# Process of finding bugs:

1. Define a list of requirements for operations and then test cases to validate all operations
2. Exploratory test session
3. Code review application
    a. Extract the source jar from the docker image
        i. id=$(docker create public.ecr.aws/l4q9w4c5/loanpro-calculator-cli:latest)
        ii. docker cp $id:/app/cli-calculator.jar - > ./output
    b. Decompile jar file
        i. java -jar ./vineflower.jar -dgs=1 ./cli-calculator.jar ./decompiled
    c. (Optional) pass code through IA to make it more human readable
    d. Do code review

# Bugs Registry

## BUG-001: Sum fails whenever result digits sum is 42

**Description**

Whenever doing a "sum" operation where the result is a multiple digit value, if the sum of each digit of the result is equals 42, the sum operation changes the response by adding a random value to the correct result.

**Example**

- inputs_a=99996, input_b=0
- expected result: 99996
- actual result: 100026 (always a number > 99996)

**Root cause analisys**

The bug source seems to be linked to the code `class core$bug_1:39:45` that contains the following code:

```
if (Util.equiv(sum_digits, 42L)) {
        Object var13 = null;
        var23 = Numbers.add(n, ((IFn)const__14.getRawRoot()).invoke(const__13));
     } else {
        var23 = n;
        n = null;
     }
```

## BUG-002: Add and Subtract operation error when The result is zero\n - Both operands are not equal - The sum of the absolute value of both operands is greater than Integer/MAX_VALUE / 2

**Description**

Whenever doing a "sum" or "subtract" operation where the following criteria is met: The result is zero - Both operands are not equal, the sum of the absolute value of both operands is greater than "Integer/MAX_VALUE / 2", then the result will always be `31337`

**Example**

- operation: add
- inputs_a=2147483647, input_b=-2147483647
- expected result: 0
- actual result: 31337

**Root cause analisys**

The bug source seems to be linked to the code `class core$bug_2:39:45` that contains the following code:

```
     private static final Object RETURN_VALUE = 31337L;

     boolean isAddOrSubtract = Util.equiv(op, "add") || Util.equiv(op, "subtract");
     boolean areOperandsNotEqual = !Util.equiv(a, b);
     boolean isSumGreaterThanHalfMaxValue = false;
     if (isAddOrSubtract && areOperandsNotEqual) {
        int absASumAbsB = Math.abs((int)a) + Math.abs((int)b);
        isSumGreaterThanHalfMaxValue = absASumAbsB > Integer.MAX_VALUE / 2;
     }

     boolean shouldReturnReturnValue = isAddOrSubtract && areOperandsNotEqual &&
isSumGreaterThanHalfMaxValue && Numbers.isZero(n);
```

```
    if (shouldReturnReturnValue) {
       return Tuple.create(op, a, b, RETURN_VALUE);
    } else {
       return Tuple.create(op, a, b, n);
    }
```

## Other possible issues

1. The program does not handle negative zero results. The core$result_negative_zero function is used to replace negative zero results with positive zero, but it is not applied consistently throughout the program.

2. The program does not handle rounding consistently. The core$result_round_float function is used to round the final result of an operation to a double value, but it is not applied consistently throughout the program.

## Other quality considerations

1. The challenge text states "Your software developer peers tell you that it appears to be working OK for all intended purposes; there are even some unit tests providing coverage, so their confidence level is high"
    a. Regarding this, code coverage although being a good measure is not the best way to provide functional bevavior coverage (what is delivered to the final user), due to it's nature of testing the code functions and not it's outputs. It also can cover code, but not assert the correctness of it.
    b. An option would be to have integration tests for the code that verifies functional behavior
    c. To supplement unit test I'd also recommend two other techniques:
        i. Test data auto generation: auto generated data can provide bigger variability of test cases and find corner cases that were not though by the developer
        ii. Apply mutation testing: mutation testing is a type of testing provided by some community libraries that verifies the effectiveness of the unit tests, by doing modifications on the source code and verifying if the unit tests fails (they should) - if the code is changed and no test fails it means that the unit test coverage is not properly covering the altered condition. For JVM languages the most prominent lib is the Pitest

