ECE 322
Lab Report 5

Arun Woosaree
XXXXXXX

December 6, 2019

1 Introduction

The purpose of this lab was to serve as an introduction to integration testing. Integration testing is a logical extension of the previous lab, in which we did unit testing. This also allows us to build on the JUnit skills we learned in the previous lab. We were also introduced to the idea of mocking, using the Mockito library to mock out relationships between classes as we desire. Generally, there are two strategies to integration testing. The first is *Non-incremental* testing, also known as Big Bang testing. With this strategy, each module is tested individually, and then there is one final test where the entire system is tested as a whole. The second integration testing approach is *Incremental* testing, in which we combine the next module to be tested with the set of previously tested modules before running tests. This can be done in either a bottom up or top down approach. In this lab, a simple command-line database program written in Java was tested. There are 7 modules which make up the system. Module A is responsible for the GUI, B for opening, C for sorting, D for modifying, E for exiting, F for displaying, and G for updating data. Files in the database contain one entry per line, which are comma separated. Both testing strategies were used to test the application in this lab. Big bang testing was done in the first part, and we chose the Bottum-up incremental testing strategy for the second portion of the lab. Where possible, the tests were crafted to cover the full functionality of the program, including statement coverage. The exceptions are modules A and B, where it was not feasible to cover some lines. The case with Module A was that line 147 cannot be covered because when the program exits, an exception is thrown, which halts execution. With Module B, testing lines 39-42 when an IO Exception is caught does not need to be tested, because there is already code which handles the case when a file is not found. Furthermore, this situation was never encountered in our testing, due to the error handing when a file is not found. Additionally, the only code that gets run if this IO Exception somehow gets thrown are library functions (Printing to stdout, and printing a stacktrace of an error), and these library functions are likely already tested extensively, otherwise programmers would not trust them. The tests were

written using **junit-jupiter:5.5.2**, **mockito-core:3.1.0**, and were run using Java 13 with the command-line argument –**enable-preview**. A **build.gradle** is provided for ease of use, from which an IDE like Intellij or Eclipse should be able to install dependencies from and run the tests.

2 Task 1

For part one of this lab, a simple Array Helper library written in Java was tested using mutation testing. A Java library named PIT test was used to generate mutants of the ArrayLib class. Test cases were created which had 100% line coverage and branch coverage. These are outlined in the table which can be found in Appendix A The failing tests are explained as follows:

1. withoutTestRemoveTwo: This test fails because this method depends on ArrayLib's implementation of indexOf, which only returns the first occurrence of an element. This results in the method only removing one occurrence of a repeated element.

2. withoutTestRemoveFirstElement: This test fails because in the method, there is a check to see if $index > 0$, but it should be $index \geq 0$. Because it strictly checks for $index > 0$, the first element is never considered for removal.

3. intersectionTestDuplicate: This test fails because if there are elements that appear more than once in both arrays, then the the method will attempt to increment the index of the intersection array multiple times. However, this array is currently limited to the length of array a. To avoid this error, the size of the intersection array should be the length of array a plus the length of array b.

To do the mutation testing, the three failing tests above were commented out, since mutation testing has a prerequisite, which is that the test suite must be green. With the PIT tool, 37 mutants were created, and 36 were killed. (More on this later).

