Testing and Debugging Assignment (Evaluation)

CSCI 221: Computer Science Fundamentals 2

Fall 2024

This assignment is designed to test your understanding of testing and debugging. Feel free to collaborate with others and use resources, but all code and writeups must be your own. For this assignment, you should submit your code along with a writeup that contains the answers to the written questions.

Due Date: Friday, September 27th at 1:00 pm.

1. (33 points) **Testing.** Using the method specified, write a series of test cases for the specified problem. For each test case, you should describe:

   1. The input to the test case.
   2. The expected behavior (output) of the test case.
   3. An explanation of how you chose the test case using the specified methodology. This does not need to be long, a sentence or two should be sufficient.
   4. What errors you expect the test case to catch.

   (a) (6 points) For this part and the next, you will be considering the following problem:

   This function will count how many times each day of the week appears in a range of calendar years. The input will consist of one character representing the day of the week, and two unsigned integers representing the starting and ending year, respectively. (Both years should be included in the count.) The characters used to represent the days of the week are `[S,M,T,W,h,F,a]` for Sunday through Saturday, respectively. The function only handles the years between 2000 and 2050.

   Create tests for this problem using equivalence testing.

   SOLUTION: I will assume that the range 2001, 2001 indicates January 1st to December 31st of the year 2001.

   a. The input cases can be (1)(S, 2001, 2007), (2)(M, 2004, 2001), (3)(T, 2051, 2060), (4) (a, 2014, 2020), (5) (T, 2032, 2038), (6) (W, 2041, 2047), (7) (h, 2022, 2028), (8) (F, 2013, 2019), (9) (b, 2014, 2020).
   b. (1) valid, (2) invalid input (wrong order of start and end years), (3)invalid input (years out of range), (4) valid, (5) valid, (6) valid, (7) valid, (8) valid, (9) invalid (day out of range).
   c. The possible input space is composed of `[all characters]` $\times \mathbb{Z}^+ \times \mathbb{Z}^+$ I divided up the input space into one valid class and three invalid classes. The valid class was then subdivided into 6 sub-classes where we expect a similar output. Meaning, for each valid class, the span of the year-range outputs number which are not that far apart.
   d. The inputs (2), (3), (9) are meant to catch errors involving unaccepted inputs. (2) catches errors involving order of years, the smaller number must come first. (3) catches error involving out of the range year input. Lastly, (9) catches error involving out of range day input.

   (b) (6 points) Create tests for the problem from the previous part using boundary testing.

   SOLUTION:

   a. The input cases are (1) (S, 2000, 2000), (2) (T, 1999, 2000), (3) (M, 2050, 2050), (4) (f, 2040, 2042), (5) (h, 2049,2051).

b. (1) valid, (2) invalid, (3) valid, (4) invalid, (5) invalid.

c. I have identified two boundaries: the year boundary (2000-2050) and case boundary. Based on that, we set up test cases. The cases (1), (3) are valid and involve cases in which we expect to get outputs, but the cases (2), (4), and (5) are invalid and each of them miss some boundary test.

d. The cases (2), (4), and (5) different errors involving the established boundaries. (2) and (4) raises an error because the years 1999 and 2051 are not within our year range. Similarly, the case error is raised by (5), since we accept `F` to indicate Friday, not `f`.

(c) (3 points) For this part and the next two parts, you will be considering the following code:

```
unsigned int turn = 1;
unsigned int last_turn = 0;
unsigned int first_score = 0;
unsigned int second_score = 0;

void reset(void) {
    turn = 1;
    last_turn = 0;
    first_score = 0;
    second_score = 0;
}

unsigned int turn(int player, unsigned int dice) {
    if (player == turn) {
        if (player == 1) {
            for (int i=0; i < dice; i++) {
                first_score += die_roll();
            }
            if (first_score > 100) {
                turn = 0;
                return 2;
            }
        }
        if (player == 2) {
            for (int i=0; i < dice; i++) {
                second_score += die_roll();
            }
            if (second_score > 100) {
                turn = 0;
                return 1;
            }
        }
        if (last_turn == 1) {
            turn = 0;
            if (first_score > second_score) {
                return 1;
            } else if (second_score > first_score) {
                return 2;
            } else {
                return 3;
            }
        }
        if (dice == 0) {
            last_turn = 1;
```

```
            }
            turn = 3 - turn;
        }
        return 0;
    }
```

This code plays the game Greed that students in CSCI 121 have to implement strategies for. If you want to see the rules to the game, look at the pdf file in the folder with the assignment.

The code takes as input an integer representing which player is trying to take a turn, and an unsigned integer representing how many dice they want to roll. It tracks the state of the game, and returns an integer indicating the state at the end of the turn. If the game is not over, the function returns 0. If the game is over, then the function returns the winning player, represented by 1 or 2. In the event of a tie, the function returns 3.

What is the difficulty of testing this code as it is given to you? How can you overcome this problem if you wanted to test it?

SOLUTION: There are a few design problems to fix before we can test the code:

1. We have a function and a variable both named `turn`. This is against standard naming practices.
2. The code doesn't offer any comments, so it's hard to follow the flow of logic.
3. Instead of using `switch` the code heavily relies on `if`, `else` and `else if` conditionals.

•

(d) (9 points) Create tests for the code from the previous part using control flow testing. You should submit an image of your control flow graph along with your test cases.

