UNIVERSIDADE D
COIMBRA

Master in Computer Engineering
Software Quality and Reliability

Dynamic Software Testing

Assignment II

Gonçalo T. A. F. Ferreira, tferreira@student.dei.uc.pt, 2019214765
João F. G. Artur, joaoartur@student.dei.uc.pt, 2019217853

May 2023

Contents

1 Introduction

In current days, most of the software produced industrially goes through what is known as the Software Testing Life Cycle (STLC), which is the procedure of testing different aspects of the software in order to improve its overall quality. One of these stages of STLC is Dynamic Software Testing, the focus of this assignment.

The goals for this assignment are to develop a software test plan and to perform the required tasks to achieve the defined objectives. Additionally, the assignment has an exploratory nature, which creates the necessity of understanding the existing testing methods and tools.

The selected piece of code is a Rice Encoder that belongs to a library for decoding and encoding Free Lossless Audio Codec (FLAC) files [1], distributed in the Java programming language. This algorithm is one of the methods available in the library for the compression and encoding. According to the documentation, this encoder estimates the encoded size of a vector of residuals, and also performs the encoding to an output stream.

The plan defined for this assignment is split into two techniques, White Box and Black Box testing. The former uses the source code and its understanding as references for test selection and adequacy. In contrast, the latter assumes there's no internal knowledge of the software, thus using the functional specification as the guiding point.

This report follows the structure guidelines defined in the suggested template, based on the international standard ISO/IEC/IEEE 29119-3 and the IEEE 829 standard for software and system test documentation. Therefore, the first section introduces the context and the testing plan. Section 2 describes any issues related to software risk, while sections 3 and 4 identify which items are covered in the plan and those that aren't. Section 5 presents the approach to follow during the testing phase. The passing criteria are established in section 6 and the testing deliverables constitute section 7. The environment and the tasks performed by each of the members of the group are described in sections 8 and 9, respectively. Lastly, section 10 reports a summary of the tests executed and the obtained results.

2 Software Risk Issues

One of the first steps of the process of defining a testing plan is to assess any risks inherent to the selected software. Understanding the associated risks helps not only to address any difficulties which might arise later, but also to identify limitations of the defined tests.

2.1 Extreme complexity

The code selected was compromised of methods whose complexity varied from relatively simple to extremely hard to understand. One of the chosen methods is considered as a warm-up, due to it simplicity, while the other one has a higher complexity, having required a bit more effort.

2.2 Poor documentation

The documentation of the code is very scarce and poor, giving almost no information about the ideas subjacent to the method. Therefore, in order to understand the code, it was needed a manual analysis of the code and discussion between the members of the team to confirm the ideas extracted from the code review.

2.3 Methods hard to test

One of the methods doesn't return any value, rendering it impossible to directly test its output. So, a workaround for this difficulty was to evaluate the result of the write operation and compare the content of the file with an expected value. However, with this solution another challenge that arose was that it was needed to invoke a third method to allow the writing into the file.

3 Items and feature to be tested

From the Rice Encoder only two of its internal methods were chosen for testing purposes:

- *writeRiceSignedInt* - writes a signed Rice integer in a stream;
- *computeBestSizeAndParam* - calculates the number of bits needed to encode the sequence of values;

4 Items and features not to be tested

All the other methods of Rice Encoder were not selected to be part of the test covered. Some of them were similar in terms of structure and complexity to one of the methods already chosen, so it was of little value, in terms of knowledge and experience gained, to test these methods. The remaining methods were deemed too complex for the kind of tests selected and the learning goals of this assignment. The rest of the library is considered to be out of the scope.

5 Testing Approach

As previously stated in sections 3 and 4, the selected testing units were the methods *writeRiceSignedInt* and *computeBestSizeAndParam*. These were subjected to two different kinds of testing approaches: White Box Testing and Black Box Testing. Both of these testing techniques were backed-up by the JUnit Framework for tests implementation.

White Box Testing is a testing approach that requires the analysis of the code's intricacies, aiming to test the control and data structures used in the program. Due to this factor, it can be split into two subsections: Control Flow Testing and Data Flow Testing.

