Краткое summary

В качестве готового фаззера была выбрана библиотека JQF, которая позволяет генерировать как рандомные входные данные, так и свои структурированные. При чём рандомные входные данные не такие уж и рандомные, так как если fuzzer сгенерирует такие данные, что они пройдут дальше в плане покрытия кода, то с этого момента генерация будет изменена с учётом только что прошедшего набора входных данных. Тестовое покрытие измерялось всё при помощи того же JaCoCo (правда пришлось повозиться несколько часов, чтобы JaCoCo видел тесты фаззинга).

Тесты писались как для классического фаззинга, так и для генерируемого (Generative random testing). Все тесты классического фаззинга находятся в классах `FuzzingTests`, а генерируемого в `GenerativeFuzzingTests`. Также для классического фаззинга был увеличен лимит запусков со стандартных 100 до 1000 или 10000 в зависимости от сложности генерируемых данных. В генерируемом количество запусков стандартное - 100 итераций.

После запуска команды `mvn clean install` только с классическим фаззингом запустился процесс тестирования, результаты которого можно видеть ниже:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running lesson5.fuzzing.FuzzingTests
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 59.125 sec - in
lesson5.fuzzing.FuzzingTests
Running lesson6.fuzzing.FuzzingTests
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.883 sec - in
lesson6.fuzzing.FuzzingTests
Running lesson7.fuzzing.FuzzingTests
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 14.726 sec - in
lesson7.fuzzing.FuzzingTests

Results :

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

Всего было запущено 5 тестов, которые ни разу не сломали проверяемые функции.

Для генерируемого фаззинга результаты следующие:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running lesson5.fuzzing.GenerativeFuzzingTests
Tests run: 11, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.117 sec - in
lesson5.fuzzing.GenerativeFuzzingTests
Running lesson6.fuzzing.GenerativeFuzzingTests
Tests run: 112, Failures: 103, Errors: 0, Skipped: 0, Time elapsed: 1.224 sec <<<
FAILURE! - in lesson6.fuzzing.GenerativeFuzzingTests
Running lesson7.fuzzing.GenerativeFuzzingTests
Tests run: 51, Failures: 49, Errors: 0, Skipped: 0, Time elapsed: 0.206 sec <<<
FAILURE! - in lesson7.fuzzing.GenerativeFuzzingTests

Results :
```

```
Tests run: 26, Failures: 3, Errors: 0, Skipped: 0

[ERROR] There are test failures.
```

Таким образом, было запущено всего 26 тестов, 4 из которых упали из-за того, что некоторые проверяемые функции реализованы неправильно с точки зрения поставленной задачи.

lesson 5

В `lesson5` тестировалось две функции: `containsIn(Map, Map)` и `mergePhoneBooks(Map, Map)`.

Классический fuzzing

MapKt

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
bagPacking(Map, int)		0 %		0 %	14	14	22	22	1	1
findSumOfTwo(List, int)		0 %		0 %	17	17	16	16	1	1
averageStockPrice(List)		0 %		0 %	4	4	9	9	1	1
propagateHandshakes(Map)		0 %		0 %	8	8	15	15	1	1
canBuildFrom(List, String)		0 %		0 %	6	6	7	7	1	1
buildGrades(Map)		0 %		0 %	3	3	8	8	1	1
findCheapestStuff(Map, String)		0 %		0 %	6	6	10	10	1	1
extractRepeats(List)		0 %		0 %	4	4	11	11	1	1
filterByCountryCode(Map, String)		0 %		0 %	4	4	7	7	1	1
hasAnagrams(List)		0 %		0 %	5	5	9	9	1	1
subtractOf(Map, Map)		0 %		0 %	4	4	2	2	1	1
removeFillerWords(List, String[])		0 %		0 %	3	3	6	6	1	1
shoppingListCost(List, Map)		0 %		0 %	3	3	6	6	1	1
whoAreInBoth(List, List)		0 %		0 %	3	3	3	3	1	1
buildWordSet(List)		0 %		0 %	2	2	3	3	1	1
containsIn(Map, Map)		71 %		50 %	2	4	1	4	0	1
mergePhoneBooks(Map, Map)		100 %		83 %	1	4	0	8	0	1
Total	1 325 of 1 431	7 %	146 of 154	5 %	89	94	135	146	15	17

В функции `containsIn(Map, Map)` покрытие по инструкциям составило 71%, а по ветвям 50%. Покрытый код функции можно видеть ниже.