SOLUTION:

a. The following are the test cases:
(1) (1,3) → (2,2) → (1, 100),
(2) (1,1) → (2,100),
(3) (1,3) → (2,2) → (1, 0) → (2,100),
(4) (1,1) → (2,0) → (1,100),
(5) (0,0),
(6) (2,1),
(7) (1,0) → (2,0)
(8) (1,1) → (1,2)

b. (1) player 1 begins playing, then player 2 is prompted to play, player one is prompted to play again, and in the end Player 2 wins. (2) player one begins playing, then player 2 is prompted to play, in the end player 1 wins. (3) player one begins playing, then player 2 is prompted to play, player one is prompted to play again but passes, player 2 plays, and in the end, player 1 wins. (4) player one begins playing, then player 2 is prompted to play but passes, player 1 is prompted to play again, in the end player 2 wins. (5) Game terminates, (6) Game terminates, (7) player begins the play but passes, player two also passes, the game is tied. (8) player one begins the play, and then player one tries playing again, but game resets.

c. The methodology relies on checking every possible route of logical flow of the game. In 8 test cases, we are able to cover most of the possible turns of the game. This is suitable because there are finitely many possible scenarios.

d. Test cases (5) and (6) catches initial input errors and terminates the game. Test case (8) catches the wrong flow of the game and terminates.

(e) (9 points) Create tests for the code from the previous part using state-based testing. You should submit an image of your finite state machine along with your test cases.

SOLUTION: The states are given by the set: { player 1 plays, player 1 wins, player 2 plays, player 2 wins, tie, pass}

The events are given by the set: { player, dice }

The valid transitions are given by the following dictionary for convenience: { player 1 plays: [player 1 wins, player 2 wins, pass, player 2 plays], player 2 plays: [player 1 wins, player 2 wins, pass, player 1 plays], pass: [tie] }

Test cases remain exactly the same as the control flow testing.

a. (1) (1,3) → (2,2) → (1, 100),
(2) (1,1) → (2,100),
(3) (1,3) → (2,2) → (1, 0) → (2,100),
(4) (1,1) → (2,0) → (1,100),
(5) (0,0),
(6) (2,1),
(7) (1,0) → (2,0)
(8) (1,1) → (1,2)

b. For convenience, I'll denote the expected behavior using the following indexing for the states: { player 1 plays = p1p, player 1 wins = p1w, player 2 plays = p2p, player 2 wins = p2w, tie = t, pass = p} (1) p1p → p2p → p1p → p2w
(2) p1p → p2p → p1w
(3) p1p → p2p → p1p → p → p2p → p1w
(4)p1p → p2p → p → p1p → p2w
(5) game terminates
(6) game terminates
(7) p1p → p → p2p → p → t
(8) game terminates.

c. The methodology is based on the fact that the game can also be treated as a finite state machine with finite set of states, events, and valid transitions. This makes it easier to get an overview of the underlying mechanisms of the game.

d. Test (5), (6) catches wrong initialization errors, and (8) catches state flow errors.

•

2. (12 points) **Debugging.** Consider the following code, which attempts to compute exponentials for non-negative integers.

```
unsigned int f(unsigned int x, unsigned int y) {
    int r = 1;
    int s = x;
    while (y > 1) {
        if (y % 2 == 1) {
            r = x * r;
        }
        s = s * s;
        y = y / 2;
    }
    return r * s;
}
```

Your job is to debug the code so that it performs as specified. You should submit the following:

1. An explanation of any defects in the original code.

2. For each defect, a brief description of how you found that defect. Your description should contain any experiments you did to verify your analysis. A few sentences should be sufficient to do this.
3. The code with the defects fixed and working.

As part of the debugging process, modify the code so that it follows good design practices.

SOLUTION:

a. (1) The first defect that I found was with the inputs of the form $(x, 0)$ where $x \in \mathbb{Z}^+ \cup \{0\}$. Instead of returning 1, it returns x, (2) the second defect is that `r = x * r` does not use updated x at odd iterations. By odd iterations, I mean iterations at which y is odd. As a result, for inputs like $(2, 7)$ and $(2, 11)$, the algorithm's output misses a factor of 2 in each case.

b. Defect (1) was found using boundary testing. That is by testing the cases $(0, 0)$, $(1, 0)$. The hypothesis was that the outputs will be 1 in both cases, which failed because the test cases never make it to the `while y>1` loop and as a result returns `r * s = 1 * x = x` where $x = 0, 1$ respectively.
Defect (2) was found using hypothesis testing. The hypothesis was that testing

$$(2, 0), (2, 1), (2, 3), \cdots, (2, 11), (2, 12)$$

should return $\{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096\}$
However, we got $\{2, 2, 4, 8, 16, 32, 32, 128, 256, 512, 1024, 1024, 4096\}$ As a result, it was clear that there is a defect in the code.

c.

```
unsigned int f(unsigned int x, unsigned int y) {
    if (y == 0){
        printf("1\n");
        return 1;
    }
    int exponent = y;
    int value = 1; // this will eventually become result, we update this
    int base = x;
    while (exponent > 1) {
        if (exponent % 2 == 1) {
            value = base * value;
        }
        base = base * base;

        exponent = exponent / 2;

    }
    int output = value * base;

    return output;
    }
```