The class resources/slides refer to Control Flow testing as a strategy that builds a model around the program's control flow. The idea behind this testing strategy is to identify a set of paths through the program. These are then used to achieve a measure of testing coverage of the code. In order to identify and visually represent these paths, a Control Flow Graph (CFG) was created with the aid of the online tool *draw.io*. This allowed the identification of basic code blocks, decision blocks (points of divergence) and junctions (points of convergence). Once the CFG was complete, the next step was to calculate McCabe's Cyclomatic Complexity. This is essential as it is directly related to the number of independent paths found within the graph. Then, the next logical step was to generate test cases that promoted the coverage of each path, as this allowed every possible alternative to be tested.

Regarding Data Flow testing, the main idea is to verify and study the variances in the method's variables, how they change throughout the code's execution and their final value. The overall testing procedure was similar to the Control Flow testing, as the beginning was to create a Data Flow Graph (DFG). This particular type of graph is split into two major elements, nodes, and edges. The former is associated to definitions and computation uses (c-use), while the latter refers to predicate uses (p-use). Once the graph was drawn, the ADUP (All du-paths) were selected as the testing criteria, and it was identified every path present for every variable. Similarly to the previous testing method, these paths were derived to find suitable test cases.

Black Box Testing, although equally important, represents a very different testing approach. Unlike the previous technique, this procedure assumes there is no internal knowledge about the code in question, analysing only its external behaviour and functional specification, disregarding the program structure and focusing primarily on the information domain [3]. This testing method was also split into two phases, these being Equivalence Class Partitioning and Boundary Value Analysis. The first phase permitted the splitting of the test cases and remove any redundant ones, meaning that if two cases are expected to be processed the same way by the program, there's only need to test one of them [3]. The Boundary Value analysis focuses on selecting values on the borders of the test case spectrum. These are usually most likely to cause the program to fail, hence the need to select these values rather than just test randomly.

6 Item Pass/Fail Criteria

This section aims to identify the pass and fail criteria for the following tests on the Rice Encoder. These criteria were split into 3 main scenarios:

- Error Detection
  - A test case is given a pass grade if every error is identified as an error and doesn't go unnoticed.
- Stream Value Computation
  - A test is classified as a pass if the stream output is the same as the expected value, although an accurate prediction might be difficult to make due to the complexity of the code
- Impossible Prediction
  - Due to the sheer size of some data being analysed, namely array data types, it might be impossible to accurately predict the final outcome of a method when such large scale variables are at play. In this scenario, a generic description regarding the final output is given and if the actual output fits the description, a passing grade is given.

7 Test Deliverables

This section describes in detail all the tests and accessory information generated during the completion of the assignment. The initial part of the section covers white box testing, while the next part is related to black box testing.

7.1 White Box

7.1.1 Control Flow Testing

As mentioned before, two methods were chosen to be subject to this kind of testing technique. In each case, the Control Flow Graph and visual identification of each node in the code, the set of independent paths, and the test cases will be provided.

7.1.1.1 writeRiceSignedInt

The implementation of this method is presented in figure 1, where the nodes of the CFG are identified. The Control Flow Graph of the method is represented in figure 2.

```
public static void writeRiceSignedInt(long val, int param,
    BitOutputStream out) throws AssertionError, IOException {          0

    assert 0 <= param && param <= 31 && out != null;                   1
    assert (val >> 52) == 0 || (val >> 52) == -1;                      2

    long unsigned = val >= 0 ? val << 1 : ((-val) << 1) - 1;           3
    int unary = (int) (unsigned >>> param);
    for (int i = 0; i < unary; i++)                                    4, 7 & 5
        out.writeInt(1, 0);                                            5
    out.writeInt(1, 1);
    out.writeInt(param, (int) unsigned);                               8
}
```

Figure 1: Identification of the graph nodes in the code

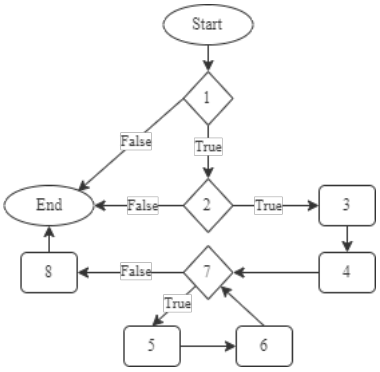

Figure 2: Control Flow Graph

The value of McCabe's cyclomatic complexity for this method can be calculated using one of the following formulas:

- $V(G) = P + 1$
- $V(G) = E - N + 2$

By counting the number of predicate nodes of the graph, one reaches the conclusion that this method has a V(G) complexity of 4. Alternatively, the formula that uses the number of nodes and the number of control flow edges gives the same result. The attained value means this method can be considered simple to understand, and, most importantly, there are at most 4 independent paths. The next list contains the paths found.

