b = 1)\}$

do this first

b) List the independence pair for each condition.
$A : \{(a = 2, b = 10),\ (a = 10, b = 2)\},\ \{(a = 5, b = 4),\ (a = 20, b = 1)\},\ \{(a = 6, b = 6),\ (a = 11, b = 1)\}$
$B : \{(a = 5, b = 4),\ (a = 1, b = 0)\}$
$C : \{(a = 6, b = 66),\ (a = 1, b = 0)\}$

7 DU-Pairs Coverage

```
public int range(int a, int b, int c) {
    int max = a;
    int min = a;
    if (a < b) {
        max = b;
    } else {
        min = b;
    }
    if (max < c) {
        max = c;
```

```
    } else {
        min = c;
    }
    return max - min;
}
```

a) List all DU pairs for variables max and min. **For max: {(1, 3), (1, 5), (1, 9), (3, 5), (3, 9), (6, 9)}, For min: {(1, 4), (1, 7), (1, 9), (4, 7), (4, 9), (8, 9) }**

b) Construct test cases that reach 100% DU-pairs coverage. For each test, list all DU pairs it covers. **(a=1, b=2, c=3): (1, 3), (3, 5), (6, 9); (1, 7), (1, 9)**
(a=3, b=2, c=1): (1, 5), (1, 9); (1, 4), (4, 7), (8, 9)
(a=1, b=3, c=2): (1, 3), (3, 5), (3, 9); (1, 7), (1, 9)
(a=3, b=2, c=3): (1, 9); (1, 4), (4, 7), (4, 9)

8 Measuring DU-Pairs Coverage

```
public void countFlips(boolean[] coinFlips, boolean countHeads {
    int heads = 0;
    int tails = 0;
    int result = 0;

    for (boolean isHeads: coinFlips) {
        if (isHeads) {
            heads = heads + 1;
        } else {
            tails = tails + 1;
        }
    }
    if (countHeads) {
        result = heads;
    } else {
        result = tails;
    }
    return result;
}
```

a) Draw the control flow graph for function countFlips and apply the algorithm for computing reaching definitions for variables heads, tails and result.

b) List the DU pairs for variables heads, tails and result.

n	Reach(n)	ReachOut(n)
1	$\emptyset$	$heads_1, tails_1, result_1$
2	$heads_1, heads_5, tails_1, tails_6, result_1$	$heads_1, heads_5, tails_1, tails_6, result_1$
3	$heads_1, heads_5, tails_1, tails_6, result_1$	$heads_1, heads_5, tails_1, tails_6, result_1$
4	$heads_1, heads_5, tails_1, tails_6, result_1$	$heads_1, heads_5, tails_1, tails_6, result_1$
5	$heads_1, heads_5, tails_1, tails_6, result_1$	$heads_5, tails_1, tails_6, result_1$
6	$heads_1, heads_5, tails_1, tails_6, result_1$	$heads_1, heads_5, tails_6, result_1$
7	$heads_1, heads_5, tails_1, tails_6, result_1$	$heads_1, heads_5, tails_1, tails_6, result_1$
8	$heads_1, heads_5, tails_1, tails_6, result_1$	$heads_1, heads_5, tails_1, tails_6, result_8$
9	$heads_1, heads_5, tails_1, tails_6, result_1$	$heads_1, heads_5, tails_1, tails_6, result_9$
10	$heads_1, heads_5, tails_1, tails_6, result_1, result_8, result_9$	$heads_1, heads_5, tails_1, tails_6, result_1, result_8, result_9$

c) Instrument the code as shown in the lecture to measure DU-pairs coverage. What is the state of maps defCover and useCover after running the test case (coinFlips=[true, true], countHeads = false)? You may assume the maps start freshly initialized.