ArrayLib.java

Mutations

9	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
10	1. Replaced integer subtraction with addition → KILLED 2. Replaced integer subtraction with addition → KILLED
12	1. mutated return of Object value for ArrayLib::reverse to (if (x != null) null else throw new RuntimeException) → KILLED
21	1. negated conditional → KILLED
22	1. Changed increment from 1 to -1 → KILLED
26	1. mutated return of Object value for ArrayLib::unique to (if (x != null) null else throw new RuntimeException) → KILLED
34	1. Changed increment from 1 to -1 → KILLED
36	1. negated conditional → KILLED
37	1. Changed increment from 1 to -1 → KILLED
42	1. mutated return of Object value for ArrayLib::intersection to (if (x != null) null else throw new RuntimeException) → KILLED
49	1. Replaced integer addition with subtraction → KILLED
51	1. Changed increment from 1 to -1 → KILLED
54	1. Changed increment from 1 to -1 → KILLED
57	1. mutated return of Object value for ArrayLib::union to (if (x != null) null else throw new RuntimeException) → KILLED
65	1. negated conditional → KILLED
66	1. Changed increment from 1 to -1 → KILLED
69	1. Replaced integer subtraction with addition → KILLED
72	1. negated conditional → KILLED
73	1. Changed increment from 1 to -1 → KILLED
77	1. mutated return of Object value for ArrayLib::compact to (if (x != null) null else throw new RuntimeException) → KILLED
81	1. changed conditional boundary → KILLED 2. negated conditional → KILLED
82	1. negated conditional → KILLED 2. negated conditional → KILLED 3. negated conditional → KILLED
83	1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
86	1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
91	1. negated conditional → KILLED 2. negated conditional → KILLED 3. negated conditional → KILLED
92	1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
95	1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
104	1. changed conditional boundary → SURVIVED

	2. negated conditional → KILLED
108	1. mutated return of Object value for ArrayLib::without to (if (x != null) null else throw new RuntimeException) → KILLED

Active mutators

- CONDITIONALS_BOUNDARY_MUTATOR
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Tests examined

- ArrayLibTest.withoutTest() (5 ms)
- ArrayLibTest.reverseTest() (5 ms)
- ArrayLibTest.indexOfTest() (147 ms)
- ArrayLibTest.intersectionTest() (7 ms)
- ArrayLibTest.unionTest() (5 ms)
- ArrayLibTest.uniqueTest() (5 ms)

Now, for the one mutant that survived, it is actually directly related to a test failure mentioned above, which we commented out before running PITest. Specifically, it has to do with the if statement which checks if $index > 0$ instead of $index \geq 0$ on line 104.

Before the mutation testing, we found three bugs in the program. After the mutants were created and killed, only one of the bugs was highlihted. By the mutation test report. In this case, the mutation testing did not expose holes in the succeeding test cases, but this was mostly due to the luck of the tester writing good test cases. In other applications, mutation testing can and will help find holes in weak test suites. This is reinforced by the fact that after we commented out the failing tests, the mutation test caught one of the bugs because a mutant survived when the conditon on line 104 $index > 0$ was mutated.

I think in the real world mutation testing can be used as an indicator, a sort of 'litmus' test, if you will for the quality of a test suite. It by no means should be the only thing one checks for to evaluate the quality of a testing suite, but on its own, it can reveal some weaknessed in a test suite. Like any other testing strategy, it is not a silver bullet, and the right testing strategies should be used for the right situations. In general, mutation testing excels at checking boundary conditions, which can be extremely useful, since programmers often mix up conditions such as 'less than' or 'less than or equal to'.

3 Task 2

For part two in this lab, the database application was tested using the bottom up incremental testing strategy. Unit tests were made starting at the lower level modules, building our way up to the higher level modules so that nothing was mocked. The actual tests can be found in Appendix **??**, and are also in the zip archive in which this report was submitted. They can be found in *"Lab 5/Lab 5/src/test/java/bottumup"* The test results are found on the next page.

The failing tests are explained as follows: It should be noted that the tests were run in the order F, G, BF, CF, DFG, E, Everything.

3.1 Test CF

In general, the sorting code appears smelly. It is not using a library function when one is available for ArrayLists, and the increment variable may not be an integer number, even though it is being used to help calculate indicies to manipulate the data. Both of the errors below can be fixed by using the **Collections.sort()** method available in java, since **compareTo()** for the Entry class is already overridden.