- P1: 0, 1 9
  - Possible
  - Result of triggering the assertion in node 1.
- P2: 0, 1, 2, 9
  - Possible
  - The assertion in node 2 is triggered.
- P3: 0, 1, 2, 3, 4, (7, 5, 6), 7, 8
  - Possible
  - The execution flow possibly enters the loop in node 7 and reaches successfully the end of the program.

All the test cases that were generated can be found in table 1. Each test is associated with only a path and is given a description of the input, the expected output, the actual output, and any comments deemed relevant.

ID	Path	Input	Expected Output	Output	Comments
1	P1	val = 100 param = 10 out = null	Assertion error	Assertion error	Assertion in node 1
2	P2	val = 9007199254740993 param = 10 out = valid BitOutputStream	Assertion error	Assertion error	Assertion due to $val >> 52 = 2$
3	P3	val = 100 param = 10 out = valid BitOutputStream	99 00	99 00	N/A

Table 1: Test cases

The first test was designed to trigger the assertion present in node 1 of the CFG. Similarly, the second case was created to achieve the same result for the assertion in node 2. The last test case is meant to verify if the method is well executed and if the result corresponds to what is expected.

7.1.1.2 computeBestSizeAndParam

Alike the previous section, the first image, 3, corresponds to the method's implementation where the nodes of the CFG are identified. Figure 4 displays the Control Flow Graph for this method.

