Strategies

1 Debugging Strategy

```
# If you've spent a lot of time debugging unfamiliar code, the way that you probably debug is to first look at the
# failure, then look at the code to understand how it's  architected, and then look for possible reasons for why the
# program failed. Once you have a guess, you probably then check # it with things like breakpoints and logging. This
# strategy often works if you can have a lot of prior experience with debugging and inspecting program state. But if you
# don't have that experience, or you happen to guess wrong, this approach can lead to a lot of dead ends.
#
# The strategy you're about to use is different. Instead of guessing and checking, this strategy involves systematically
# working backwards from the code that directly caused the failed output to all of the code that caused that failed
# output to occur. As you work backwards, you'll check each statement for defects. If you work backwards like this,
# following the chain of causality from failure to cause, you will almost certainly find the bug.

STRATEGY debug()
    # This first step will give you enough familiarity to find
    # lines in the program that create the program's output.
    # Read the names of all of the functions and variables in the program
    # Some programs produce command line output with print statements.
    # Is the faulty output you're investigating printed to a command line?
    IF the faulty output is logged to a command line
        # To find print statements, try searching for keywords related to 'log' or 'print'
        SET outputLines TO the line numbers of calls to console logging functions
    # Graphical output includes things like colored lines and rectangles
    IF the faulty output is graphical output
        # To find these lines, try searching for keywords related to graphical output, like 'draw' or 'fill'.
        #Focus on lines that directly render something, not on higher-level functions that indirectly call rendering
        # functions.
        SET outputLines TO the line numbers of function calls that directly render graphics to the screen
    # Now that you have some lines that could have directly produced the faulty output, you're going to check each
    # line, see if it executed, and then find the cause of it executing. If you're lucky, you only have one
    # output line to check.
    FOR EACH 'line' IN 'outputLines'
        # First, let's make sure the line executed. You want to be sure that this is actually the source of the wrong
        # output. You can check this by inserting a logging statement, or setting a breakpoint on the line.
        IF the program executed 'line'
            Analyze the line to determine its role in the overall behavior of the program
            # Check for errors such as the wrong function being called, the wrong argument being passed to a
            # function, the wrong variable being referenced, or a wrong operator being used
            IF any part of 'line' is inconsistent with its purpose
                # You found the bug
                RETURN 'line'
            # If the output statement is not wrong, perhaps the line was not supposed to execute at all?
            IF 'line' was not supposed to execute at all
                # The conditional might be in the same function as the output statement, or it might have been a
                # conditional in a function that called this function. Check the call stack if necessary by setting a
                # breakpoint. Find the conditional that led this line to being executed
                # Some value in the conditional's boolean expression must have been wrong. Which value was it?
                SET 'wrongValue' to the value in the conditional's boolean expression that ultimately allowed the
                    faulty output to execute
                # We'll use another strategy to find the source of the incorrect value.
                RETURN  localizeWrongValue('wrongValue')
            # If the line was supposed to execute, but it executed with an incorrect value, find that value.
            IF 'line' executed with an incorrect value
                SET 'wrongValue' TO the incorrect value
            # We'll use another strategy to find the cause of the incorrect value.
            RETURN localizeWrongValue('wrongValue')
    # If you made it to this line, then you must have missed something. Is it possible you made a mistake above? If
    # so,  go back and verify your work, because something caused the faulty output.
    RETURN nothing

STRATEGY localizeWrongValue(wrongValue)
    # The approach of this strategy is to recursively search backwards for the source of a value.
    # We begin by finding all of the lines of code that could have produced the wrong value.
    # For example, if a value was stored in a variable, we would find all of the assignments to that variable
    # that could have defined the variables current value. If it was a function's return value,
    # find the return statements that returned that wrong value.
    # These lines include expressions that computed the wrong value, a value passed in through a parameter, or a
    # function call that returned a value. Inspect the code to find the source of the incorrect value.
    SET 'lines' to all of the lines of the the program that could have produced 'wrongValue'
    # We'll check each line for errors, or for faulty values.
    FOR EACH 'line' in 'lines'
        # Use a logging statement or a breakpoint to verify that this line actually executed.
        IF 'line' executed
            # Does the line incorrectly compute the value? If so, you found the defect!
            IF 'line' is defective
                RETURN 'line'
            # If the line itself wasn't defective, maybe one of the values it used to execute was defective.
            SET 'badValue' TO any incorrect value used by the line to execute
            IF 'value' isn't nothing
                RETURN localizeWrongValue('badValue')
    # If you made it to this line, then you didn't find the cause of the wrong value. Is it possible you made a mistake
    # above? If so, check your work and start over.
    RETURN nothing
```