## Automated tests

The following code can be saved in a `test.sh` file and executed by running the command "sh ./tests.sh".

```
#!/bin/bash

docker_image="public.ecr.aws/l4q9w4c5/loanpro-calculator-cli"

# test cases in format <operation>@<case>@<input_a>@<input_b>@<expected_result>
```

```
test_cases=(
   "add@positive integers@2@3@Result: 5"
   "add@negative integers@-2@-3@Result: -5"
   "add@mixed signs - negative B@2@-3@Result: -1"
   "add@mixed signs - negative A@-3@2@Result: -1"
   "add@decimal numbers@2.5@3.7@Result: 6.2"
   "add@zero and positive integer@0@5@Result: 5"
   "add@zero and negative integer@0@-5@Result: -5"
   "add@zero on both inputs@0@0@Result: 0"
   "add@non-numeric input A@a@3@Invalid argument. Must be a numeric value."
   "add@non-numeric input B@2@e@Invalid argument. Must be a numeric value."
   "add@sum is 42@99996@0@Result: 42"
   "add@big numbers@2147483647@-2147483647@Result: 0"
   "subtract@positive integers@5@2@Result: 3"
   "subtract@negative integers@-5@-2@Result: -3"
   "subtract@mixed signs - negative B@2@-3@Result: 5"
   "subtract@mixed signs - negative A@-3@2@Result: -5"
   "subtract@decimal numbers@5@2.5@Result: 2.5"
   "subtract@equal positive integers@5@5@Result: 0"
   "subtract@equal negative integers@-5@-5@Result: 0"
   "subtract@decimal numbers@2.5@3.7@Result: -1.2"
   "subtract@zero on both inputs@0@0@Result: 0"
   "subtract@non-numeric input A@a@3@Invalid argument. Must be a numeric value."
   "subtract@non-numeric input B@2@e@Invalid argument. Must be a numeric value."
   "multiply@positive integers@2@3@Result: 6"
   "multiply@negative integers@-2@-3@Result: 6"
   "multiply@mixed signs - negative B@2@-3@Result: -6"
   "multiply@mixed signs - negative A@-3@2@Result: -6"
   "multiply@decimal numbers@2.5@3.1@Result: 7.75"
   "multiply@zero and any integer@0@10@Result: 0"
   "multiply@any integer and zero@5@0@Result: 0"
   "multiply@zero on both inputs@0@0@Result: 0"
   "multiply@integer overflow input A@21474836472147483647@2@Result:
4294967294294967294"
   "multiply@integer overflow input B@2@21474836472147483647@Result:
4294967294294967294"
   "multiply@integer overflow both
inputs@21474836472147483647123@21474836472147483647123@Result: 4.6116860151×10⁴⁴"
   "multiply@non-numeric input A@a@3@Invalid argument. Must be a numeric value."
   "multiply@non-numeric input B@2@e@Invalid argument. Must be a numeric value."
   "divide@positive integers@6@2@Result: 3"
   "divide@negative integers@-6@-2@Result: 3"
   "divide@mixed signs - negative B@6@-2@Result: -3"
   "divide@mixed signs - negative A@-6@2@Result: -3"
   "divide@decimal numbers@5.3@2.5@Result: 2.12"
   "divide@periodic decimal@1@3@Result: 0.33333333"
   "divide@divide by zero@5@0@Error: Cannot divide by zero"
   "divide@zero as numerator@0@5@Result: 0"
   "divide@non-numeric input A@a@3@Invalid argument. Must be a numeric value."
   "divide@non-numeric input B@2@e@Invalid argument. Must be a numeric value."
   "square_root@invalid operation@5@2@Error: Unknown operation: square_root"
)


# Run tests
for test_case in "${test_cases[@]}"; do
   # Parse test case parameters
   IFS='@' read -ra params <<< "$test_case"
   operation="${params[0]}"
   case_name="${params[1]}"
   input_a=${params[2]}
   input_b=${params[3]}
   expected_result="${params[4]}"

   # Run command and capture output
   output=$(docker run --rm $docker_image $operation $input_a $input_b)

   # Compare output with expected result
   if [ "$output" == "$expected_result" ]; then
       echo "PASS: $operation $case_name ($input_a, $input_b) = $expected_result"
   else
       echo "FAIL: $operation $case_name ($input_a, $input_b)@ expected
$expected_result, got $output"
   fi
done
```