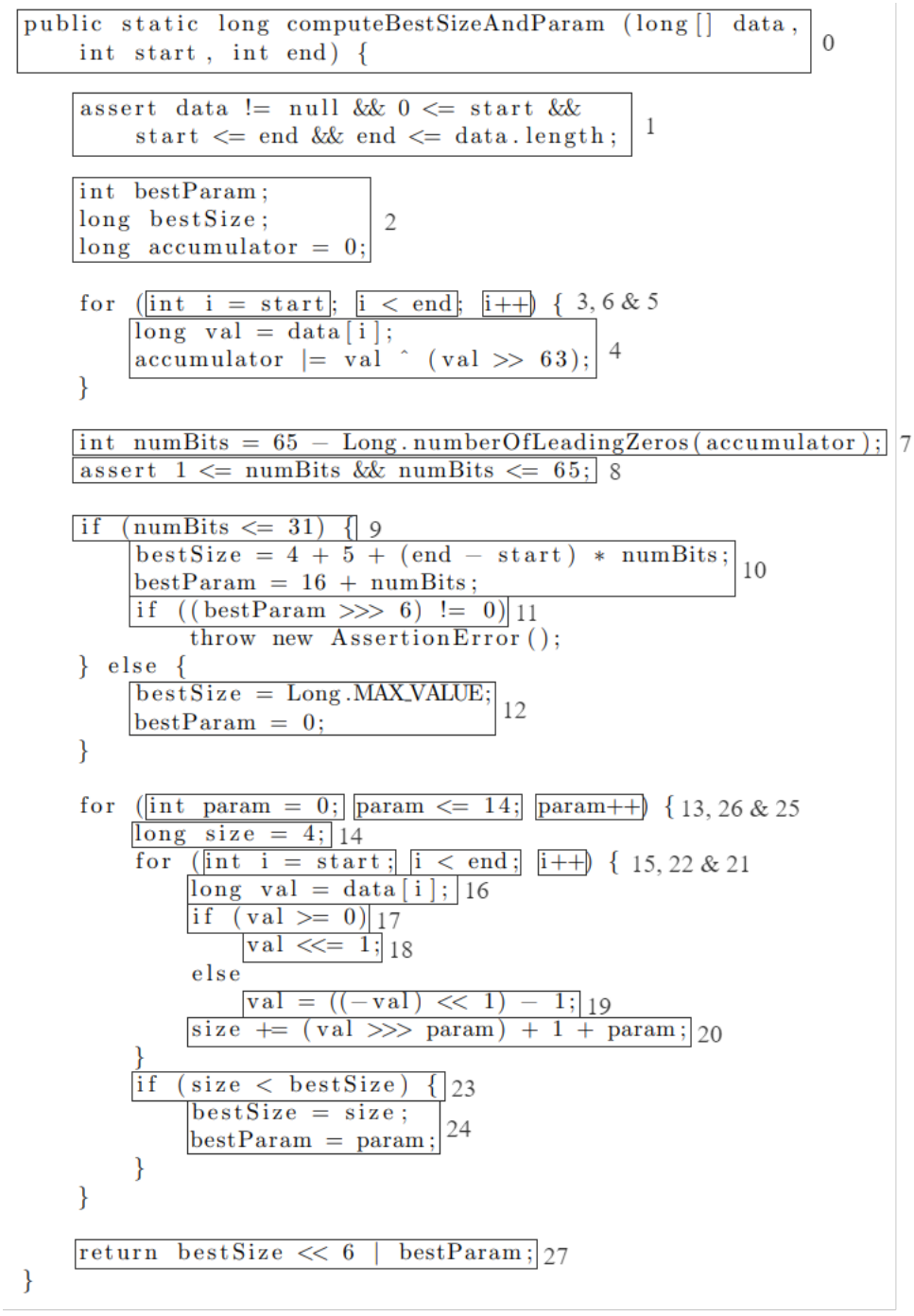

Figure 3: Identification of the graph nodes in the code

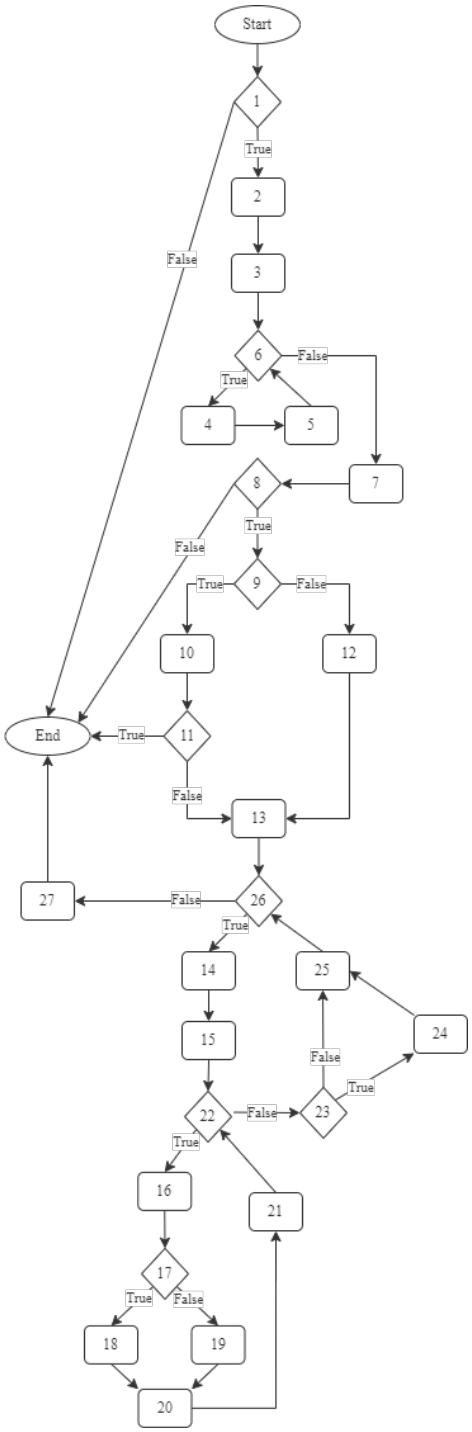

Figure 4: Control Flow Graph

Regarding McCabe's cyclomatic complexity, *computeBestSizeAndParam* has a V(G) complexity of level 10. Therefore, the set of independent paths is expected to have a maximum size of ten paths. Furthermore, this level of complexity also means that understanding this method requires substantial effort. If the value was higher than this, one recommendation would be to redo the implementation in order to try to lower the complexity.

