Lab 7 - IT314

Name - Jaikrit Sanandiya
ID - 202201484

Task 1 : Program Inspection:

Code Fragment - 1

1. How many errors are there in the program? Mention the errors you have identified.

Data reference errors:

- In the `CvHOGEvaluator::setImage` method, the use of `hist` and `normSum` might cause issues if these matrices are not properly initialized before use. There is no explicit check ensuring they contain valid data before operations are performed.
- **Pointer errors**: The code uses direct memory access via `ptr()` calls, which is prone to dangling references if the matrices have been reallocated or altered in size between calls.

Data-declaration errors:

- Variables like `grad`, `qangle`, and `integralHist` in the `integralHistogram` method are critical for calculations, but there's no check to verify if the matrices are of correct dimensions or types before they are used.
- The size of `AutoBuffer<int> mapbuf(gradSize.width + gradSize.height + 4)` might lead to errors if the image size changes and exceeds buffer capacity.

Computation errors:

- In `generateFeatures`, there are several hard-coded values for `blockStep` and iterations over `winSize` that might cause unexpected behavior if the window size is changed or unusual values are passed.
- **Order of precedence** in complex arithmetic expressions within `CvHOGEvaluator::integralHistogram` could lead to incorrect results without proper debugging, especially with polar coordinate calculations.

Control-flow errors:

- The loops in `generateFeatures` method are heavily nested, and missing boundary checks or incorrect loop control could result in performance issues or off-by-one errors.

2. Which category of program inspection would you find more effective?

- Given the nature of this program, the Data Reference Errors category is the most effective. Since the code deals heavily with matrix operations and pointer manipulations, ensuring correct memory handling, pointer validity, and array bounds are crucial. It also uses OpenCV functions that rely on correct data initialization, which makes this category more relevant.

3. Which type of error you are not able to identified using the program inspection?

- Using program inspection alone, errors related to runtime performance or resource handling (such as memory leaks) may not be easily detected. While we can review the code for correct logic and structure, issues such as excessive memory consumption, poor cache usage, or inefficient operations would need dynamic analysis tools (profilers or memory analyzers) to identify during actual execution.

4. Is the program inspection technique is worth applicable?
    - Yes, program inspection techniques are particularly worth applying for this code. The checklist-based approach helps uncover potential memory handling issues, loop control flaws, and uninitialized data that are difficult to catch during normal compilation. Especially in low-level operations with pointers and OpenCV matrices, careful inspection can prevent hard-to-trace bugs, although it should be supplemented with dynamic analysis for comprehensive debugging.

Code Fragment - 2

1. How many errors are there in the program? Mention the errors you have identified.

Data Reference Errors:

- There are multiple references to arrays such as `stageTypes[]` and `featureTypes[]` that depend on correct indexing. Improper index bounds could result in accessing memory out of range, especially in methods like `printDefaults` and `scanAttr`.
- In the `CvCascadeClassifier::predict` method, there is a potential issue if `stageClassifiers` is empty or not fully initialized. Accessing uninitialized or improperly loaded elements could cause runtime failures.

Data-declaration errors:

- The variables `stageType` and `featureType` in the constructor of `CvCascadeParams` are initialized using constants like `defaultStageType` and `defaultFeatureType`. However, if these constants are not properly defined or assigned, it might cause unexpected behavior.
- In `CvCascadeClassifier::fillPassedSamples`, there is a `Mat img` variable that depends on `cascadeParams.winSize`. If `winSize` is improperly initialized (e.g., set to negative dimensions), it could result in unexpected errors when processing images.

Computation Errors:

- In `CvCascadeClassifier::train`, the calculation of `requiredLeafFARate` could encounter floating-point inaccuracies due to the `pow` function and division of `maxFalseAlarm` by `max_depth`. If any input values are unusual (e.g., zero), it could cause divide-by-zero errors.
- In `CvCascadeClassifier::fillPassedSamples`, there is an implicit assumption that the division `((double)getcount+1)/(double)(int64)consumed` will not result in divide-by-zero. However, if `consumed` is zero, this can cause a crash.

