

Assignment 6 - Solution

Exercise 1

1. See Figure 1.
2. The bug in `indexOf` is revealed by computing the index of an element that is not in the array.

```
@Test
public void testIndexOfBug() {
    int actual = Array.indexOf(new int[]{0, 1, 2, 3}, 4);
    int expected = -1;
    assertEquals(expected, actual);
}
```

3. The following tests for `indexOf` achieve 100% statement coverage and miss the bug.

```
@Test(expected = IllegalArgumentException.class)
public void testIndexOfNullArgument() {
    Array.indexOf(null, 0);
}
```

```
@Test
public void testIndexOfCoverage() {
    int actual = Array.indexOf(new int[]{0, 1, 2, 3}, 2);
    int expected = 2;
    assertEquals(expected, actual);
}
```

4. There is no test suite for `indexOf` that achieves 100% branch coverage and misses the bug.
5. The bug in `average` is revealed if the result depends on the first element of the array.

```

@Test
public void testAverageBug() {
    int actual = Array.average(new int[]{1});
    int expected = 1;
    assertEquals(expected, actual);
}

```

6. The following tests for `average` achieve 100% branch coverage and miss the bug.

```

@Test
public void testAverage() {
    int actual = Array.average(new int[]{0, 2, 4, 10});
    int expected = 4;
    assertEquals(expected, actual);
}

```

```

@Test(expected = IllegalArgumentException.class)
public void testAverageNullArgument() {
    Array.average(null);
}

```

```

@Test(expected = IllegalArgumentException.class)
public void testAverageEmptyList() {
    Array.average(new int[]{});
}

```

7. Only loop coverage guarantees to find the bug.
8. We find the bug if the values of `l` and `r` are the same. The probability for this to happen is $1/101$.

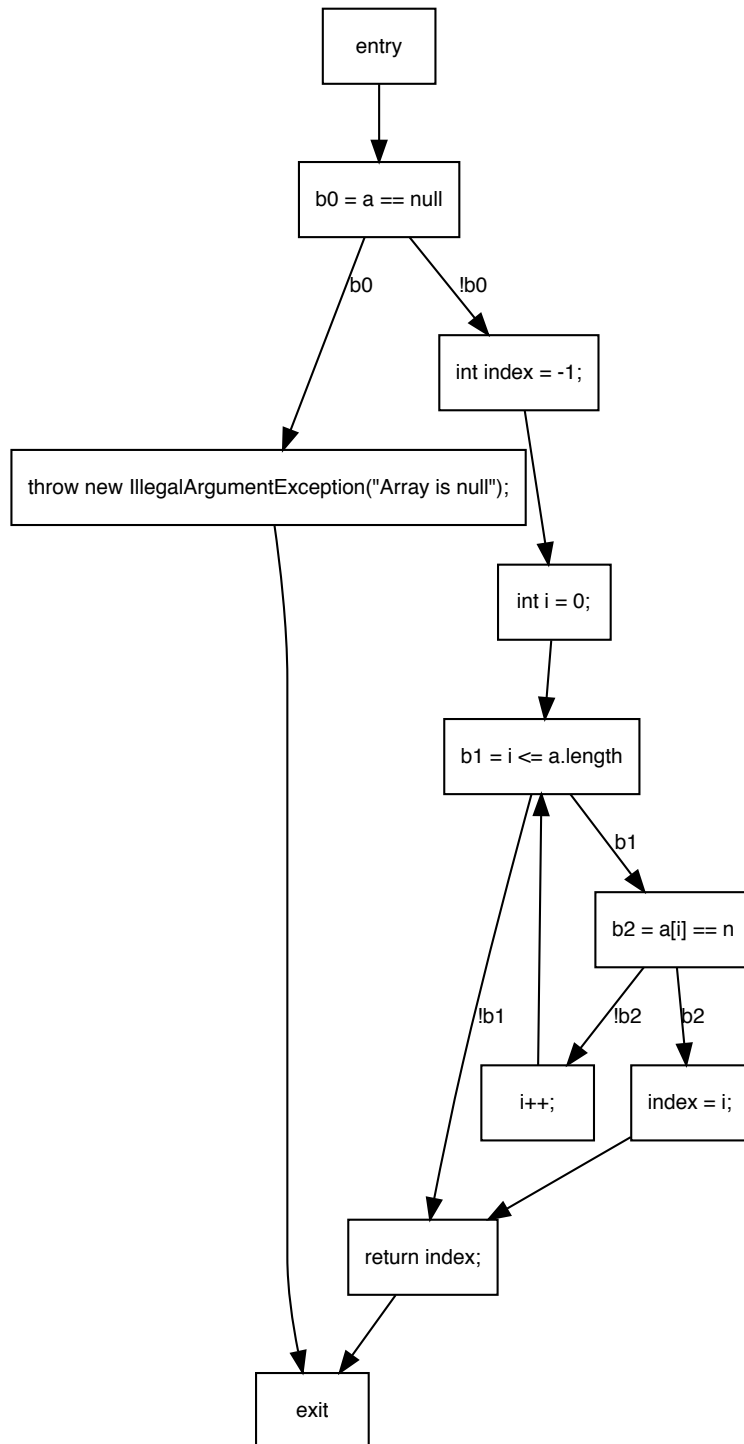


Figure 1: The control flow graph for the method `indexOf`.

Exercise 2

1. Below are some examples of functional tests.

```
@Test
public void testKnapsackFunctional() {
    // no items
    assertEquals(0, Knapsack.solve(
        new int[] {},
        new int[] {}, 0));

    // one item
    assertEquals(0, Knapsack.solve(
        new int[] {0},
        new int[] {0}, 0));
    assertEquals(2, Knapsack.solve(
        new int[] {2},
        new int[] {0}, 1));
    assertEquals(0, Knapsack.solve(
        new int[] {0},
        new int[] {2}, 1));
    assertEquals(1, Knapsack.solve(
        new int[] {1},
        new int[] {1}, 1));

    // several items
    assertEquals(5, Knapsack.solve(
        new int[] {2, 1, 3},
        new int[] {2, 1, 3}, 5));
    assertEquals(9, Knapsack.solve(
        new int[] {3, 6, 2},
        new int[] {2, 2, 2}, 5));
}
```

2. Adding the following test will yield 100% branch coverage. It will also expose a bug that produces a stack overflow.

```
@Test
public void testKnapsackCoverage() {
    int actual = Knapsack.solve(
        new int[] {1, 2, 5},
        new int[] {1, 2, 5}, 4);
    int expected = 3;
    assertEquals(expected, actual);
}
```