The set of independent paths contains:

- P1: 0, 1, 28
  - Possible
  - Result of triggering the assertion in node 1.
- P2: 0, 1, 2, 3, (6, 4, 5), 6, 7, 8, 9, 10, 11, 28
  - Impossible
  - The condition in node 10 would have to be true. For this to happen, the variable *bestParam* would need to be negative or least 64. In the first case, *numBits* must have a value of -17 or less, but this can never happen due to the assertion in node 8. For the second alternative, *numBits* must be at least 48, but neither can this be true because of the condition in node 9.
- P3: 0, 1, 2, 3, (6, 4, 5), 6, 7, 8, 28
  - Impossible
  - Assertion in node 8 is triggered, since variable *numBits* is less than 1 or bigger than 65. However, both cases are invalid. The extreme values for this variable are either 1 or 65, because *accumulator* is a variable of type *long* (in Java is represented using 64 bits – 8 bytes). This means the maximum value of number of leading zeros is 64 and the minimum value is 0.
- P4: 0, 1, 2, 3, (6, 4, 5), 6, 7, 8, 9, 10, 11, 13, 26, 27, 28
  - Impossible
  - The program does not enter the loop around node 26. However, it's impossible for this loop to not happen, since the number of iterations is defined explicitly defined in the code as 14.
- P5: 0, 1, 2, 3, (6, 4, 5), 6, 7, 8, 9, 12, 13, 26, 27, 28
  - Impossible
  - The motive is the same as the previous path.

- P6: 0, 1, 2, 3, (6, 4, 5), 6, 7, 8, 9, 10, 11, 13, (26, 14, 15, (22, 16, 17, 18, 20, 21), 22, 23, 24, 25), 26, 27, 28
  - Possible
  - The path enters inside the cycles in node 26 and in node 22, respectively. Within the latter, the condition in node 17 is true, therefore the execution flow goes through node 18. The condition in node 23 is also true, so node 24 is included in the path.
- P7: 0, 1, 2, 3, (6, 4, 5), 6, 7, 8, 9, 10, 11, 13, (26, 14, 15, (22, 16, 17, 18, 20, 21), 22, 23, 25), 26, 27, 28
  - Possible
  - The difference between this path and P6 is that the condition in node 23 is false, so the flow goes from node 23 directly to node 25.
- P8: 0, 1, 2, 3, (6, 4, 5), 6, 7, 8, 9, 10, 11, 13, (26, 14, 15, (22, 16, 17, 19, 20, 21), 22, 23, 24, 25), 26, 27, 28
  - Possible
  - This path is similar to P6, with the difference being that the condition in node 17 is false. Thus, the execution flow passes through node 19 instead of node 18.
- P9: 0, 1, 2, 3, (6, 4, 5), 6, 7, 8, 9, 10, 11, 13, (26, 14, 15, (22, 16, 17, 19, 20, 21), 22, 23, 25), 26, 27, 28
  - Possible
  - The condition in node 23 is not verified, thus the flow goes from node 23 to 25, without passing in node 24. The remaining nodes are the same from path P8.

The test cases for the valid paths can be found in table 2. For the other paths no test case was defined since the paths were impossible to be covered.

ID	Path	Input	Expected Output	Output	Comments
4	P1	data = null start = 0 end = 0	Assertion error	Assertion error	Assertion in node 1
5	P6	data = [1] start = 0 end = 1	448	448	N/A
6	P7	data = [10, 20] start = 0 end = 2	1092	1092	N/A
7	P8	data = [-1] start = 0 end = 1	384	384	N/A
8	P9	data = [-10, -5] start = 0 end = 2	963	963	N/A

Table 2: Test cases

The first test triggers the assertion in node 1. Both test cases 5 and 6 were designed such that the condition in node 17 is verified, however the assessment of the condition in node 23 is true for the first case and is false for the second one. The last two test cases follow a very similar logic of the test cases for P6 and P7, but the condition in node 18 does not hold true.

7.1.2 Data Flow Testing

In contract to the previous technique, this section will only cover one of the chosen methods, *writeSignedRiceInt*. The reason for this is that the other method, *computeBestSizeAndParam*, was considered too complex to analyze without proper automatic testing tools, requiring a full manual process.

Similarly to the structure of the previous section, the Data Flow Graph as well as a set of paths along with the test cases for each variable will be provided.

7.1.2.1 writeRiceSignedInt

The Data Flow Graph of the method *writeRiceSignedInt* is shown in image 5. The pieces of code that take part of the graph are explicitly identified, so the implementation of this method is not exhibited here.