Control-flow errors:

- In the `train` method, there are several `break` conditions for loop termination, such as checking `tempLeafFARate`. However, if the conditions do not behave as expected (e.g., improper updates to `tempLeafFARate`), the loop might not terminate properly, causing infinite loops.
- There is a risk of **non-exhaustive decisions** in methods like `read` and `readStages`. If the file structure differs from expectations (e.g., missing or corrupted XML nodes), the function might fail silently without giving the user clear feedback on why it failed.

2. Which category of program inspection would you find more effective?

- The Data Reference Errors category is again the most effective for this code. This is because it deals extensively with accessing external files (e.g., training data, XML files for parameters), and improper file reading or memory allocation could lead to major issues. Ensuring that data is properly referenced and initialized is key to preventing memory errors or segmentation faults.

3. Which type of error you are not able to identified using the program inspection?

As in the previous case, errors related to runtime performance or resource handling are hard to identify with just static inspection. For example:

- Memory leaks could happen if dynamically allocated memory is not freed (e.g., through ptr objects), but static inspection won't detect this.
- The performance bottlenecks, such as inefficiencies in loops or recursive calls, would only become evident with dynamic analysis.

4. Is the program inspection technique is worth applicable?

- Yes, program inspection is very much worth applying for this code. The detailed error checklist helps uncover potential memory access issues, file handling problems, and uninitialized data references, which are crucial in a codebase that interacts with external resources (such as files) and employs complex image processing logic. It ensures early detection of issues that could lead to runtime crashes.

Code Fragment - 3

1. How many errors are there in the program? Mention the errors you have identified.

Data Reference Errors:

- In the `CvHaarEvaluator::setImage` method, there is no check to ensure the image (`img`) passed has the correct size or is valid. If an invalid image or one of unexpected dimensions is passed, it could cause undefined behavior when calling `integral()` or manipulating the feature vectors.
- In `CvHaarEvaluator::generateFeatures`, the nested loops that iterate through `winSize.width` and `winSize.height` heavily depend on properly defined window sizes. If `winSize` has invalid or negative values, this would result in out-of-bound memory accesses when generating features.
- **Pointer safety issues**: In the `Feature::Feature` constructor, the `CV_SUM_OFFSETS` and `CV_TILTED_OFFSETS` macros manipulate pointers directly. If `rect[j].r` holds invalid data or if the offset is calculated incorrectly, this could lead to incorrect memory access.

Data Declaration Errors:

- In `CvHaarEvaluator::init`, the matrix initialization with `sum`, `tilted`, and `normfactor` relies on the correctness of `_winSize` and `_maxSampleCount`. Any unexpected values for these parameters (such as zero or negative values) might lead to errors in matrix creation.

- In `CvHaarEvaluator::Feature::Feature`, the feature constructor depends on the `offset` and feature dimensions (`x`, `y`, `dx`, `dy`) for correct initialization of the feature rectangles. Without bounds checking on these variables, invalid values could be passed that may lead to out-of-bounds memory access or undefined behavior.

Computation Errors:

- In `CvHaarEvaluator::generateFeatures`, complex arithmetic expressions with `dx`, `dy`, and feature offsets are used for calculating the Haar features. These calculations might overflow if the feature window (`winSize`) is too large, causing unexpected behavior in feature generation.
- In `CvHaarEvaluator::setImage`, the calculation of the normalization factor (`normfactor.ptr<float>(0)[idx] = calcNormFactor(innSum, innSqSum);`) depends on `calcNormFactor()`. Without additional checks, this could result in incorrect values if the sum or squared sum are not calculated properly.

Control-flow errors:

- In `CvHaarEvaluator::read`, if an invalid or corrupted configuration is passed via the `FileNode`, the function might fail silently without providing detailed feedback on why the feature parameters failed to load. This could leave the feature parameters in an inconsistent state.
- There is a potential **loop termination issue** in `generateFeatures`. If the feature generation process runs into invalid conditions (e.g., window size issues), the loop could fail to terminate properly, resulting in performance bottlenecks or infinite loops.

