Q1

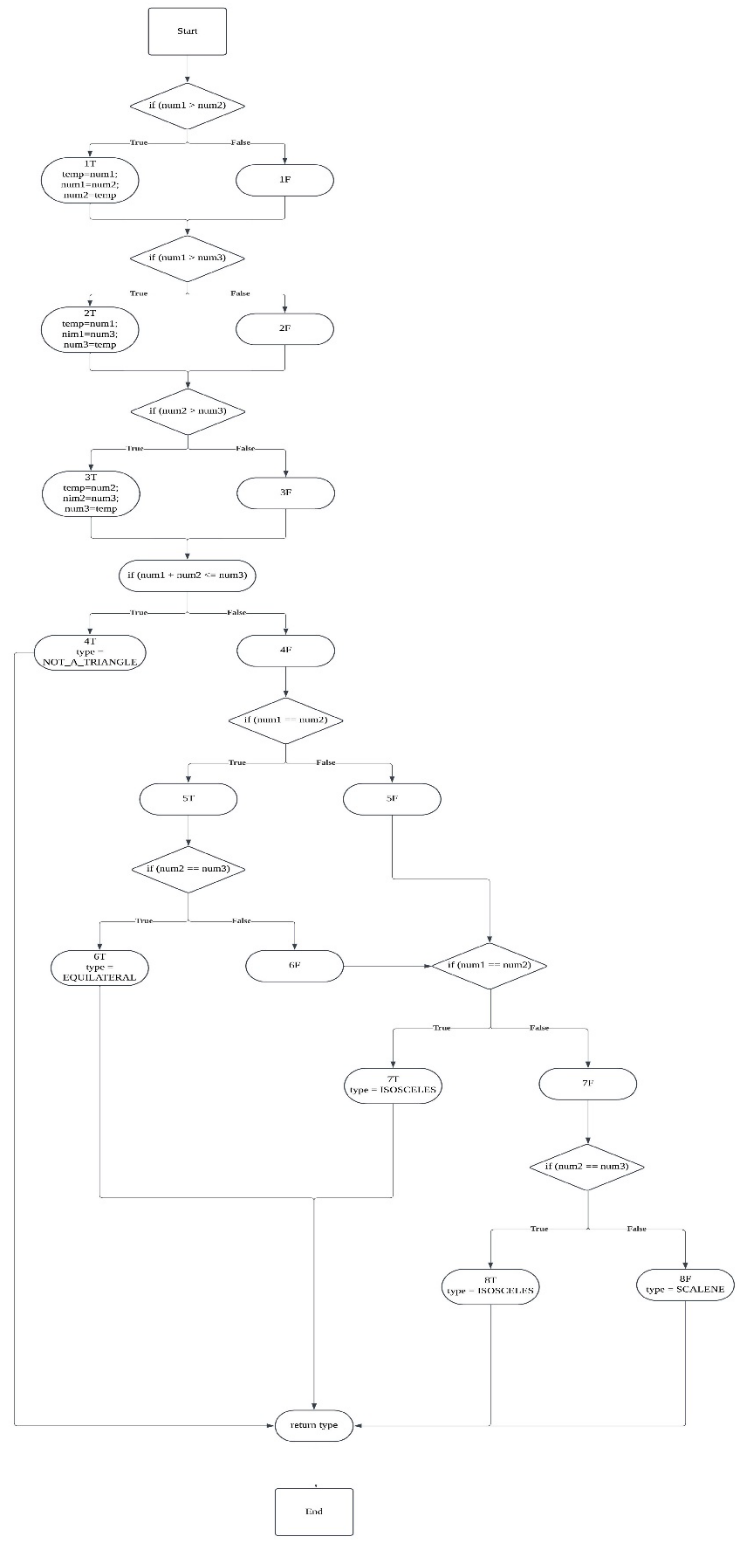
AVM operates with two movement modes: exploratory moves and pattern moves. AVM initially performs an exploratory move, where it makes a small change to one of the variables to check if moving in that direction improves the result. If the change improves the result, AVM proceeds with a pattern move in that direction. The initial step size for a pattern move is 1, which then increases. Once it reaches a point where the fitness declines, indicating it has surpassed the optimal solution, AVM will revert to the point before the decline and attempt an exploratory move with another variable. AVM optimizes each variable in this way until, after a round of optimization, all variables remain unchanged, signifying that AVM has found a local optimum.

After finding a local optimum, AVM provides a restart mechanism to escape the current local optimum. AVM restarts the process by making a random change to the vector, attempting to find a different solution.

There are some notable differences between AVM and Hill Climbing. First, the optimization method for variables is different: AVM optimizes each variable sequentially, moving to the next only after one variable is optimized, while Hill Climbing optimizes all variables in the vector simultaneously. Second, Hill Climbing lacks the "acceleration" method found in AVM's pattern move. This means that if Hill Climbing starts far from a local optimum with small steps, it will take significantly longer than AVM. Finally, Hill Climbing lacks AVM’s restart mechanism, meaning it cannot escape once it reaches a local optimum.

There are also significant differences between AVM and Simulated Annealing algorithms. Although AVM has a restart mechanism, it still cannot guarantee finding the global optimum. Simulated Annealing algorithms, however, use "temperature" adjustments: at higher "temperatures" early on, they accept poorer solutions to avoid local optima. As the "temperature" decreases, they progressively reject less optimal solutions, ensuring that the algorithm eventually finds the global optimum.