```
165. fun containsIn(a: Map<String, String>, b: Map<String, String>): Boolean {
166.     for (key in a.keys) {
167.         if (!b.containsKey(key)) return false
168.         if (a[key] != b[key]) return false
169.     }
170.     return true
171. }
```

На рисунке видно, что первый `if` покрыт не полностью, а второй и вовсе не был достигнут при тестировании. Это говорит нам о том, что случайно сгенерированные мапы даже не имели одинаковых ключей, иначе второй `if` был бы как минимум жёлтым.

В функции `mergePhoneBooks(Map, Map)` покрытие по инструкциям составило 100%, а вот по ветвям всего 83%. Покрытый код можно видеть ниже.

```
221. fun mergePhoneBooks(mapA: Map<String, String>, mapB: Map<String, String>): Map<String, String> {
222.     var res = mapA
223.     for ((name, phone) in mapB) {
224.         if (res.containsKey(name) && res[name] != phone) {
225.             var tmp = res[name]
226.             tmp = tmp!! + ", $phone"
227.             res = res + Pair(name, tmp)
228.         } else {
229.             res = res + Pair(name, phone)
230.         }
231.     }
232.     return res
233. }
```

Здесь были пройдены не все возможные условия, а если конкретно, то лишь одно, так как наведя на жёлтый ромбик всплывает окно окно "1 of 4 branches missed". Какое конкретно условие не было пройдено - сказать невозможно. Можно лишь утверждать, что это неудовлетворяющее условие, так как в `if` стоит логическое И, то есть из 4 возможных комбинаций условию будет удовлетворять лишь одна, а положительное ветвление раскрашено в зелёный цвет, значит как минимум один раз по этой ветке был проход.

Generative random testing

MapKt

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
bagPacking(Map, int)		0 %		0 %	14	14	22	22	1	1
findSumOfTwo(List, int)		0 %		0 %	17	17	16	16	1	1
averageStockPrice(List)		0 %		0 %	4	4	9	9	1	1
propagateHandshakes(Map)		0 %		0 %	8	8	15	15	1	1
canBuildFrom(List, String)		0 %		0 %	6	6	7	7	1	1
buildGrades(Map)		0 %		0 %	3	3	8	8	1	1
findCheapestStuff(Map, String)		0 %		0 %	6	6	10	10	1	1
extractRepeats(List)		0 %		0 %	4	4	11	11	1	1
filterByCountryCode(Map, String)		0 %		0 %	4	4	7	7	1	1
hasAnagrams(List)		0 %		0 %	5	5	9	9	1	1
subtractOf(Map, Map)		0 %		0 %	4	4	2	2	1	1
removeFillerWords(List, String[])		0 %		0 %	3	3	6	6	1	1
shoppingListCost(List, Map)		0 %		0 %	3	3	6	6	1	1
whoAreInBoth(List, List)		0 %		0 %	3	3	3	3	1	1
buildWordSet(List)		0 %		0 %	2	2	3	3	1	1
mergePhoneBooks(Map, Map)		100 %		100 %	0	4	0	8	0	1
containsIn(Map, Map)		100 %		100 %	0	4	0	4	0	1
Total	1 315 of 1 431	8 %	142 of 154	7 %	86	94	134	146	15	17

В генерируемом фаззинге покрытие для обеих функций по обеим метрикам составило 100%, что намного лучше классического фаззинга, правда и времени на написание этих тестов было потрачено в разы больше.

Тестовое покрытие осталось на том же уровне, что и в результатах исследования [3 лабораторной работы](#).

lesson 6

В `lesson6` тестировалось две функции: `fromRoman(String)` и `plusMinus(String)`.

Классический fuzzing

ParseKt

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
fromRoman(String)		9 %		15 %	14	17	42	45	0	1
dateStrToDigit(String)		0 %		0 %	9	9	27	27	1	1
dateDigitToStr(String)		0 %		0 %	7	7	25	25	1	1
flattenPhoneNumber(String)		0 %		0 %	14	14	24	24	1	1
computeDeviceCells$executeCommands(Ref.IntRef, Ref.IntRef, int, String, int, List)		0 %		0 %	13	13	20	20	1	1
plusMinus(String)		28 %		21 %	13	17	12	16	0	1
bestHighJump(String)		0 %		0 %	6	6	11	11	1	1
firstDuplicateIndex(String)		0 %		0 %	7	7	13	13	1	1
computeDeviceCells$findReversedNestedLoop(String, int)		0 %		0 %	6	6	20	20	1	1
mostExpensive(String)		0 %		0 %	5	5	10	10	1	1
computeDeviceCells$findNestedLoop(String, int)		0 %		0 %	5	5	15	15	1	1
computeDeviceCells$check(String)		0 %		0 %	8	8	10	10	1	1
bestLongJump(String)		0 %		0 %	4	4	10	10	1	1
computeDeviceCells(int, String, int)		0 %		0 %	4	4	7	7	1	1
timeSecondsToStr(int)		0 %		n/a	1	1	4	4	1	1
timeStrToSeconds(String)		0 %		0 %	2	2	6	6	1	1
safeToInt(String)		0 %		0 %	3	3	3	3	1	1
checkIfNotOk(String)		0 %		0 %	3	3	3	3	1	1
twoDigitStr(int)		0 %		0 %	4	4	1	1	1	1
Total	2 119 of 2 214	4 %	220 of 232	5 %	128	135	263	270	17	19

В функции `fromRoman(String)` покрытие по инструкциям составило ничтожные 9%, а по ветвям 15%. Покрытый код функции можно видеть ниже.