2. Which category of program inspection would you find more effective?

- The Data Reference Errors category is the most effective for this code as well. Since the code deals heavily with matrix operations, image processing, and feature extraction using OpenCV, ensuring that all data is correctly initialized, accessed, and manipulated is crucial. Proper memory handling and avoiding out-of-bounds access in arrays or matrices will prevent a significant number of runtime errors.

3. Which type of error you are not able to identified using the program inspection?

Static inspection alone won't help identify runtime performance bottlenecks or memory-related issues like:

- Memory leaks: Due to dynamic memory allocations in the mat objects and feature vectors, memory could be leaked over time if resources are not properly released.
- Performance issues: The efficiency of feature generation depends heavily on the size of the window and input parameters. These issues would only surface during actual execution and profiling.

Additionally, floating-point precision errors in calculations like normfactor are difficult to detect without dynamic analysis.

4. Is the program inspection technique is worth applicable?

- Yes, the program inspection technique is worth applying to this code. The inspection helps identify possible data reference issues, boundary checks, and proper initialization, which are critical in ensuring the robustness of feature extraction algorithms. These types of checks are particularly useful when working with low-level matrix manipulations and pointer arithmetic that, if mishandled, can result in hard-to-trace bugs. It should, however, be supplemented with dynamic analysis for runtime errors and performance profiling.

Code Fragment - 4

1. How many errors are there in the program? Mention the errors you have identified.

- **Data Reference Errors**:
  - In `CvDTreeTrainData::set_data`, the handling of several dynamically allocated arrays (e.g., `buf`, `int_ptr`, `priors`, `cat_map`) could lead to memory issues if not properly initialized or released in the event of an error. If these resources are not managed correctly, it could result in memory leaks.
  - The `responses_copy` is used to store a copy of the responses, but the code does not check if the responses matrix is valid and correctly initialized before performing the copy in `do_responses_copy`. If the responses matrix is invalid or null, it could lead to undefined behavior.
  - The handling of `_responses`, `_train_data`, and other matrices assumes they are valid. There are no checks to verify if the input matrices are non-empty, correctly sized, or of compatible types before processing.
- **Data-declaration errors**:
  - The array `var_type` is dynamically allocated in `CvDTreeTrainData::set_data` using `cvCreateMat`. However, if the allocation fails or if the matrix is used before initialization, it could lead to errors. There is no explicit check ensuring successful memory allocation before the matrix is used.
  - The `priors` matrix is created based on the number of classes in `set_data`, but if the number of classes (`get_num_classes()`) returns an unexpected value (like zero), it might lead to incorrect initialization of this matrix.
- **Computation Errors**:
  - In `set_params`, the conditions for setting `params.max_depth` and `params.max_categories` involve several minimum and maximum constraints (e.g., `params.max_categories =`

`MIN(params.max_categories, 15);`). If these constraints are violated, especially with edge-case values for the parameters, it could result in unintended behavior or errors.
    - There is a risk of **overflows** when calculating the memory buffer sizes in `effective_buf_size` and `effective_buf_height`. The code attempts to handle large datasets by calculating buffer sizes for the training data, but if the size exceeds the limits of integer or memory allocations, it could lead to crashes or memory errors.
- **Control-flow errors**:
    - In `CvDTreeTrainData::set_data`, the code checks the consistency of new and old training data (`cvNorm(...)`). If the check fails, the error message could be improved to provide more specific information about why the new data is incompatible. This would aid in debugging, as the current message is somewhat vague.
    - The `CV_FUNCNAME("CvDTreeTrainData::set_data")` calls are used to manage the error handling with `__BEGIN__` and `__END__`, but if the flow of the program jumps between different error-handling blocks, it could lead to inconsistent states of dynamically allocated resources (like `tree_storage` and `temp_storage`), leaving them unreleased.

2. Which category of program inspection would you find more effective?

- The Data Reference Errors category is the most effective for this code as well, especially given that the code uses several dynamically allocated arrays, buffers, and matrices (e.g., `priors`, `buf`, `responses_copy`, etc.). Ensuring that all memory is properly allocated, initialized, and released is critical to prevent memory leaks, segmentation faults, or access to uninitialized memory.