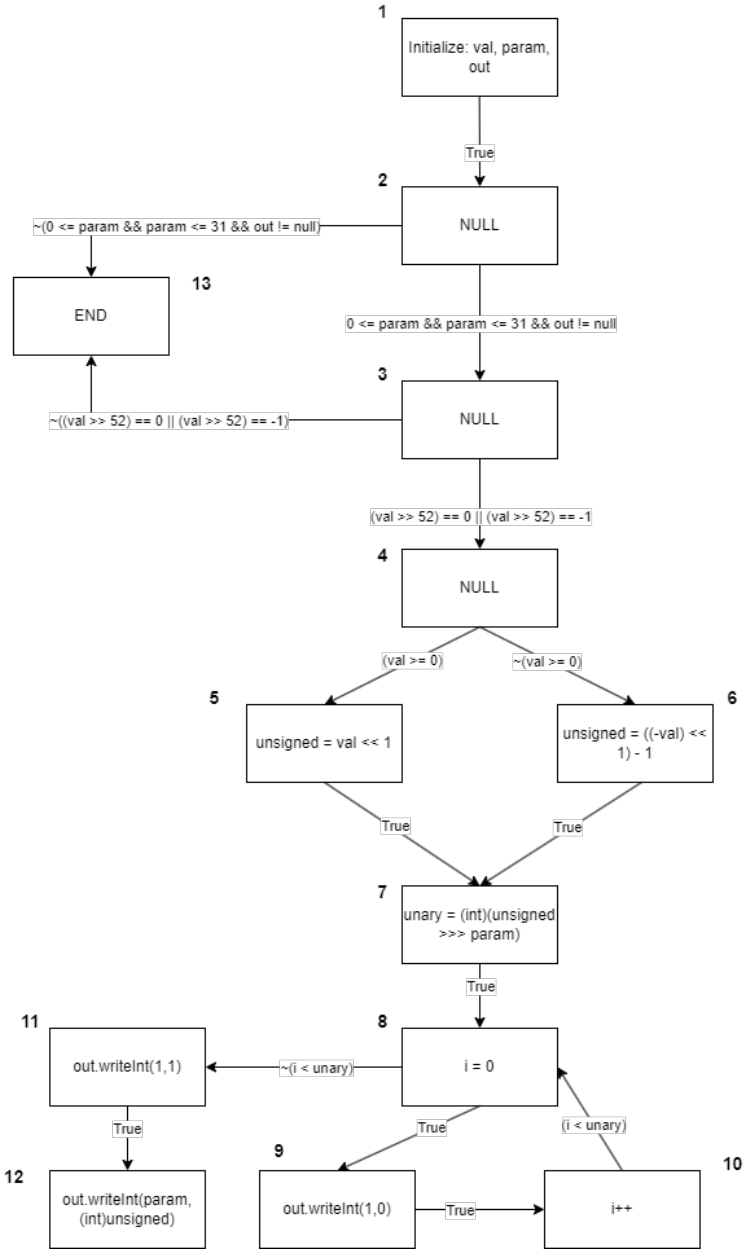

Figure 5: Data Flow Graph

The list below presents the variables identified in the code, along with the paths found for each one of the variables. Some paths are repeated, thus have the same identifier.

- param
  - P1: 1, 2, 13
  - P2: 1,2,3,4,5,7,8,11,12
- val
  - P2: 1,2,3,4,5,7,8,11,12
  - P3: 1,2,3,13
  - P4: 1,2,3,4,6,7,8,11,12
- out
  - P1: 1, 2, 13
  - P5: 1,2,3,4,5,7,8
  - P6: 1,2,3,4,5,7,8,9,10,8,11,12
- unary
  - P6: 1,2,3,4,5,7,8,9,10,8,11,12
- unsigned
  - P2: 1,2,3,4,5,7,8,11,12
  - P4: 1,2,3,4,6,7,8,11,12
- i
  - P6: 1,2,3,4,5,7,8,9,10,8,11,12

The table 3 contains all the test cases defined. It's worth mentioning that since some paths are repeated across different variables, a test case can cover multiple variables.