8.1 Solution

a) The table:

See next page for better version

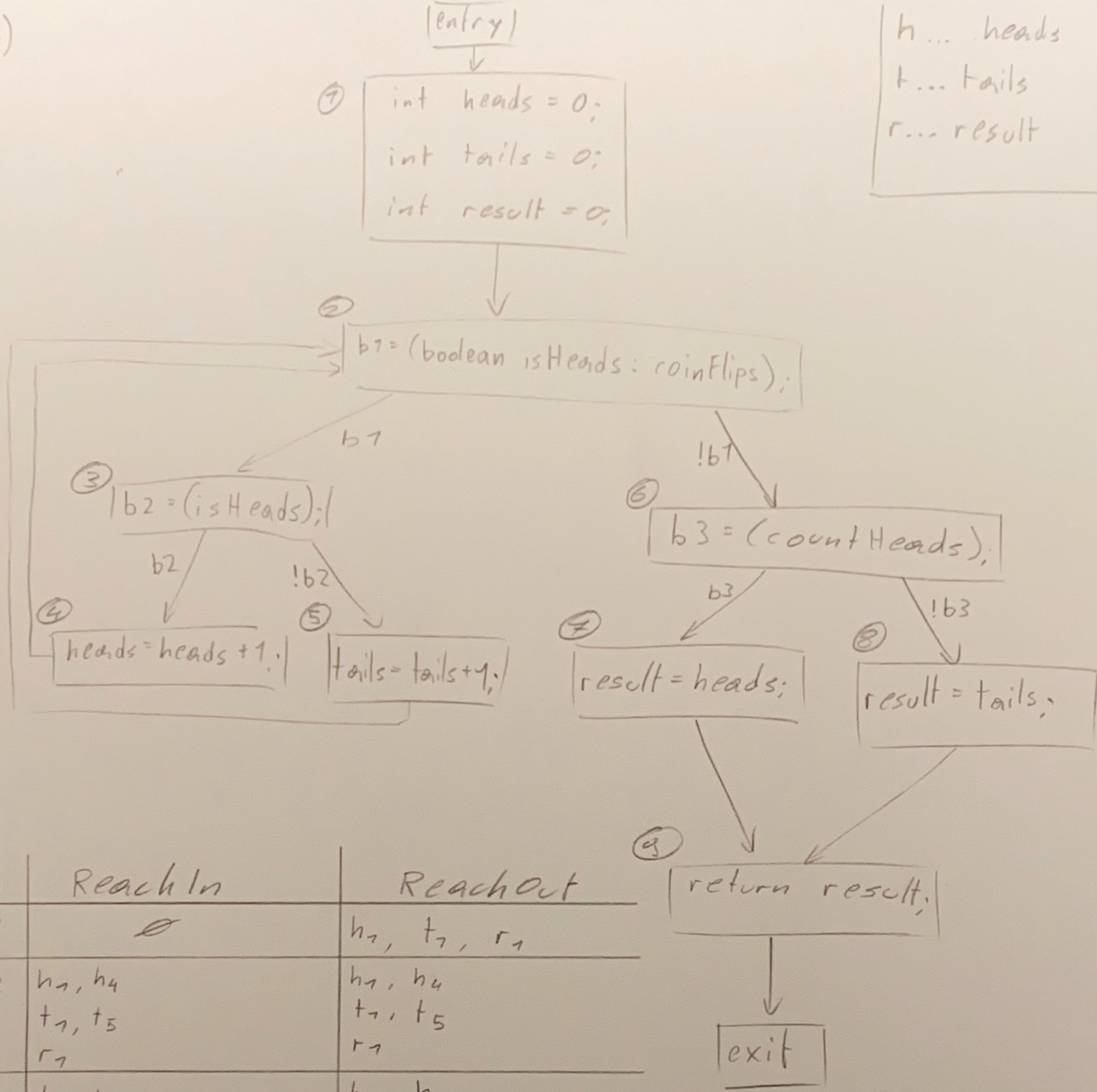
b) DU-Pairs for heads: $(1,5), (5,5), (1,8), (5,8)$
DU-Pairs for tails: $(1,6), (6,6), (1,9), (6,9)$
DU-Pairs for result: $(1,8), (1,9), (8,10), (9,10)$