3. Which type of error you are not able to identified using the program inspection?

Static inspection won't help identify runtime memory issues or performance bottlenecks, such as:

- Memory leaks: Although inspection helps identify potential issues with memory management, actual leaks (where memory is not released) would require dynamic analysis tools (such as valgrind).
- Performance inefficiencies: The code handles large datasets, so performance bottlenecks related to sorting or memory copying could arise when running on large-scale data. These would require profiling to identify.
- Concurrency issues: The code does not handle concurrency, but if multiple threads were used to handle data in parallel, static inspection wouldn't help detect potential race conditions or deadlocks.

4. Is the program inspection technique is worth applicable?

- Yes, program inspection is valuable for this code. The checklist helps identify potential data reference issues, memory allocation problems, and boundary conditions that could cause runtime errors. Given that this code deals with dynamic memory allocation and complex data structures, ensuring that all resources are correctly managed is essential to maintaining robustness and preventing crashes. However, it should be complemented by dynamic analysis to fully catch runtime and performance issues.

Task 2 : CODE DEBUGGING:

Code 1 : Armstrong

1. How many errors are there in the program? Mention the errors you have identified.

Errors:

- Logic error in remainder calculation: The remainder is calculated using division (`remainder = num / 10;`) instead of the modulus operator. This results in incorrect values for the remainder, breaking the logic for calculating Armstrong numbers.
- Logic error in updating `num`: The line `num = num % 10;` should instead be `num = num / 10;` because after extracting the remainder, the number should be reduced by removing the last digit, which is done by integer division.

2. How many breakpoints you need to fix those errors?
   a. What are the steps you have taken to fix the error you identified in the code fragment?

Breakpoints required:

- Breakpoint 1: At the start of the `while` loop to check how `num` and `remainder` are being updated.
- Breakpoint 2: After the `if` statement to verify the final comparison of `check` and `n` and ensure that the logic has been properly applied.

Steps to fix the errors:

- **Fix the remainder calculation**: Change `remainder = num / 10;` to `remainder = num % 10;` to correctly extract the last digit of the number.
- **Fix the number update**: Change `num = num % 10;` to `num = num / 10;` to correctly reduce the number by removing its last digit after each iteration.
- **Fix grammatical error**: Update the `else` statement output to "is not an Armstrong Number" for grammatical correctness.

3. Submit your complete executable code.

```java
// Armstrong Number
class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num;
        int check = 0, remainder;

        while(num > 0) {
            remainder = num % 10;
            check = check + (int) Math.pow(remainder, 3);
            num = num / 10;
        }

        if(check == n)
            System.out.println(n + " is an Armstrong Number");
        else
            System.out.println(n + " is not an Armstrong Number");
    }
}
```

Code 2 : GCD_LCM

1. How many errors are there in the program? Mention the errors you have identified.

Errors:

- **GCD Logic Error**: The `while` loop condition in the `gcd` function is incorrect. It should be `while(a % b != 0)` instead of `while(a % b == 0)`. The current logic would terminate prematurely and return an incorrect GCD.
- **LCM Logic Error**: The condition `if(a % x != 0 && a % y != 0)` in the `lcm` function is incorrect. It should be `if(a % x == 0 && a % y == 0)` because `a` should be divisible by both `x` and `y` to be the least common multiple.
- **LCM Efficiency Issue**: The `lcm` function can be improved for efficiency. Instead of incrementing `a` one by one, the LCM can be computed using the relationship `LCM(x, y) = (x * y) / GCD(x, y)`.

2. How many breakpoints you need to fix those errors?
   a. What are the steps you have taken to fix the error you identified in the code fragment?

Breakpoints required:

- **Breakpoint 1**: Inside the `gcd` function to observe how `a`, `b`, and `r` are updated, particularly after the modulus operation to ensure it loops until the remainder is zero.
- **Breakpoint 2**: In the `lcm` function, right after the condition `if(a % x != 0 && a % y != 0)` to verify if the condition is correctly identifying the least common multiple.
- **Breakpoint 3**: Just before returning the final GCD and LCM values to ensure they are correct.

