HW3 | Ria Singh and Anika Bhatnagar | Senior project 2

Problem 7.1, Stephens page 169

The greatest common divisor (GCD) of two integers is the largest integer that evenly divides them both. For example, the GCD of 84 and 36 is 12, because 12 is the largest integer that evenly divides both 84 and 36. You can learn more about the GCD and the Euclidean algorithm, which you can find aten.wikipedia.org/wiki/Euclidean_algorithm. [Hint: It should take you only a few seconds to fix these comments. Don't make a career out of it.]

```
        // Use Euclid's algorithm to calculate the GCD.
         provate long GCD( long a, long b )
         {
            // Repeat until we're done
            for( ; ; )
            {
               // Set remainder to the remainder of a / b
               long remainder = a % b;
               // If remainder is 0, we're done.  Return b.
               If( remainder == 0 ) return b;
               // Set a = b and b = remainder.
               a = b;
               b = remainder;
            };
         }
```

Problem 7.2, Stephens page 170

According to your textbook, under what two conditions might you end up with the bad comments shown in the previous code?

1. Writing comments after code implementation: When comments are added after the code is written, they may not accurately reflect the code's functionality or intent, leading to misleading or unhelpful documentation.
2. Writing comments without proper planning: Without thoughtful planning, comments can become inconsistent, redundant, or lack clarity, reducing their effectiveness in explaining the code.

Problem 7.4, Stephens page 170

How could you apply offensive programming to the modified code you wrote for exercise 3? [Yes, I know that problem wasn't assigned, but if you take a look at it you can still do this exercise.]

Offensive programming involves writing code that assumes things will go wrong and aggressively checks for errors using assertions and explicit validation.

To apply offensive programming to the modified code from Exercise 3, we could:

- Use Assertions: Ensure that input values are within expected bounds before processing.
- Validate User Input: Before using input values, check that they meet the required format, such as valid numbers, strings, or data structures.
- Fail Fast: If a critical error occurs, exit early instead of allowing the program to continue in an unstable state.
- Check Preconditions and Postconditions: Verify function inputs and outputs to maintain consistency.
- Debug Logs: Add debug statements to track execution flow, making it easier to pinpoint errors.

By implementing these strategies, we make the code more robust and prevent hidden errors from causing unexpected behavior.

Problem 7.5, Stephens page 170

Should you add error handling to the modified code you wrote for Exercise 4? Explain your reasoning.

Yes, error handling should be added to the modified code from Exercise 4, but only where necessary. The goal is to handle predictable issues without hiding serious bugs.

Reasons to Add Error Handling:

User Input Handling: If the program accepts user input, we must check for invalid values to prevent crashes.

File Operations: Reading or writing files may fail due to permissions or missing files. Use try-except to catch these errors.

Network Requests: If the code communicates over a network, handle timeouts and connection failures gracefully.

When Not to Add Error Handling:

- For Developer Bugs: Catching exceptions should not mask programming errors like IndexError in a loop—those should be fixed, not handled.
- Silent Failures: Avoid generic try-except blocks that suppress errors without logging them, as this makes debugging harder.

Best Approach: Use error handling for expected issues (user input, I/O operations) but rely on offensive programming to catch developer errors early.

Problem 7.7, Stephens page 170

Using top-down design, write the highest level of instructions that you would use to tell someone how to drive your car to the nearest supermarket. (Keep it at a very high level.) List any assumptions you make.

High-Level Instructions:

1. Prepare for the trip.

   - Ensure you have your car keys, wallet, and any necessary items.

   - Check that the car has enough fuel.

   - Fasten your seatbelt.

2. Start the car and navigate to the supermarket.

   - Turn on the engine.

   - Use GPS or prior knowledge to determine the route.

   - Follow traffic rules and drive safely toward the supermarket.

3. Park the car at the supermarket.

   - Find an available parking spot.

   - Park within the designated lines.

   - Turn off the engine and exit the vehicle.

Assumptions:

- The driver has a valid driver's license and knows how to operate the car.

