Assignment 3: Unit Testing in RUST

Tests in Rust

Rust supports writing software tests to ensure the correctness of the code. There are three styles for testing: Unit testing, Doc testing, and Integration testing. Moreover, Rust supports specifying Dev-dependencies. Tests are functions that are written to verify that the non-test code is running properly. The structure of a test function typically involves some setup, run the "non-test code" you wish to test, then assert whether the results are what we expect.

Unit tests should be combined into a test *mod* with the #[cfg(test)] attribute. To define test functions, functions should be marked with the #[test] attribute before the function definition. For example, look at the following code:

```
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
// This is a really bad adding function
#[allow(dead_code)]
fn bad_add(a: i32, b: i32) -> i32 {
    a - b
}

#[cfg(test)]
mod tests {
    // Note this useful idiom: importing names from outer scope.
    use super::*;
    #[test]
    fn test_add() {
        assert_eq!(add(1, 2), 3);
    }
    #[test]
    fn test_bad_add() {
        // This assert would fire and test will fail.
        // Please note, that private functions can be tested too!
        assert_eq!(bad_add(1, 2), 3);
    }
}
```

Tests can be run with cargo test.

```
$ cargo test

running 2 tests
test tests::test_bad_add ... FAILED
test tests::test_add ... ok

failures:
---- tests::test_bad_add stdout ----
        thread 'tests::test_bad_add' panicked at 'assertion failed:
`(left == right)`
```

```
  left: `-1`,
 right: `3`', src/lib.rs:21:8
note: Run with `RUST_BACKTRACE=1` for a backtrace.

failures:
    tests::test_bad_add

test result: FAILED. 1 passed; 1 failed; 0 ignored; 0 measured; 0
filtered out
```

Question 1:

Source code: UTApp.zip

The application provides 4 test modules, question1, question2, question3, and question4.

- **a-** Implement the functionality defined by *question1* to pass all the tests in this module. Your task is to define the functions *add* and *subtract* in the calculator.rs file.

- **b-** Running the tests in question2.rs will pass with no problems because there no actual tests in the file. Your task is to write the functionality defined by the tests (*multiply* and *divide* functions) and write the tests cases for your code to test these two functions.

- **c-** Implement the functionality defined by question3 to pass all the tests in this module. Your task is to define the functions *get_square_root* in the calculator.rs file. Then write the code for the four test functions defined in question3.

- **d-** Implement the functionality defined by question4 to pass all the tests in this module. Your task is to define the functions *get_roots* to calculate the roots for quadratic equations. Then, you need to write the code for four tests functions defined in question4.

Question 2:

Hamcrest is a framework for defining 'match' rules and designing matcher objects. When writing tests, it could be challenging to have the right balance between over-specifying the test (and making it brittle to changes), and not specifying enough (making the test less valuable since it continues to pass even when the thing being tested is broken).

Having a tool that allows you to pick out precisely the aspect under test and describe the values it should have, to a controlled level of precision, helps significantly in writing tests that are "just right". Such tests fail when the behavior of the aspect under test deviates from the expected behavior, yet continue to pass when minor, unrelated changes to the behavior are made.

Let us assume we have a struct for a player instance as follows:

```
struct Player {
    id: i32,
    first_name: String,
    last_name: String,
```

```
}
```

Use the Hamcrest framework to write tests that assert that:

a- Player has a property called last_name with the type String.
b- Two Players created with the same id, have the same first name and last name.

Question 3: Redo question 1 using the Hamcrest framework. Submit your code as UTApp_hamcrest.

Question 4: Combination function algorithm

a- Write a program that finds the value of the combination function, c(a, b) where a and b are both read as user input (*Hint:* you may need to use io::stdin().read_line()).

Combination function is defined as:

$$\boldsymbol{c(a,b)} = \frac{\boldsymbol{a}!}{(\boldsymbol{a}-\boldsymbol{b})!\,\boldsymbol{b}!}$$

b- Write a test that ensures **a** is not smaller than **b**.
c- Write a test that ensures both a and b are integers (Hint: use should_panic)

Question 5: Write a *tax()* function that calculates the taxed income based on the following table:

Income	Tax Rate (%)
0 ≤ income < 10,000	0
10,000 ≤ income < 50,000	10
50,000 ≤ income < 100,000	20
100,000 ≤ income < 1,000,000	30
Income ≥ 1,000,000	40

The function should return the taxed income as:

*Taxed_income = income - (income * tax_rate)*

Write tests that ensure that your code can handle the following error conditions (Hint: use should_panic):

- Income is a negative number.
- Income is not an integer.