Steps to fix the errors:

Fix the GCD Logic: Change the condition in the `while` loop in the `gcd` function to `while(a % b != 0)` to correctly calculate the greatest common divisor.

Fix the LCM Condition: Modify the condition in the `lcm` function to `if(a % x == 0 && a % y == 0)` to ensure `a` is divisible by both `x` and `y`.

Optimize LCM Calculation: Use the formula `LCM(x, y) = (x * y) / GCD(x, y)` to improve efficiency, avoiding the need for a loop that increments `a` unnecessarily.

3. Submit your complete executable code.

```java
import java.util.Scanner;

public class GCD_LCM {
    static int gcd(int x, int y) {
        int r = 0, a, b;
        a = (x > y) ? x : y; // a is the greater number
        b = (x < y) ? x : y; // b is the smaller number

        r = b;
        while (b != 0) { // Fixed the logic to check until b becomes 0
            r = a % b;
            a = b;
            b = r;
        }
        return a; // Return the GCD, which is stored in 'a' at the end
    }

    static int lcm(int x, int y) {
        return (x * y) / gcd(x, y); // Use formula for LCM based on GCD for efficiency
    }

    public static void main(String args[]) {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter the two numbers: ");
        int x = input.nextInt();
        int y = input.nextInt();

        System.out.println("The GCD of two numbers is: " + gcd(x, y));
        System.out.println("The LCM of two numbers is: " + lcm(x, y));
        input.close();
    }
}
```

Code 3 : Knapsack

1.How many errors are there in the program? Mention the errors you have identified.

Errors:

- Logic error in the array access (`opt[n++][w]`): The expression `opt[n++][w]` in the loop increments `n` by 1 after accessing `opt[n][w]`. This causes the loop to skip subsequent iterations and breaks the logic. It should be `opt[n-1][w]` to correctly access the previous item's value.
- Logic error in calculating `option2`: The condition for `option2` is wrong. The code checks if `weight[n] > w`, but the correct check should be `weight[n] <= w` to ensure that the weight of the current item can fit into the knapsack. Additionally, the value of `option2` should use `profit[n]` (not `profit[n-2]`) to calculate the current profit.
- Incorrect loop update for determining items to take: The loop where the items to be taken are determined is correct, but the weight `w` is decremented by the wrong value in some cases because of the faulty `option2` calculation.

2.How many breakpoints you need to fix those errors?

a. What are the steps you have taken to fix the error you identified in the code fragment?

Breakpoints required:

- **Breakpoint 1**: In the nested loop that calculates the `opt[n][w]` values. This is to observe how the decision variables (`option1` and `option2`) are being calculated and ensure that the logic is followed correctly.
- **Breakpoint 2**: In the loop that determines which items are taken. This will help verify if the selected items and the remaining weight `w` are updated as expected.

Steps to fix the errors:

- **Fix the array access**: Change `opt[n++][w]` to `opt[n-1][w]` to correctly reference the value for item `n-1` without skipping the iteration.
- **Fix the logic for `option2`**: Update the condition to `if(weight[n] <= w)` to ensure that we only take the item if its weight is less than or equal to the current available weight. Also, fix the calculation of `option2` to use `profit[n]` instead of `profit[n-2]`.
- **Fix the weight update for determining items**: Ensure that the weight `w` is decremented correctly based on the current item's weight after the correct item is selected.

3. Submit your complete executable code.

```java
// Knapsack Problem
public class Knapsack {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);   // number of items
        int W = Integer.parseInt(args[1]);   // maximum weight of knapsack
        int[] profit = new int[N + 1];
        int[] weight = new int[N + 1];
        // generate random instance, items 1..N
        for (int n = 1; n <= N; n++) {
            profit[n] = (int) (Math.random() * 1000);
            weight[n] = (int) (Math.random() * W);
        }
        int[][] opt = new int[N + 1][W + 1];
        boolean[][] sol = new boolean[N + 1][W + 1];
        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {
                int option1 = opt[n - 1][w];
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w) {  // Corrected logic to ensure weight fits
                    option2 = profit[n] + opt[n - 1][w - weight[n]];
                }
                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
        }
        boolean[] take = new boolean[N + 1];
        for (int n = N, w = W; n > 0; n--) {
            if (sol[n][w]) {
                take[n] = true;
                w = w - weight[n];  // Update the weight correctly
            } else {
                take[n] = false;
            }
        }
        System.out.println("item" + "\t" + "profit" + "\t" + "weight" + "\t" + "take");
        for (int n = 1; n <= N; n++) {
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" + take[n]);
        }
    }
}
```

Code 4 : magic number

1.How many errors are there in the program? Mention the errors you have identified.