3.1.1 sortFourElementsTest()

The application treats the sorting differently for some reason when there are four elements to sort, because it does a check for when the size of the collection

divided by 2 is equal to 2. In this case, it looks like with four elements, the application returns an unsorted ArrayList

```
Input:
[(ccc, ccc), (aaa, aaa), (bbb, ddd), (bbb, aaa)]
Expected :
[(aaa, aaa), (bbb, aaa), (bbb, ddd), (ccc, ccc)]
Actual :
[(ccc, ccc), (aaa, aaa), (bbb, ddd), (bbb, aaa)]
```

3.1.2 sortDataTest()

The application seems to mostly sort the data ok, but it forgets about sorting the first element in the data, and leaves it untouched.

```
Input    :[(ddd, aaa), (bbb, bbb), (ccc, ccc), (aaa, aaa), (ccc, aaa), (bbb, aaa)]
Expected :[(aaa, aaa), (bbb, aaa), (bbb, bbb), (ccc, aaa), (ccc, ccc), (ddd, aaa)]
Actual   :[(ddd, aaa), (aaa, aaa), (bbb, bbb), (bbb, aaa), (ccc, aaa), (ccc, ccc)]
```

3.2 Test DFG

3.2.1 updateDataTest()

The updateData function takes in an index, but then it adds one to the index. This results in a test failure when the test expects index 5, for example to be updated, the application actually ends up updating the entry at index 6. It is interesting how this unit test fails yet when testing everything at once, the update command in module A works as expected. Curiously, in module A, the index is subtracted by 2, so this is probably how the functionality ends up still working in the end.

```
Expected :
[(testName0, testNumber0),
 (testName1, testNumber1),
 (testName2, testNumber2),
 (testName3, testNumber3),
 (testName4, testNumber4),
 (testName, testNumber),
 (testName6, testNumber6),
 (testName7, testNumber7),
 (testName8, testNumber8),
 (testName9, testNumber9)]
Actual :
[(testName0,
 testNumber0),
 (testName1, testNumber1),
 (testName2, testNumber2),
```

```
(testName3, testNumber3),
(testName4, testNumber4),
(testName5, testNumber5),
(testName, testNumber),
(testName7, testNumber7),
(testName8, testNumber8),
(testName9, testNumber9)]
```

3.3 Test F

3.3.1 testModuleF()

When Module F displays data, it skips the first line, or the first entry. This is because the for loop on line 15 in Module F starts from $i = 1$, instead of $i = 0$, which means it starts from indexing the second element in the ArrayList, instead of the first element.

```
Expected:
Current Data:
1 name1, number1
2 name2, number2
3 name3, number3
4 name4, number4
5 name5, number5

Actual:
Current Data:
2 (name2, number2)
3 (name3, number3)
4 (name4, number4)
5 (name5, number5)
```

3.4 Test Everything

3.4.1 testSort()

This test fails because the file itself is not sorted. Instead, we have a case where Module A is calling Module C to sort the data, but Module A never tells another module to update the file with the sorted data, so the database file is left unmodified and therefore unsorted.

```
Input:
ddd,aaa
bbb,bbb
ccc,ccc
aaa,aaa
ccc,aaa
```

```
bbb,aaa
Expected :
aaa,aaa
bbb,aaa
bbb,bbb
ccc,aaa,
ccc,ccc
ddd,aaa
Actual :
ddd,aaa
bbb,bbb
ccc,ccc
aaa,aaa
ccc,aaa
bbb,aaa
```

3.4.2 testDeleteInvalidArguments()

The application is not equipped to handle arguments after the delete command which are non numeric. In such a case where a letter or word is inputted after the word 'delete', the program will ungracefully panic, instead of handling the error.

3.4.3 testUpdateInvalidArguments()

This test fails for the exact same reason for which testDeleteInvalidArguments() fails.

4 Conclusion

In this lab, we were introduced to integration testing. We tested a simple command-line database application using two integration testing strategies: non-incremental testing, and the bottom up incremental testing strategy.

In general, integration testing seems beneficial over testing individual modules. It allows us to verify that the system as a whole is working as expected. For example, we noticed that as a whole, functions like updateData worked as expected, even though they failed when testing the modules individually.

This type of testing might be tedious for a large scale system, but it would definitely be effective, since it would ensure that each module is communicating with other modules as expected. If the modules were tested individually for a large scale system without integration, we might find that the system as a whole does not work, even though the unit tests are passing. Furthermore, using integration testing strategies allows for the errors to be found and traced to the modules where the error is happening fairly easily. Imagine trying to debug a large scale system without knowing where the error is originating when two unknown modules communicate with each other.