2 Test Driven Development Strategy

```
# This is a strategy for doing design by writing tests first and then making sure code passes the tests.
# This strategy requirements one parameter: requirements. Requirements describe in detail what your
# application should do. They often take the form of text explaining how your application should behave,
# particularly in response to specific inputs. In this strategy, you will need to reference the requirements
# for your application. You can either copy them here or simply refer to them being elsewhere.
STRATEGY testDrivenDevelopment(requirements)
    # The first step in test-driven development is enumerating all of the user scenarios.
    # You want to ensure that you enumerate specific requirements that are focused and small and
    # can be described in a sentence or less. You should try to find all of the user scenarios which might exist.
    # You should separate each scenario with a comma.
    SET 'scenarios' to be short descriptions of the testable user scenarios in requirements
    FOR EACH 'scenario' IN 'scenarios'
        Create a new test for scenario
        # Check that the new test demonstrates that the scenario is not yet implemented by checking that it does not pass
        UNTIL the new test does not pass
            Fix the new test so that it does not pass
        # Make it work
        Implement the code to make the test pass
        Run the tests
        UNTIL all of the tests pass
            Edit the code to address the test failure
            Run the tests
        # Make it right
        # Look to see if there any issues that make the design less than ideal.
        SET 'designIssue' TO be an unaddressed design issue if any or nothing otherwise
        UNTIL 'designIssue' is nothing
            Edit the code to fix the design issue
            Run the tests
            SET 'designIssue' TO be an unaddressed design issue if any or nothing otherwise
        # Make it fast
        # Look to see if there any performance issues that might cause it to be slow in some circumstances
        SET 'perfIssue' TO be an unaddressed performance issue if any or nothing otherwise
        UNTIL 'perfIssue' is nothing
            Edit the code to fix the performance issue
            Run the tests
            SET 'perfIssue' TO be an unaddressed performance issue if any or nothing otherwise
```

3 Merge GitHub Branches Strategy

```
\begin{scriptsize}
# This Strategy helps you merge 2 branches in githup and resolve conflicts
STRATEGY GitMerge()
        # Open the teminal, and use cd(change directory) command to move to the local git pr
        Open the terminal and navigate to your git project directory
        IF you are not in the master branch
                # Run the command git checkout master
                checkout to the master branch
        IF you are in the master branch
                # To merge the second branch with the master branch run the command "git mer
                Merge the two branches
                IF the merge has a conflict
                        SET 'conflictedFiles' TO the project files that have a conflict
                        FOR EACH 'file' IN 'conflictedFiles'
                                DO fixConflict('file')
                # Run GIT STATUS to see the latest changes
                # Run GIT ADD
                # Run GIT COMMIT -m ""
                # Run GIT PUSH
                Commit and push the changes
        RETURN nothing

STRATEGY fixConflict(conflictedFile)
        Open the conflictedFile with your favorite text editor
        # To find a line with conflict in your file, search the file for the conflict marke
        SET 'line' TO a line number that has conflict
        # until there is no line with <<<<<<< HEAD tag in the file
        UNTIL the file has no lines of conflict
                # You'll see the changes from the master branch after the line <<<<<<< HEAD
                # Next youll see ======= which divides 2 merged lines followed by >>>>>>> SH
                # Decide if you want to keep only your branch's changes, keep only the other
                # Also remove >>>>>>Head & ====== from the lines and make sure that the file
                Edit the file following the description
                SET 'line' TO the next line number that has conflict
        Return nothing
```

4 Tower of Hanoi Strategy

```
\begin{scriptsize}# Enter text for the following variables below.
```

```
# Let 'level' be the number of discs, 'source' be A, the
# leftmost peg, 'target' be C, the rightmost peg, and
# 'auxiliary' be B, the center peg
STRATEGY towerOfHanoi(level source target auxiliary)
        SET 'topDiscs' TO 'level' minus one
        IF 'level' greater than 1
                # We will now jump to a substrategy to move some discs to the auxilary.
                DO towerOfHanoi('topDiscs' 'source' 'auxiliary' 'target')
        Move the disc at 'source' to 'target'
        IF 'level' greater than 1
                # We will now jump to a substrategy to move discs back from the auxilary.
                DO towerOfHanoi('topDiscs' 'auxiliary' 'target' 'source')
```