Errors:

- Logic error in inner while loop: The inner `while` loop has an incorrect condition `sum == 0`. This will cause the loop to never execute since `sum` starts as a positive number. The correct condition should be `sum > 0` to ensure the sum is processed digit by digit.
- Logic error in calculating `s`: The expression `s = s * (sum / 10)` is incorrect. The goal is to sum the digits, not multiply them. It should be `s = s + (sum % 10)` to add each digit to `s`.
- Syntax error: The line `sum = sum % 10` is missing a semicolon (`;`). This will cause a compilation error.

2.How many breakpoints you need to fix those errors?

a. What are the steps you have taken to fix the error you identified in the code fragment?

Breakpoints required:

- **Breakpoint 1**: After entering the first `while(num > 9)` loop to verify how `num` and `sum` are updated.
- **Breakpoint 2**: Inside the inner `while(sum > 0)` loop to ensure each digit is being correctly processed and added to the sum.
- **Breakpoint 3**: Before the final `if(num == 1)` statement to confirm that the correct value of `num` is being used for the Magic Number check.

Steps to fix the errors:

- **Fix the condition for the inner loop**: Change `while(sum == 0)` to `while(sum > 0)` to correctly process each digit.

- **Fix the logic for summing digits**: Change `s = s * (sum / 10)` to `s = s + (sum % 10)` to correctly sum the digits of the number.
- **Add missing semicolon**: Add the missing semicolon in the line `sum = sum % 10`.

3. Submit your complete executable code.

```java
// Program to check if a number is a Magic Number in Java
import java.util.*;

public class MagicNumberCheck {
    public static void main(String args[]) {
        Scanner ob = new Scanner(System.in);
        System.out.println("Enter the number to be checked.");
        int n = ob.nextInt();
        int sum = 0, num = n;

        while (num > 9) {
            sum = num;
            int s = 0;
            while (sum > 0) {  // Corrected condition to process the sum
                s = s + (sum % 10);  // Corrected to sum the digits
                sum = sum / 10;  // Added missing semicolon and corrected the division
            }
            num = s;
        }

        if (num == 1) {
            System.out.println(n + " is a Magic Number.");
        } else {
            System.out.println(n + " is not a Magic Number.");
        }

        ob.close();  // Closing the scanner to prevent resource leak
    }
}

```

Code 5 : merge sort

1.How many errors are there in the program? Mention the errors you have identified.

Errors:

- Incorrect array slicing in `mergeSort`: The expressions `int[] left = leftHalf(array+1);` and `int[] right = rightHalf(array-1);` are incorrect. Arrays cannot be directly manipulated with arithmetic operations. It should be `leftHalf(array)` and `rightHalf(array)` to correctly pass the original array for splitting.
- Incorrect array manipulation in merge: The expressions `merge(array, left++, right--);` are incorrect. Incrementing (`left++` and `right--`) changes the reference of the arrays, which leads to errors. These should be `merge(array, left, right);` to pass the correct array references.
- Off-by-one errors in array splitting: The splitting of arrays in the `rightHalf` method is correct, but in the `mergeSort` method, if the array has an odd length, the merging process might skip an element due to incorrect split logic.

2.How many breakpoints you need to fix those errors?

a. What are the steps you have taken to fix the error you identified in the code fragment?

Breakpoints required:

- Breakpoint 1: After splitting the array into `left` and `right` halves, to verify if the array was correctly split and ensure the arrays contain the expected elements.
- Breakpoint 2: Inside the `merge` method, to verify that the merging logic correctly processes and merges the two sorted halves into the result array.