Drivers and stubs are an effective method of isolating modules. They allow for a module which depends on other modules to be tested independently of the other modules it may depend on or communicate with. In other words, isolating the modules.

I think the bottom up strategy would be best for test-driven development. That way, we can start at the lower level modules which do not depend on anything, and work our way up. As we get to higher level modules, we have continuous integration of the newer code, which means the testing process is less painful. As the icing on top, we get validation that the system is working as intended every step of the way.

However, for testing software libraries, the big bang method might make more sense, since libraries are isolated from the code that a user writes. It would not make sense for the burden of testing to be passed on to the user of a library. The user expects the library to work when they use it, so in this way we end up with something resembling the big bang strategy, where the library and the users' code is tested individually, and then all together at once in the end.

I think that the The big bang strategy would be the easiest testing strategy to maintain over time since if an API changes with any module, we just need to update the tests for one module, and the one integration test that tests everything.

Personally, I like the bottom-up strategy for incremental testing, since no stubs are needed, so we know that the actual code is always being tested. Plus, each test builds off of the previous one, so if we encounter an error during this process, it is very easy to find where it is originating. Compared to big-bang testing, what if there is a situation where each module individually passes, but the one big integration test fails? Where would one even start to look for the error, and determine the offending modules?

A Part 1 Testing Results

	input	expected	actual	
reverseTest	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[9, 8, 7, 6, 5, 4, 3, 2, 1]	[9, 8, 7, 6, 5, 4, 3, 2, 1]	PASS
uniqueTest	[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, null]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	PASS
intersectionTest	a: [1, 2, 3, 4, 5, 6, 7, 8, 9] b: [3, 4, 5]	[3, 4, 5]	[3, 4, 5]	PASS
unionTest	a: [1, 2, 3, 4] b: [5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	PASS
indexOfTest	a: [1, 2, 3, 4, 5, 6, 7, 8, 9, null] b: 5	4	4	PASS
	a: [1, 2, 3, 4, 5, 6, 7, 8, 9, null] b: null	10	10	PASS
withoutTest	array: [1,2,3,4, 4,5,6,7,8,9], remove: [3, 4, 5, 10]	[1, 2, 6, 7, 8, 9]	[1, 2, 6, 7, 8, 9]	PASS
withoutTestRemoveTwo	array: [1,2,3,4,5,6,7,8,9], remove: [3, 4, 4, 5, 10]	[1, 2, 6, 7, 8, 9]	[1, 2, 4, 6, 7, 8, 9]	FAIL
withoutTestRemoveFirstElement	array: [1,2,3,4,5,6,7,8,9], remove: [1, 3, 4, 5]	[2, 6, 7, 8, 9]	[1, 2, 6, 7, 8, 9]	FAIL
intersectionTestDuplicate	a: [1, 2, 2, 3, 4] b: [2, 2, 3, 4]	[2, 2, 3, 4]	IndexOutOfBoundsException	FAIL