```
348. fun fromRoman(roman: String): Int {
349.     if (roman == "") return -1
350.     val ok = "IVXLCDM"
351.     for (ch in roman) if (ch !in ok) return -1
352. 
353.     var str = roman
354.     var res = 0
355. 
356.     while ("CM" in str) {
357.         res += 900
358.         str = str.substringBefore("CM") + str.substringAfter("CM")
359.     }
360.     while ("M" in str) {
361.         res += 1000
362.         str = str.substringBefore("M") + str.substringAfter("M")
363.     }
364.     while ("CD" in str) {
365.         res += 400
366.         str = str.substringBefore("CD") + str.substringAfter("CD")
367.     }
368.     while ("D" in str) {
369.         res += 500
370.         str = str.substringBefore("D") + str.substringAfter("D")
371.     }
372.     while ("XC" in str) {
373.         res += 90
374.         str = str.substringBefore("XC") + str.substringAfter("XC")
375.     }
376.     while ("C" in str) {
377.         res += 100
378.         str = str.substringBefore("C") + str.substringAfter("C")
379.     }
380.     while ("XL" in str) {
381.         res += 40
382.         str = str.substringBefore("XL") + str.substringAfter("XL")
383.     }
384.     while ("L" in str) {
385.         res += 50
386.         str = str.substringBefore("L") + str.substringAfter("L")
387.     }
388.     while ("IX" in str) {
389.         res += 9
390.         str = str.substringBefore("IX") + str.substringAfter("IX")
391.     }
392.     while ("X" in str) {
393.         res += 10
394.         str = str.substringBefore("X") + str.substringAfter("X")
395.     }
396.     while ("IV" in str) {
397.         res += 4
398.         str = str.substringBefore("IV") + str.substringAfter("IV")
399.     }
400.     while ("V" in str) {
401.         res += 5
402.         str = str.substringBefore("V") + str.substringAfter("V")
403.     }
404.     while ("I" in str) {
405.         res += 1
406.         str = str.substringBefore("I") + str.substringAfter("I")
407.     }
408. 
409.     return res
410. }
```

Видно, что дальше 351 строчки тестирование не зашло, так как ни одна сгенерированная строка не содержала только символы римской системы счисления.

В функции `plusMinus(String)` покрытие будет получше, однако всё равно далеко от идеала: 28% по инструкциям и 21% по ветвям.