ID	Variable(s)	Path	Input	Expected Output	Output	Comments
9	param	P1	val = 100 param = -1 out = valid BitOutputStream	Assertion error	Assertion error	Assertion due to $param < 0$
10	param	P1	val = 100 param = 32 out = valid BitOutputStream	Assertion error	Assertion error	Assertion due to $param > 32$
11	param val unsigned	P2	val = 100 param = 10 out = valid BitOutputStream	99 00	99 00	N/A
12	val	P3	val = 9007199254740993 param = 10 out = valid BitOutputStream	Assertion error	Assertion error	Assertion due to $val >> 52 = 2$
13	val unsigned	P4	val = -100 param = 10 out = valid BitOutputStream	98 E0	98 E0	N/A
14	out	P1	val = 100 param = -1 out = null	Assertion error	Assertion error	Assertion due to $out = null$
15	out	P5	val = 100 param = 10 out = closed BitOutputStream	Blank write	Blank write	Writes are lost
16	out unary i	P6	val = 1024 param = 10 out = valid BitOutputStream	3000	2000	A 12-bits number is written using only 10 bits

Table 3: Test cases

7.2 Black Box

For each method, the first step was to define the equivalence classes and classify as valid or invalid. After that, a test case was created per class, so only weak equivalence is considered.

7.2.1 writeRiceSignedInt

In order to define the class partitions it's needed first to inspect the parameters of the method. In this case, the parameters are: i) val - long; ii) param - int; iii) out - BitOutputStream. The table 7 describes the different classes identified for each of the parameters.

Input	**Valid Equivalence Classes**	**Invalid Equivalence Classes**
val	$[-(2^{52}-1), 2^{52}-1]$	$< -(2^{52}-1)$ $> 2^{52}-1$ Malformed numbers Non-numeric strings Empty value
param	$[0, 31]$	< 0 > 31 Malformed numbers Non-numeric strings Empty value
out	valid bitOutputStream	null closed bitOutputStream

Table 4: Equivalence classes

Despite black box testing assuming no internal knowledge and that the tests are defined in function of the functional definition, there was no such information for this method. Thus, the valid equivalence classes for the input variables *val* and *param* were defined based on the knowledges of the method's implementation. In the case of invalid equivalence classes, any numerical value outside the valid range is considered invalid and other types of input that are not numerical are also included. For the input variable *out*, only a valid bitOutputStream is considered a valid class, while remaining classes are invalid.

Based on the equivalence classes in table 7, the next step to define boundary cases for each of the classes. The results can be found in table 5.

Input	**Boundary cases**
val	1048575, 1048576 0
param	-1, 0 31, 32

Table 5: Boundary analysis

The cases depicted in table 5 illustrate the boundary cases for the black box analysis. In particular, the *val* variable can assume 2 distinct border values (1048575 and 1048576). These are boundary values due to the shift right operation in the method, implying that all values above 1048576 lead to an Assert Exception. Regarding the *param* variable, the boundary cases are simpler to identify. The variable must have values between 0 and 31 in order to not cause an Assertion Exception. Therefore, the values -1, 0, 31 and 32 are perfect candidates for boundary values to be used in test cases.

In table 6 it's possible to find all the test cases. These were generated based on the knowledge of the equivalence partitioning and the boundary analysis techniques applied to this method.

ID	Input	Expected Output	Output	Comments
17	val = 1048575 param = 15	0000000000000001FFFC	0000000000000001FFFC	N/A
18	val = 1048576 param = 15	Assertion error	0000000000000008000	Expected assertion error $1048576 \gg 52 == 1$
19	val = 0 param = 15	8000	8000	N/A
20	val = 10 param = -1	Assertion error	Assertion error	N/A
21	val = 10 param = 0	000008	000008	N/A
22	val = 10 param = 31	80000014	80000014	N/A
23	val = 10 param = -1	Assertion error	Assertion error	N/A

Table 6: Test cases

7.2.2 computeBestSizeAndParam

The parameters of the method *computeBestSizeAndParam* are: i) data - long array; ii) start - int; iii) end - int. Unlike the previous section, it was possible to define the equivalence classes without resorting to the internal knowledge gained during the previous sections. All the equivalence classes are described in table 7.