- The supermarket is within a reasonable driving distance.

- Roads are clear and normal driving conditions apply (e.g., no extreme weather).

- The driver has access to navigation tools (GPS or prior knowledge of the route).

- The supermarket has parking available.

Problem 8.1, Stephens page 199

Two integers are *relatively prime* [or *coprime*] if they have no common factors other than 1. For example, 21 = 3 X 7 and 35 = 5 X 7 are *not* relatively prime because they are both divisible by 7. However, integers 8 = 2 X 4 and 9 = 3 X 3 ARE relatively prime because the only common factor they have is 1 [even though NEITHER of them is prime by itself!] By definition -1 and 1 are relatively prime to every integer, and they are the only numbers relatively prime to 0.

Suppose you've written an efficient **`isRelativelyPrime()`** method that takes two integers between -1 million and 1 million as parameters and returns **`true`** if they are relatively prime. Use either your favorite programming language or pseudocode to write a program that ***tests*** the **`isRelativelyPrime()`** method.
[Hint: You may find it useful to write the **`isRelativelyPrime()`** method itself as well.]

```
import math

def isRelativelyPrime(a, b):
    """Returns True if a and b are relatively prime (GCD is 1)."""
    return math.gcd(a, b) == 1


def test_isRelativelyPrime():
    """Tests the isRelativelyPrime function with sample cases."""
    test_cases = [
        (8, 9, True),   # 8 and 9 have only 1 as common factor
        (21, 35, False), #both divisible by 7
        (13, 27, True),  #prime and composite with no common factors
        (-5, 14, True),  #negative number case
        (0, 7, True),    # by definition, ±1 and 0 are relatively prime to all numbers
```

```
        (100, 200, False), #common factor 100
        (101, 103, True)  #both prime, different numbers
    ]

    for a, b, expected in test_cases:
        result = isRelativelyPrime(a, b)
        print(f"isRelativelyPrime({a}, {b}) → {result} (Expected: {expected})")
        assert result == expected, f"Test failed for {a}, {b}"

    print("All test cases passed!")

# run test function
test_isRelativelyPrime()
```

Problem 8.3, Stephens page 199

1. What testing techniques did you use for the program in Exercise 8.1? [Exhaustive, black-box, white-box, or gray-box?]
2. Which ones *could* you use and under what circumstances? [Justify your answer with a short paragraph to explain.]

Testing Techniques Used:

1. Black-Box Testing

- The test cases were designed without looking at the internal implementation of `isRelativelyPrime()`.

- focused on input-output behavior to ensure correctness across different types of inputs (e.g., prime numbers, negative numbers, 0, and composite numbers).

2. Boundary Testing

- included edge cases like `0`, negative numbers, and large values to verify the function's behavior at its limits.

Other Techniques That Could Be Used:

- White-Box Testing

- If we had analyzed the internal logic (such as ensuring the correct use of `math.gcd()`), this would count as white-box testing.

- This would be useful for optimizing performance or ensuring the function follows an expected execution path.

- Gray-Box Testing

- If we had partial knowledge of the implementation (e.g., understanding that it relies on the `math.gcd()` function but not the internal details of how `math.gcd()` works), it would be considered gray-box testing.

- This would help in designing test cases that target specific conditions based on internal logic.

- Exhaustive Testing

- Checking every possible pair of integers from `-1,000,000` to `1,000,000` would be exhaustive testing.

- However, this is impractical due to the vast number of possibilities, so selective representative test cases are a more efficient alternative.