```
fun plusMinus(expression: String): Int {
    if (expression.isEmpty()) throw IllegalArgumentException("Empty expression")
    val ok = "0123456789 "
    val ops = "+-"
    for (check in expression) if (check !in (ok + ops)) throw IllegalArgumentException("$check is not allowed in expression")
    if (" " !in expression) return safeToInt(expression)
    val commands = expression.split(" ")
    for (i in commands.indices) {
        if ((i % 2 == 0 && commands[i] in ops) || (i % 2 != 0 && commands[i] !in ops)) throw IllegalArgumentException("Bad expression format")
    }
    var res = commands[0].toInt()
    for (i in 1..(commands.size - 2) step 2) {
        val x = commands[i + 1].toInt()
        if (x < 0) throw IllegalArgumentException("Non-positive number in $expression")
        when (commands[i]) {
            "+" -> res += x
            "-" -> res -= x
        }
    }
    return res
}
```

Из рисунка видно, что проблема точно такая же, что и у `fromRoman(String)` - входная строка не содержала корректные для функции символы.

Generative random testing

ParseKt

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
dateStrToDigit(String)		0 %		0 %	9	9	27	27	1	1
dateDigitToStr(String)		0 %		0 %	7	7	25	25	1	1
flattenPhoneNumber(String)		0 %		0 %	14	14	24	24	1	1
computeDeviceCells$executeCommands(Ref.IntRef, Ref.IntRef, int, String, int, List)		0 %		0 %	13	13	20	20	1	1
bestHighJump(String)		0 %		0 %	6	6	11	11	1	1
firstDuplicateIndex(String)		0 %		0 %	7	7	13	13	1	1
computeDeviceCells$findReversedNestedLoop(String, int)		0 %		0 %	6	6	20	20	1	1
mostExpensive(String)		0 %		0 %	5	5	10	10	1	1
computeDeviceCells$findNestedLoop(String, int)		0 %		0 %	5	5	15	15	1	1
computeDeviceCells$check(String)		0 %		0 %	8	8	10	10	1	1
bestLongJump(String)		0 %		0 %	4	4	10	10	1	1
computeDeviceCells(int, String, int)		0 %		0 %	4	4	7	7	1	1
timeSecondsToStr(int)		0 %		n/a	1	1	4	4	1	1
timeStrToSeconds(String)		0 %		0 %	2	2	6	6	1	1
checkIfNotOk(String)		0 %		0 %	3	3	3	3	1	1
twoDigitStr(int)		0 %		0 %	4	4	1	1	1	1
fromRoman(String)		100 %		100 %	0	17	0	45	0	1
plusMinus(String)		100 %		93 %	2	17	0	16	0	1
safeToInt(String)		100 %		100 %	0	3	0	3	0	1
Total	1 587 of 2 214	28 %	166 of 232	28 %	100	135	206	270	16	19

Как и в предыдущей лабораторной работе тестовое покрытие функции `fromRoman(String)` достигло максимума, а `plusMinus(String)` немного не дотянул до идеала - лишь 93% покрытия по ветвям. Этот результат оказался на 3% лучше, чем покрытие тестами модульного тестирования.