c) Instrument the code as shown in the lecture to measure DU-pairs coverage.

```
public void countFlips(boolean[] coinFlips, boolean countHeads {
1     int heads=0; defCover["heads"]=1;
1     int tails=0; defCover["tails"]=1;
1     int result=0; defCover["result"]=1;

2,3   for (boolean isHeads: coinFlips) {
4         if (isHeads) {
5             heads=heads+1; useCover["heads", defCover["heads"], 5]++;
                             defCover["heads"]=5;
          } else {
6             tails=tails+1; useCover["tails", defCover["tails"], 6]++;
                             defCover["tails"]=6;
          }
```

a)



h ... heads
t ... tails
r ... result

n	ReachIn	ReachOut
1	∅	h_1, t_1, r_1
2	h_1, h_4 t_1, t_5 r_1	h_1, h_4 t_1, t_5 r_1
3	h_1, h_4 t_1, t_5 r_1	h_1, h_4 t_1, t_5 r_1
4	h_1, h_4 t_1, t_5 r_1	h_4 t_1, t_5 r_1
5	h_1, h_4 t_1, t_5 r_1	h_1, h_4 t_5 r_1
6	h_1, h_4 t_1, t_5 r_1	h_1, h_4 t_1, t_5 r_1
7	h_1, h_4 t_1, t_5 r_1	h_1, h_4 t_1, t_5 r_7
8	h_1, h_4 t_1, t_5 r_1	h_1, h_4 t_1, t_5 r_8
9	h_1, h_4 t_1, t_5 r_7, r_8	h_1, h_4 t_1, t_5 r_7, r_8

b) Var-Use underlined in table

heads: (1, 4), (4, 4), (1, 7), (4, 7)

tails: (1, 5), (5, 5), (1, 8), (5, 8)

result: (7, 9), (8, 9)

```
     }
7    if (countHeads) {
8         result=heads; useCover["heads", defCover["heads"], 8]++;
                        defCover["result"]=8;
     } else {
9         result=tails; useCover["tails", defCover["tails"], 9]++;
                        defCover["result"]=9;
     }
10   return result; useCover["result", defCover["result"], 10]++;
}
```

What is the state of maps defCover and useCover after running the test case (coinFlips=[true, true], countHeads = false)? You may assume the maps start freshly initialized.

```
defCover["heads"] = 5;
defCover["tails"] = 1;
defCover["result"] = 9;
useCover["heads", 1, 5] = 1;
useCover["heads", 5, 5] = 1;
useCover["heads", 1, 8] = 0;
useCover["heads", 5, 8] = 0;
useCover["tails", 1, 9] = 1;
useCover["tails", 1, 6] = 0;
useCover["tails", 6, 6] = 0;
useCover["tails", 6, 9] = 0;
useCover["result", 9, 10] = 1;
useCover["result", 8, 10] = 0;
```

9 Property-Based Testing

a) An Austrian drink wholesaler would like to apply property-based testing to their web shop. The company offers beverages with alcohol ranging from 0% to 53%. The policy of the web shop states that customers under 16 are only allowed to buy non-alcoholic beverages (0% alcohol). Customers between 16 and 17 are allowed to buy drinks with under 20% of alcohol. There are no restrictions for customers of age 18 or older.

   Apply property-based testing to function canOrderDrink.

   ```
   public boolean canOrderDrink(int age, int alcoholPercent) { ... }
   ```

   In particular, design property-based tests for the following requirements:

(a) People under the age of 16 are allowed to order alcohol-free drinks.

(b) From the age of 16 to 17, people are allowed to order drinks whose alcohol percentage is under 20.

(c) From the age of 18, people are allowed to order any kind of drink.

(d) People under the age of 16 are not allowed to order any alcoholic drink.

(e) From the age of 16 to 17, people are not allowed to order drinks with an alcohol percentage of 20 or above.

b) Which of the following statements are correct about property-based testing?