Q2


Start
if (num1 > num2)
True
False
1T
temp=num1;
num1=num2;
num2=temp
1F
if (num1 > num3)
True
False
2T
temp=num1;
nim1=num3;
num3=temp
2F
if (num2 > num3)
True
False
3T
temp=num2;
nim2=num3;
num3=temp
3F
if (num1 + num2 <= num3)
True
False
4T
type =
NOT_A_TRIANGLE
4F
if (num1 == num2)
True
False
5T
5F
if (num2 == num3)
True
False
6T
type =
EQUILATERAL
6F
if (num1 == num2)
True
False
7T
type = ISOSCELES
7F
if (num2 == num3)
True
False
8T
type = ISOSCELES
8F
type = SCALENE
return type
End

Q3

To achieve branch coverage for the classify method, the following test cases exercise both the true and false outcomes for each conditional branch in the method.

Test Case 1
Input: classify(3, 4, 5)
Expected Output: TriangleType.SCALENE
Description: This case covers the branches where num1, num2, and num3 are sorted without triggering any equality checks for ISOSCELES or EQUILATERAL.

Test Case 2
Input: classify(1, 2, 3)
Expected Output: TriangleType.NOT_A_TRIANGLE
Description: This case tests the branch where num1 + num2 <= num3, setting type to NOT_A_TRIANGLE.

Test Case 3
Input: classify(2, 2, 3)
Expected Output: TriangleType.ISOSCELES
Description: This case covers the path where num1 == num2 but num2 != num3, setting type to ISOSCELES.

Test Case 4
Input: classify(3, 3, 3)
Expected Output: TriangleType.EQUILATERAL
Description: This case covers the path where all sides are equal (num1 == num2 and num2 == num3), setting type to EQUILATERAL.

Test Case 5
Input: classify(2, 3, 2)
Expected Output: TriangleType.ISOSCELES
Description: This case exercises the sorting branches and covers the condition num1 != num2 && num2 == num3 for ISOSCELES.

Test Case 6
Input: classify(5, 3, 4)
Expected Output: TriangleType.SCALENE
Description: This case exercises the sorting logic and confirms that SCALENE is returned when no two sides are equal.

These test cases ensure that every branch in the classify method is exercised at least once, achieving comprehensive branch coverage.

Q4

Smallest test cases:

Test Case 1: (a = 2, b = 3)
Covers: b > a (True for if (b > a)), executes b = b - a; Print b.
Branches: Covers the true branch of if (b > a).

Test Case 2: (a = 3, b = 2)
Covers: a > b (True for if (a > b)), executes b = a - b; Print b.
Branches: Covers the true branch of if (a > b).

Test Case 3: (a = 0, b = 0)
Covers: a == b (True for if (a == b)) and a == 0 (True for nested if (a == 0)), executes Print '0'.
Branches: Covers the true branch of if (a == b) and true branch of nested if (a == 0).

Branches Covered:
Test Case 1 covers the true branch of if (b > a), the false branch of if (a > b), and the false branch of if (a == b).
Test Case 2 covers the false branch of if (b > a), the true branch of if (a > b), and the false branch of if (a == b).
Test Case 3 covers the false branches of if (b > a) and if (a > b), the true branch of if (a == b), and the true branch of if (a == 0).
Therefore, the given code also achieves branch coverage.

Q5

Test Suite
Test Case 1: Two lines that intersect within the segment bounds.
Input: Line1: (0,0) to (4,4), Line2: (0,4) to (4,0)
Expected Output: Intersection point at (2,2)

Test Case 2: Two non-parallel lines that do not intersect within the segment bounds.
Input: Line1: (0,0) to (1,1), Line2: (2,2) to (3,3)
Expected Output: No intersection

This test suite achieves statement coverage but does not achieve branch coverage because it does not cover the following branches:
Parallel Lines Check: This branch is not covered as neither test case involves parallel lines.
Collinear Lines Check: This branch is not covered since neither test case involves collinear lines.

Q6

In the AVMf framework, the fitness function combines two metrics—approach level and branch distance—to evaluate and compare test inputs. The approach level measures how close a test input is to reaching the target branch in terms of control dependencies. A lower approach level means the test input is structurally closer to the target, and this metric serves as the primary criterion for comparison. If two test inputs have the same approach level, the framework then uses the branch distance metric as a secondary criterion. The branch distance measures how close the input is to satisfying a specific branch condition, with smaller distances indicating closer proximity to meeting the condition.

This combined comparison is implemented in the NumericalObjectiveValue class within the AVMf framework. Specifically, the betterThan(ObjectiveValue other) method in NumericalObjectiveValue prioritizes approach level first, and only compares branch distances if the approach levels are equal.

Q7

We need to normalize the branch distance metric but not the approach level because these two metrics are on different scales and serve different purposes.

The approach level is an integer that represents the number of control dependencies (e.g., conditional statements) between the point reached by a test input and the target branch. Since approach level values are generally small and discrete, they are naturally comparable without requiring normalization.

On the other hand, the branch distance is a continuous metric that can vary widely, depending on how far the test input is from satisfying a specific branch condition. Without normalization, the branch distance values could dominate the combined score, overshadowing the approach level. By normalizing the branch distance, we scale it to a comparable range, allowing the combined metric (approach level + normalized branch distance) to accurately represent both structural proximity (approach level) and closeness to condition satisfaction (branch distance).