CSCI Testing and Debugging

Riley Wood

September 2024

1 Problem 1

(a) 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

(1) **Input:** S, 2000, 2005.
Behavior: Outputs the correct answer
Reasoning: Ensure program functions properly for expected usage.
Errors Caught: Ensures no mathematical mistakes. Ensures start and end year are included.

(2) **Input:** S, 2000, 2000.
Behavior: Outputs the number of Sundays in 2000
Reasoning: Ensure program functions for two entries of the same year
Errors Caught: Ensures start and end year are not double counted when the same.

(3) **Input:** S, 2005, 2000.
Behavior: Outputs the number of Sundays between 2000-2005
Reasoning: Ensure program functions for valid inputs entered out of order (this could be made to expect an error if that is the preferred behavior in this case).
Errors Caught: Ensures that out of order inputs are not resulting in negative days, etc.

(4) **Input:** k, 2000, 2005.
Behavior: Outputs an invalid day error
Reasoning: Ensure that invalid day of the week results in expected error.
Errors Caught: Processing the number of k's in the years (somehow).

(5) **Input:** S, -100, 2005
Behavior: Outputs an invalid year error
Reasoning: Ensure that entry of a year below the lower bound results in expected error
Errors caught: Ensure there is no counting of negative years/years out of range.

(6) **Input:** S, 2000, 15000000.
Behavior: Outputs an invalid year error
Reasoning: Ensure that entry of a year outside the upper bound results in expected error.
Errors Caught: Ensures the program cannot continue counting outside of bounds.

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

(1) **Input:** S, 2005, 2010.
Behavior: Outputs the right answer
Reasoning: Ensure program functions properly for expected usage.
Errors Caught: Ensures no mathematical mistakes. Ensures start and end year are included.

(2) **Input:** S, 2000, 2005.
Behavior: Outputs the correct answer
Reasoning: Ensure program functions for lower year boundary
Errors Caught: Avoids off by one error

(3) **Input:** S, 1999, 2005.
Behavior: Outputs year out of bounds error
Reasoning: Check the lower bound of the years parameter.
Errors Caught: Ensures that the program will not run on a year outside of its bounds.

(4) **Input:** S, 2005, 2050.
Behavior: Outputs the correct answer
Reasoning: Check if the function works for the upper year bound
Errors Caught: Avoids off by one error

(5) **Input:** S, 2005, 2051
Behavior: Outputs an invalid year error
Reasoning: Ensure that entry of a year above the upper bound results in expected error
Errors caught: Ensure there is no counting of years out of range.

(6) **Input:** k, 2005, 2010.
Behavior: Outputs an invalid day error
Reasoning: Ensure the program will not run on invalid days
Errors Caught: Catches errors caused by trying to count an unknown day.

(c) 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?

The main challenge of testing this code is its lack of any helper functions or header files that allow interfacing with the function. I would likely write a function that allows testing of multiple inputs with a single command, and allow it to be used by another testing C file with a header

(d) 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.

See photos of my flow and state diagrams in the submission folder. My test cases would include:

(1) **Input:** Player != turn.
Behavior: Returns immediately
Reasoning: Ensure that program behaves to specification for invalid player.
Errors Caught: Avoids any input handling issues.

(2) **Input:** Player = 1 and dice = 0.
Behavior: Begins dice loop from player 1 and terminates immediately.
Reasoning: Ensure dice loop works for player 1 with no dice.
Errors Caught: Avoids loop errors, off-by-one errors, logical errors with player 1.

(3) **Input:** Player = 1 and dice = 6.
Behavior: Begins dice loop from player 1 and runs loop for some number of iterations without score ever exceeding 100.
Reasoning: ensure dice loop works for player 1 with some number of dice but a score that will cause the program to return to normal flow.
Errors Caught: Avoids loop errors, off-by-one errors, logical errors with player 1.

(4) **Input:** Player = 1 and dice = 600.
Behavior: Begins dice loop from player 1 and runs loop for some number of iterations guaranteeing the score exceeds 100.
Reasoning: ensure dice loop works for player 1 and the program behaves properly for score exceeding 100.
Errors Caught: Avoids loop errors, logical errors with player 1.

(5) **Input:** Test cases 2, 3, 4 repeated with Player = 2.

(6) **Input:** Player is neither 1 nor 2 and last turn = 1.
Behavior: Skips execution of player 1 and 2 branches and tests last turn logic

Reasoning: Ensure that the last turn logic functions properly.
Errors Caught: Any logical error with last turn handling.

(7) **Input:** Player is neither 1 nor 2, last turn = 0, and dice = 0.
Behavior: Skips execution of player 1 and 2, passes last turn check and executes dice = 0 logic.
Reasoning: Ensure that the dice = 0 logic functions properly.
Errors Caught: Any logical error with dice count handling.

(8) **Input:** Player is neither 1 nor 2, last turn = 0, and dice ¿ 0.
Behavior: Skips execution of player 1 and 2, passes last turn check and executes dice != 0 logic.
Reasoning: Ensure that the dice != 0 logic functions properly.
Errors Caught: any logical error with dice count handling.

(e) 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:

(1) **Input:** Player = 1
Behavior: Checks state entry to rolling P1
Reasoning: Ensure all logic for state entry to P1 works.
Errors Caught: any logical error with P1 entry.

(2) **Input:** Player = 2
Behavior, Reasoning, Errors Caught: see Player = 1

(3) For each of those:

i. **Input:** Last turn false, score > 100
Behavior: Checks state transition from rolling to return
Reasoning: Ensure state transition to return is captured.
Errors Caught: Errors with if statements, loop errors.

ii. **Input:** Last turn false, score < 100
Behavior: Checks state transition from rolling to exit
Reasoning: Ensure state transition to exit is captured by tests.
Errors Caught: Errors with if statements, loop errors.

iii. **Input:** Last turn true, score < 100
Behavior: Checks state transition from rolling to last turn
Reasoning: Ensure state transition to last turn is captured by tests.
Errors Caught: Any error with checking if last turn is handled correctly.

(4) **Input:** last turn = 1 (player != 1, 2)
Reasoning: Ensure state transition to last turn is captured by tests.
Errors Caught: Any error with checking if last turn is handled correctly.

(5) **Input:** Player != 1, 2. last turn =0.
Behavior: Checks state transition from entry to exit

Reasoning: Ensure state transition to exit is captured by tests.
Errors Caught: Any logical error in the string of if-statements

2 Problem 2

One defect was found in the original code: in the case $y = 0$, the program would output $x^0 = x$ rather than $x^0 = 1$. This defect was found via boundary testing (0 is the smallest non-negative integer). I did a static analysis and tested the code using gdb after the boundary test failed and found that not handling the case $y = 0$ was the defect.