- Writing properties is easier than constructing tests manually
- Property-based testing should always be used instead of example-based testing
- With property-based testing, it may be difficult to get an adequate distribution of input values.
- With property-based testing, it is always inexpensive to generate the desired data.
- A property-based testing framework tries to find a counterexample to break the defined properties.

c) Consider function compute

```
public void compute(int a, int b);
@Property
void computeTest(
    @ForAll @IntRange(min = 1, max = 100) int a,
    @ForAll @IntRange(min = 1, max = 100) int b
) {
    // ...
}
```

Function compute has a bug and incorrectly throws an exception if inputs a and b are equal. Assume that for each run of property computeTest the values for a and b are sampled independently and uniformly in the given range.

- What is the probability that a generated pair of input values reveals the bug?
- What is the probability if the max value for both variables a and b is increased to 1000?

9.1 Solution

a)
```
@Property
void testProperty1CanOrderDrink (
    @ForAll
```

```
        @IntRange(min = 0, max = 15)
        int age,
        @ForAll
        @IntRange(min = 0, max = 0)
        int alcoholPercent) {
        System.out.println("Age: " + age + " Alcohol: " + alcoholPercent);
        assertTrue(canOrderDrink(age, alcoholPercent));
    }

    @Property
    void testProperty2CanOrderDrink (
        @ForAll
        @IntRange(min = 16, max = 17)
        int age,
        @ForAll
        @IntRange(min = 0, max = 19)
        int alcoholPercent) {
        System.out.println("Age: " + age + " Alcohol: " + alcoholPercent);
        assertTrue(canOrderDrink(age, alcoholPercent));
    }

    @Property
    void testProperty3CanOrderDrink (
        @ForAll
        @IntRange(min = 18, max = 100)
        int age,
        @ForAll
        @IntRange(min = 0, max = 53)
        int alcoholPercent) {
        System.out.println("Age: " + age + " Alcohol: " + alcoholPercent);
        assertTrue(canOrderDrink(age, alcoholPercent));
    }

    @Property
    void testProperty4CanOrderDrink (
        @ForAll
        @IntRange(min = 0, max = 15)
        int age,
        @ForAll
        @IntRange(min = 1, max = 53)
```

```
        int alcoholPercent) {
        System.out.println("Age: " + age + " Alcohol: " + alcoholPercent);
        assertFalse(canOrderDrink(age, alcoholPercent));
    }

    @Property
    void testProperty5CanOrderDrink (
        @ForAll
        @IntRange(min = 16, max = 17)
        int age,
        @ForAll
        @IntRange(min = 20, max = 53)
        int alcoholPercent) {
        System.out.println("Age: " + age + " Alcohol: " + alcoholPercent);
        assertFalse(canOrderDrink(age, alcoholPercent));
    }
```

b)
- Writing properties is easier than constructing tests manually ~~**true**~~ false
- Property-based testing should always be used instead of example-based testing **false**
- With property-based testing, it may be difficult to get an adequate distribution of input values. **true**
- With property-based testing, it is always inexpensive to generate the desired data. **false**
- A property-based testing framework tries to find a counterexample to break the defined properties. **true**

c) The probability is similar to a sampling with replacement. We choose on value a, the probability that the second value b is equal to a is $\frac{1}{100} = 1\%$. Therefore if we increase the range this probability becomes $\frac{1}{1000} = 0.1\%$

10 Test Doubles

Classify the following objects into one of the five kinds of test doubles.

a) An external API server that returns pre-defined responses and verifies that specific requests were made during testing. **Mock**

b) A database connection wrapper that records every query made to a particular table. **Spy**

c) A database connection that returns pre-defined data for specific queries. **Stub**

d) A logger that does not perform any logging and is only used to fulfill a method requirement. **Dummy Object**

e) A file system that emulates the behavior of a real file system without actually writing to disk. **Fake Object**

f) An HTTP server that returns pre-defined responses to specific requests. **Stub**

g) An email service that captures and stores outgoing emails and triggers pre-defined incoming email events. ~~**Fake Object**~~ Mock

h) A database connection that ignores all operations and is not used during testing. **Dummy Object**

i) A logger that records information about logged messages during testing and checks for the existence of certain string patterns. ~~**Mock**~~ Spy

j) A data prediction unit that, in contrast to its production implementation, uses a simplified algorithm to decrease the runtime of the tests. **Fake Object**