Input	**Valid Equivalence Classes**	**Invalid Equivalence Classes**
data	ordered array	null Invalid elements: [10, a, #] Invalid array: +, 129, %
start	$[0, end]$	< 0 $> end$ Malformed numbers Non-numeric strings Empty value
end	$[start, data.length]$	$< start$ $> data.length$ Malformed numbers Non-numeric strings Empty value

Table 7: Equivalence classes

An analysis of the classes identified in table 7 lead to the boundary cases presented in table 8.

Input	**Boundary cases**
data	[] $[1, 2, ..., 2147483647]$
data	-1, 0, 1 $> end$
end	-1, 0, 1 $< start$

Table 8: Boundary analysis

All the test cases resulting from the boundary analysis of the method *computeBestParamAndSize* can be found in table 9.

ID	Input	Expected Output	Output	Comments
24	data = [] start = 0 end = 0	256	256	N/A
25	data = [1,2,...,199491582] start = 0 end = 199491582	Long value	Long value	N/A
26	data = [1,2,3,4,5,6,7,8,9,10] start = -1 end = 1	Assertion error	Assertion error	N/A
27	data = [1,2,3,4,5,6,7,8,9,10] start = 0 end = 1	448	448	N/A
28	data = [1,2,3,4,5,6,7,8,9,10] start = 1 end = 1	256	256	N/A
29	data = [1,2,...,199491582] start = 1 end = 0	Assertion error	Assertion error	N/A
30	data = [1,2,...,199491582] start = 0 end = -1	Assertion error	Assertion error	N/A

Table 9: Test cases

8 Environmental Needs

The tools used during the development of this assignment were:

- Draw IO - Online diagrams tool with support to design the Control Flow and the Data Flow graphs.
- IntelliJ/ VS Code - IDEs with Java support
- JUnit - Java framework for test cases implementation

9 Staffing and Responsibilities

Regarding the work division and human resources needed in the development phase, the assignment can be divided into four major topics: i) Control Flow; ii) Data Flow; iii) Black box; iv) Report. There was no clear division between the work done by each team member, with both contributing to each stage.

10 Test Completion Report

Consider a **Pass** as a match between the expected and attained outputs, and a **Fail** as a discrepancy between these two attributes. According to the established criteria and by analysing the **JUnit** tests, the following table can be achieved.

ID	**Category**	Method	**Result**
1	Control Flow	writeRiceSignedInt	Pass
2	Control Flow	writeRiceSignedInt	Pass
3	Control Flow	writeRiceSignedInt	Pass
4	Control Flow	bestSizeAndParam	Pass
5	Control Flow	bestSizeAndParam	Pass
6	Control Flow	bestSizeAndParam	Pass
7	Control Flow	bestSizeAndParam	Pass
8	Control Flow	bestSizeAndParam	Pass
9	Data Flow	writeRiceSignedInt	Pass
10	Data Flow	writeRiceSignedInt	Pass
11	Data Flow	writeRiceSignedInt	Pass
12	Data Flow	writeRiceSignedInt	Pass
13	Data Flow	writeRiceSignedInt	Pass
14	Data Flow	writeRiceSignedInt	Pass
15	Data Flow	writeRiceSignedInt	Pass
16	Data Flow	writeRiceSignedInt	Fail
17	Black Box	writeRiceSignedInt	Pass
18	Black Box	writeRiceSignedInt	Fail
19	Black Box	writeRiceSignedInt	Pass
20	Black Box	writeRiceSignedInt	Pass
21	Black Box	writeRiceSignedInt	Pass
22	Black Box	writeRiceSignedInt	Pass
23	Black Box	writeRiceSignedInt	Pass
24	Black Box	bestSizeAndParam	Pass
25	Black Box	bestSizeAndParam	Pass
26	Black Box	bestSizeAndParam	Pass
27	Black Box	bestSizeAndParam	Pass
28	Black Box	bestSizeAndParam	Pass
29	Black Box	bestSizeAndParam	Pass
30	Black Box	bestSizeAndParam	Pass

Table 10: Test Completion Table

References

[1] FLAC Library Java, `https://www.nayuki.io/page/flac-library-java`, [Online; Accessed on 18/04/2023]

[2] FLAC Library Java, `https://github.com/nayuki/FLAC-library-Java`, [Online; Accessed on 18/04/2023]

[3] Black Box Testing Slides, `https://ucstudent.uc.pt`, [Online; Accessed on 09/05/2023]