B Part 2 Testing Results

	Input	Expected	Actual - MathPackage.java	Actual - Commit.java	Actual - Fixed.java	MathPackage.java	Commit.java	Fixed.java
randomTest:	Generate 1000 arrays using the random function	all values contained within [a,b]	all values contained within [a,b]	all values contained within [a,b]	all values contained within [a,b]	PASS	PASS	PASS
maxTest	Generate 1000 arrays, sort them and compare the last element with the return value of max	last element of array == max	last element of array == max	last element of array == max	last element of array == max	PASS	PASS	PASS
minTest	Generate 1000 arrays, sort them and compare the first element with the return value of min	first element of array == min	first element of array == min	first element of array == min	first element of array == min	PASS	PASS	PASS
normalizeTest	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]	[0, -0.1, -0.2, -0.3, -0.4, -0.5, -0.6, -0.7, -0.8, -0.9, -1.0]	[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]	[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]	FAIL	PASS	PASS
	[0, 33, 66, 100]	[0, 0.33, 0.66, 1.0]	[0, -0.33, -0.66, -1.0]	[0, 0.33, 0.66, 1.0]	[0, 0.33, 0.66, 1.0]	FAIL	PASS	PASS
sumTest	Generate 1000 arrays, compare the sum against Java's built in method for calculating sum	sum	sum	sum	sum	PASS	PASS	PASS
stddev	[0.8448535275473217, 0.7356655820407797, 0.9431584337138527, 0.9885039386933584, 0.26752295318703023, 0.30118478380150293, 0.5614795756611817, 0.27410185661451103, 0.3727308526619961, 0.9350197461150006, 0.3430515987014403, 0.14325655442623153, 0.09474732578722389, 0.8118925398379901, 0.7889524243808093, 0.6174482765472112, 0.08282552694523304, 0.9906861613488818, 0.1006718689797883, 0.07076982982755198]	0.33383429942515 +/- 1e-10	0.33383429942515 +/- 1e-11	0.33383429942515 +/- 1e-12	0.33383429942515 +/- 1e-13	FAIL	FAIL	FAIL
	[24.351879794282766, -1.2369080960006045, 21.33618933269294, 38.40700888177969, 81.07224984081901, 14.773395426428706, -23.465737641671126, -29.74766797865162, -80.53273274920562, 82.69835022556401, 16.765612733609586, 85.27325420746189, -25.56842729615137, -85.17577250342885, 91.14866661682103, 47.56747687575623, 71.62517118559902, 26.263337092658247, -78.61956099645082, -18.809917650844213, 31.415431556119216, 15.021802528535659, -45.50777659561043, 76.15463305868838, 71.73129701754698, 42.00719896644702, 98.93219072662015, 29.911361379866946, -7.304486822031862, 72.46129116466469, 25.17287827553399, 6.610331737331364, 34.267027445622745, 10.796643684552663, -44.6877977606748, 99.08890435845663, 18.379318834519594, 54.01219466707613, -11.329038630356479, -23.230019858073646, 33.03020694951988, -59.205375268977974, -8.495912401158435, 83.71104890003406, 80.19371195739095, 70.17088288687546, -27.756415190854483, 67.22645204084466, 2.237307845125258, 90.69398038338696, 13.575182881518089, 66.0876889384611, -30.1366615029317, -20.084766272685243, 10.62213430500914, 70.12939720918465, -53.19292944066913, -84.43071331549132, -53.30861375844398, 36.75691360838863, 4.481177488201666, 89.23026437856967, 28.481208229172694, 4.041593290473799, -19.144188062781325, 71.48911820023875, -97.3407210733165, -82.95697886617639, -52.019808046281035, -20.500795201979386, -65.75240723052059, -76.4564498827019, -48.255740133965006, 20.75249609534157, -45.919570907416606, 55.70989509490502, -30.432532466301396, -51.161171652519121, -7.213072379759879, 62.480197271840865, -14.266598956238383, -14.496858737726953, -6.199152015512155, 86.73375309074737, 24.576298742407715, -50.335938636762336, -98.21921344917936, -97.29469852961215, 92.13635661651074, -6.363254915974139, 34.622868801612384, -47.36683311550864, -57.59764296582546, 53.28248250936096, 26.007936605144735, -86.505212849109, -7.68902146399742, -18.99542397736265, -5.991562076158388, 82.91883854718108]	54.00891607	54.00891607	54.00891607	54.00891607	PASS	PASS	PASS
arrayAddTestSameLength	Generate 1000 * 2 arrays with the same length. Compare the result against an alternative way to sum the arrays using Java 8 lambdas	element-wise sum	element-wise sum	element-wise sum	element-wise sum	PASS	PASS	PASS
arrayAddDifferentLength	d1: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] d2: [1, 2, 3, 4, 5, 6]	Assertion error different lengths	[11, 11, 11, 11, 11, 11]	[11, 11, 11, 11, 11, 11]	Assertion error different lengths	FAIL	FAIL	PASS
	d1: [1, 2, 3, 4, 5, 6] d2: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	Assertion error different lengths	IndexOutOfBoundsException	IndexOutOfBoundsException	Assertion error different lengths	FAIL	FAIL	PASS
negateTest	Generate 1000 arrays, and negate them, compare against return result	array negation	array negation	array negation, but 1 is also subtracted from each element	array negation	PASS	FAIL	PASS

