

Preparation: Before understanding the codebase, read its requirements that describe its intended functionality. Be aware that this implementation may differ from your previous experiences or knowledge. Throughout the debugging process, if needed, you can refer back to the codebase functionality description to avoid incorrect assumptions about any section's purpose or behavior.

Procedure:

1. Get a quick overview of the codebase to develop a high-level understanding of the code structure
 1. Start from the codebase's entry point, which is the `Main` function in file `Program.cs` on line 9.
 2. Trace the general control flow through the codebase, observe how different mathematical operations are demonstrated (lines 14-20), and how various classes are used. Take stock of the codebase structure. Pay attention to:
 1. Functions/components: `PrimeTester`, `PrimeFactorization`, `EuclideanAlgorithm`, `ModularArithmetic`, `FibonacciNumbers`, `PerfectNumbers`, and `MathUtils` classes
 2. Their locations within the code structure: `NumberTheory.cs`, `AdvancedNumberTheory.cs`, and `MathUtils.cs` files
 3. How they interact with each other (i.e., method calls): How different methods are called in the demonstration functions in `Program.cs`
2. Identify and examine potential bug-containing code sections
 1. First, decide which code sections require more thorough examination: Based on your overview gained from the previous step, prioritize sections with a higher chance of containing the bug (such as those with core logic functions, complex calculations, loops, and conditional structures). For instance, the `GetPrimeFactors` method in the `PrimeFactorization` class (`NumberTheory.cs`, line 44) is a prime candidate for examination due to its core role in the prime factorization logic.
 2. Start with the `GetPrimeFactors` method in `NumberTheory.cs` that you believe is the most potentially bug-relevant.
 3. Trace the data flow through the method, focusing on the main `for loop` (lines 47-55) and the factor addition logic. If needed, refer back to the overall functionality description to ensure accurate understanding.
 4. Identify what this section's input(s) should be and propose inputs likely to trigger the bug. For example, consider numbers like 84 (composite) and 97 (prime).
 5. Perform mental calculations with your proposed inputs: Go through this section and calculate its intermediate output/behavior. Take notes on how the factors list is populated.

6. Compare the calculated output (or observed behavior) with the expected output:
 1. If match: conclude this section is likely bug-free, move to the next section (e.g., `ModularExponentiation` method in `ModularArithmetic` class), and repeat from Step 2.2.
 2. If they don't match: conclude this section likely contains the bug. Form a hypothesis about which statement(s) are problematic. Based on your previous calculations, compare each statement's intermediate output/behavior with the expected output to identify the mismatch. Once identified, propose a fix and move to Step 3 to validate your hypothesis.
7. If the bug remains undetected, revisit potentially bug-relevant sections identified earlier, such as the `IsPrime` method in `PrimeTester` class or the `GCD` method in `EuclideanAlgorithm` class, rechecking them (Step 2.1) to ensure proper understanding.
8. If still unresolved, expand your analysis to sections initially considered less likely to contain the bug, such as the `MathUtils` class, applying the same process (Steps 2.2 to 2.6) to each.
3. Validate your proposed bug fix
 1. Focus on the `GetPrimeFactors` method you believe contains the bug. Assume you've implemented the fix and other sections work correctly.
 2. Redo the mental calculation from Step 2.5 with the assumed fix in place. Take notes on recalculated intermediate outputs:
 1. If you are confident about your identified bug, you may choose to recalculate only the fixed statement.
 2. Otherwise, if you are less certain, you have the option to recalculate the entire section for a more thorough check.
 3. Compare the new output with the expected output:
 1. If they match: Your proposed fix likely solves the bug
 2. If they don't match: Your fix may be incorrect, or this section may not contain the bug. Consider:
 1. If you have another hypothesis for this section, return to Step 3.1 to validate it.
 2. Otherwise, return to Step 2 to analyze other code sections.
 4. Repeat Steps 2 and 3 until the bug is resolved or all possibilities are exhausted.