```
fun plusMinus(expression: String): Int {
    if (expression.isEmpty()) throw IllegalArgumentException("Empty expression")
    val ok = "0123456789 "
    val ops = "+-"
    for (check in expression) if (check !in (ok + ops)) throw IllegalArgumentException("$check is not allowed in expression")
    if (" " !in expression) return safeToInt(expression)
    val commands = expression.split(" ")
    for (i in commands.indices) {
        if ((i % 2 == 0 && commands[i] in ops) || (i % 2 != 0 && commands[i] !in ops)) throw IllegalArgumentException("Bad expression format")
    }
    var res = commands[0].toInt()
    for (i in 1..(commands.size - 2) step 2) {
        val x = commands[i + 1].toInt()
        if (x < 0) throw IllegalArgumentException("Non-positive number in $expression")
        when (commands[i]) {
            "+" -> res += x
            "-" -> res -= x
        }
    }
    return res
}
```

По сравнению с результатами классического fuzzing'а практически всё окрашено в зелёный, а если сравнивать с покрытием из 3 лабораторной работы, то недостающее состояние на 277 строчке наконец было достигнуто. Скорее всего этого позволил достичь тест-кейс с пробелами вместо знаков операции, так как такой случай при написании модульных тестов я ранее не рассматривал. Однако мои догадки по поводу зависимости неполного покрытия 272 строчки от 277 полностью провалились, так как даже при полностью окрашенным в зелёный цвет теле цикла он всё равно остаётся не полностью покрытым с точки зрения ветвления. Что касается возможных причин неполного ветвления в этих двух циклах, то моя позиция остаётся неизменной - не могу предположить, что нужно сделать, чтобы эти две строчки кода считались полностью покрытыми.

lesson 7

В `lesson7` тестировалась лишь одна функция `countSubstrings(String, List)`.

Классический fuzzing

FilesKt

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
markdownToHtml(String, String)		0 %		0 %	42	42	95	95	1	1
markdownToHtmlSimple(String, String)		0 %		0 %	30	30	68	68	1	1
printDivisionProcess(int, int, String)		0 %		0 %	8	8	36	36	1	1
transliterate(String, Map, String)		0 %		0 %	12	12	22	22	1	1
markdownToHtmlLists(String, String)		0 %		0 %	21	21	37	37	1	1
top20Words(String)		0 %		0 %	11	11	19	19	1	1
sibilants$correct(String)		0 %		0 %	17	17	25	25	1	1
printMultiplicationProcess(int, int, String)		0 %		0 %	3	3	29	29	1	1
centerFile(String, String)		0 %		0 %	11	11	23	23	1	1
chooseLongestChaoticWord(String, String)		0 %		0 %	7	7	12	12	1	1
sibilants(String, String)		0 %		0 %	5	5	11	11	1	1
alignFile(String, int, String)		0 %		0 %	7	7	13	13	1	1
deleteMarked(String, String)		0 %		0 %	6	6	9	9	1	1
alignFile$append(Ref.IntRef, int, BufferedWriter, String)		0 %		0 %	3	3	9	9	1	1
checkLetters(String)		0 %		0 %	3	3	6	6	1	1
isMarkdownEmpty(String)		0 %		0 %	6	6	1	1	1	1
alignFileByWidth(String, String)		0 %		n/a	1	1	1	1	1	1
pop(List)		0 %		n/a	1	1	3	3	1	1
countSubstrings(String, List)		100 %		100 %	0	7	0	15	0	1
Total	3 628 of 3 773	3 %	352 of 364	3 %	194	201	419	434	18	19

В тестируемой функции классический fuzzing дал свои плоды - покрытие составило 100% по обеим метрикам. Значит можно сделать вывод, что был сгенерирован список, элемент которого содержался в текстовом файле.

Generative random testing

FilesKt

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
markdownToHtml(String, String)		0 %		0 %	42	42	95	95	1	1
markdownToHtmlSimple(String, String)		0 %		0 %	30	30	68	68	1	1
printDivisionProcess(int, int, String)		0 %		0 %	8	8	36	36	1	1
transliterate(String, Map, String)		0 %		0 %	12	12	22	22	1	1
markdownToHtmlLists(String, String)		0 %		0 %	21	21	37	37	1	1
top20Words(String)		0 %		0 %	11	11	19	19	1	1
sibilants$correct(String)		0 %		0 %	17	17	25	25	1	1
printMultiplicationProcess(int, int, String)		0 %		0 %	3	3	29	29	1	1
centerFile(String, String)		0 %		0 %	11	11	23	23	1	1
chooseLongestChaoticWord(String, String)		0 %		0 %	7	7	12	12	1	1
sibilants(String, String)		0 %		0 %	5	5	11	11	1	1
alignFile(String, int, String)		0 %		0 %	7	7	13	13	1	1
deleteMarked(String, String)		0 %		0 %	6	6	9	9	1	1
alignFile$append(Ref.IntRef, int, BufferedWriter, String)		0 %		0 %	3	3	9	9	1	1
checkLetters(String)		0 %		0 %	3	3	6	6	1	1
isMarkdownEmpty(String)		0 %		0 %	6	6	1	1	1	1
alignFileByWidth(String, String)		0 %		n/a	1	1	1	1	1	1
pop(List)		0 %		n/a	1	1	3	3	1	1
countSubstrings(String, List)		100 %		100 %	0	7	0	15	0	1
Total	3 628 of 3 773	3 %	352 of 364	3 %	194	201	419	434	18	19

Генерируемые входные данные также достигли 100% покрытия по обеим метрикам, как классический fuzzing, так и модульные тесты в 3 лабораторной работе.

Выводы

По проделанной работе можно сделать вывод, что fuzzer - это очень интересная штука. С одной стороны можно воспользоваться готовым генератором и просто подавать функции на вход генерируемые данные, а с другой можно самому написать генератор, который будет подавать на вход функции корректные данные. Думаю, что если классическому fuzzer'у дать всё время, что я потратил на разбирательство с библиотекой и написание тестов для Generative random testing (порядка 10-12 часов), то рандомные входные данные дали бы результаты не хуже, а может даже и лучше, чем Generative random testing.