arraySubtractTestSameLength	Generate 1000 * 2 arrays with the same length. Compare the result against an alternative way to subtract the arrays using Java 8 lambdas	element-wise subtraction	n/a	element-wise subtraction	element-wise subtraction		PASS	PASS
arraySubtractTestDifferentLength	d1: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] d2: [1, 2, 3, 4, 5, 6]	Assertion error different lengths	n/a	[-9, -7, -5, -3, -1, 1]	Assertion error different lengths		FAIL	PASS
	d1: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] d2: [1, 2, 3, 4, 5, 6]	Assertion error different lengths	n/a	[9, 7, 5, 3, 1, -1]	Assertion error different lengths		FAIL	PASS
distanceTest	d1 or d2 are not of length 2	Assertion error different lengths	n/a	0	Assertion error different lengths		FAIL	PASS
	d1: [-1, -2] d2: [3, 4]	7.211102551	n/a	0	7.211102551		FAIL	PASS
	d1: [6, 7] d2: [-8,-9]	21.26029163	n/a	0	21.26029163		FAIL	PASS
arrayDeviationTest	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	[-4.5, -3.5, -2.5, -1.5, -0.5, 0.5, 1.5, 2.5, 3.5, 4.5]	n/a	null	[-4.5, -3.5, -2.5, -1.5, -0.5, 0.5, 1.5, 2.5, 3.5, 4.5]		FAIL	PASS
	[4, 8, 1, 3, 9, 5, 10, 2, 7, 6]	[-1.5, 2.5, -4.5, -2.5, 3.5, -0.5, 4.5, -3.5, 1.5, 0.5]	n/a	null	[-1.5, 2.5, -4.5, -2.5, 3.5, -0.5, 4.5, -3.5, 1.5, 0.5]		FAIL	PASS
	input	expected	actual					
reverseTest	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[9, 8, 7, 6, 5, 4, 3, 2, 1]	[9, 8, 7, 6, 5, 4, 3, 2, 1]	PASS				
uniqueTest	[1, 1, 2, 3, 4, 5, 6, 7, 8, 9, null]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	PASS				
intersectionTest	a: [1, 2, 3, 4, 5, 6, 7, 8, 9] b: [3, 4, 5]	[3, 4, 5]	[3, 4, 5]	PASS				
unionTest	a: [1, 2, 3, 4] b: [5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	[1, 2, 3, 4, 5, 6, 7, 8, 9]	PASS				
indexOfTest	a: [1, 2, 3, 4, 5, 6, 7, 8, 9, null] b: 5	4	4	PASS				
	a: [1, 2, 3, 4, 5, 6, 7, 8, 9, null] b: null	10	10	PASS				
withoutTest	array: [1,2,3,4, 4,5,6,7,8,9], remove: [3, 4, 5, 10]	[1, 2, 6, 7, 8, 9]	[1, 2, 6, 7, 8, 9]	PASS				
withoutTestRemoveTwo	array: [1,2,3,4,5,6,7,8,9], remove: [3, 4, 4, 5, 10]	[1, 2, 6, 7, 8, 9]	[1, 2, 4, 6, 7, 8, 9]	FAIL				
withoutTestRemoveFirstElement	array: [1,2,3,4,5,6,7,8,9], remove: [1, 3, 4, 5]	[2, 6, 7, 8, 9]	[1, 2, 6, 7, 8, 9]	FAIL				
intersectionTestDuplicate	a: [1, 2, 2, 3, 4] b: [2, 2, 3, 4]	[2, 2, 3, 4]	IndexOutOfBoundsException	FAIL				