Problem 8.5, Stephens page 199 - 200

the following code shows a C# version of the **`areRelativelyPrime()`** method and the **`GCD`** method it calls.

```
        // Return true if a and b are relatively prime.
         private bool areRelativelyPrime( int a, int b )
         {
            // Only 1 and -1 are relatively prime to 0.
            if( a == 0 ) return ((b == 1) || (b == -1));
            if( b == 0 ) return ((a == 1) || (a == -1));

            int gcd = GCD( a, b );
            return ((gcd == 1) || (gcd == -1));
         }
```

```
        // Use Euclid's algorithm to calculate the
        // greatest common divisor (GCD) of two numbers.
        // See https://en.wikipedia.org/wiki/Euclidean_algorighm
        private int GCD( int a, int b )
        {
           a = Math.abs( a );
           b = Math.abs( b );

           // if a or b is 0, return the other value.
           if( a == 0 ) return b;
           if( b == 0 ) return a;

           for( ; ; )
           {
              int remainder = a % b;
              if( remainder == 0 ) return b;
              a = b;
              b = remainder;
           };
        }
```

The **areRelativelyPrime()** method checks whether either value is 0. Only -1 and 1 are relatively prime to 0, so if a or b is 0, the method returns **true** only if the other value is -1 or 1.

The code then calls the **GCD** method to get the greatest common divisor of **a** and **b**. If the greatest common divisor is -1 or 1, the values are relatively prime, so the method returns **true**. Otherwise, the method returns **false**.

Now that you know how the method works, implement it and your testing code in your favorite programming language. Did you find any bugs in your initial version of the method or in the testing code? Did you get any benefit from the testing code?

Bugs found:

- Bug in Original C# Code:
  - The C# implementation incorrectly checks Math.abs() as Math.abs(a) instead of Math.abs(a), which would cause a syntax error.
  - The for(; ;) loop is unnecessary; a while loop is a cleaner approach.
- Edge Cases That Might Cause Issues:
  - The C# implementation does not explicitly handle negative values before the loop, but it works because %handles negatives correctly.

- If the function were implemented incorrectly, it might fail when testing (0, 2), since 0 is not relatively prime to numbers other than ±1.

benefit:

1. Caught edge cases: It verified behavior for 0, negative numbers, and prime/composite pairs.
2. Debugging efficiency: If a test failed, it pointed to exactly which case was incorrect, allowing quick fixes.

Problem 8.9, Stephens page 200

Exhaustive testing actually falls into one ot the categories black-box, white-box, or gray-box. Which one is it and why?

Exhaustive testing falls under white-box testing because it requires complete knowledge of the system's internal workings and ensures that every possible input combination is tested.

Problem 8.11, Stephens page 200

Suppose you have three testers: Alice, Bob, and Carmen. You assign numbers to the bugs so the testers find the sets of bugs {1, 2, 3, 4, 5}, {2, 5, 6, 7}, and {1, 2, 8, 9, 10}. How can you use the Lincoln index to estimate the total number of bugs? How many bugs are still at large?

The Lincoln Index is used in software testing to estimate the total number of bugs in a system based on overlapping discoveries between multiple testers.

1. Define Sets

The testers found the following sets of bugs:

- Alice: {1, 2, 3, 4, 5}
- Bob: {2, 5, 6, 7}
- Carmen: {1, 2, 8, 9, 10}

The total unique bugs found across all testers:

{1,2,3,4,5,6,7,8,9,10}{1,2,3,4,5,6,7,8,9,10}

Total unique bugs found = 10.

2. Apply the Lincoln Index Formula

N=(A×B)CN=C(A×B)

Where:

- A = Number of bugs found by Alice = 5
- B = Number of bugs found by Bob = 4 (since Bob found {2, 5, 6, 7})
- C = Number of bugs found by both Alice and Bob = 2 (common bugs: {2, 5})

N=(5×4)2=202=10N=2(5×4)=220=10

3. Compare with Observed Unique Bugs

- The Lincoln Index estimates 10 total bugs.
- Since the testers have already found 10 unique bugs, this suggests that all bugs have been discovered.
- Bugs still at large: 0.

Problem 8.12, Stephens page 200

What happens to the Lincoln estimate if the two testers don't find any bugs in common? What does it mean? Can you get a lower bound estimate of the number of bugs?

If the two testers do not find any bugs in common, the Lincoln Index formula:

N=(A×B)CN=C(A×B)

becomes undefined because C (the number of bugs found in common) is 0, leading to a division by zero error.