Steps to fix the errors:

- Fix array slicing: Change the incorrect expressions in `mergeSort` from `leftHalf(array+1)` and `rightHalf(array-1)` to `leftHalf(array)` and `rightHalf(array)` to correctly pass the array for splitting.
- Fix array references in merge: Change `merge(array, left++, right--);` to `merge(array, left, right);` to pass the correct array references.
- Verify array splits: Ensure that the array is split correctly when passed to `leftHalf` and `rightHalf`.

3. Submit your complete executable code.

```java
// This program implements the merge sort algorithm for
// arrays of integers.

import java.util.*;

public class MergeSort {
    public static void main(String[] args) {
        int[] list = {14, 32, 67, 76, 23, 41, 58, 85};
        System.out.println("before:  " + Arrays.toString(list));
        mergeSort(list);
        System.out.println("after:  " + Arrays.toString(list));
    }

    // Places the elements of the given array into sorted order
    // using the merge sort algorithm.
    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            // split array into two halves
            int[] left = leftHalf(array);  // Corrected
            int[] right = rightHalf(array);  // Corrected

            // recursively sort the two halves
            mergeSort(left);
            mergeSort(right);

            // merge the sorted halves into a sorted whole
            merge(array, left, right);  // Corrected
        }
    }

    // Returns the first half of the given array.
    public static int[] leftHalf(int[] array) {
        int size1 = array.length / 2;
        int[] left = new int[size1];
        for (int i = 0; i < size1; i++) {
            left[i] = array[i];
        }
        return left;
    }
```

```java
        return left;
    }

    // Returns the second half of the given array.
    public static int[] rightHalf(int[] array) {
        int size1 = array.length / 2;
        int size2 = array.length - size1;
        int[] right = new int[size2];
        for (int i = 0; i < size2; i++) {
            right[i] = array[i + size1];
        }
        return right;
    }

    // Merges the given left and right arrays into the given result array.
    // pre : result is empty; left/right are sorted
    // post: result contains result of merging sorted lists
    public static void merge(int[] result, int[] left, int[] right) {
        int i1 = 0;   // index into left array
        int i2 = 0;   // index into right array

        for (int i = 0; i < result.length; i++) {
            if (i2 >= right.length || (i1 < left.length &&
                    left[i1] <= right[i2])) {
                result[i] = left[i1];    // take from left
                i1++;
            } else {
                result[i] = right[i2];   // take from right
                i2++;
            }
        }
    }
}
```

Code 6 : multiply matrices

1.How many errors are there in the program? Mention the errors you have identified.

Errors:

- **Indexing errors in matrix multiplication**: In the innermost loop of the multiplication logic, the expressions `first[c-1][c-k]` and `second[k-1][k-d]` are incorrect. Matrix indices are not supposed to be decremented like this (`c-1` or `k-1`), as this leads to invalid access to negative indices. The correct indices should be `first[c][k]` and `second[k][d]`.
- **Incorrect loop range for multiplication**: The multiplication loop's third variable (`k`) should iterate over the number of columns in the first matrix (i.e., `n`) rather than `p`, which refers to the number of rows in the second matrix.

2.How many breakpoints you need to fix those errors?

a. What are the steps you have taken to fix the error you identified in the code fragment?

Breakpoints required:

- Breakpoint 1: At the start of the matrix multiplication loop, to check the intermediate values being calculated for `sum`, `first`, and `second`.
- Breakpoint 2: After calculating each element of the `multiply` matrix, to verify that the correct product is being stored in the result matrix.

Steps to fix the errors:

- Fix matrix indexing: Replace `first[c-1][c-k]` with `first[c][k]` and `second[k-1][k-d]` with `second[k][d]` to correctly access the matrix elements for multiplication.
- Fix the loop range for k: Ensure that k iterates over the number of columns in the first matrix (n) instead of p, since n is the dimension that both matrices share.

3. Submit your complete executable code.

```java
// Java program to multiply two matrices
import java.util.Scanner;

class MatrixMultiplication {
    public static void main(String args[]) {
        int m, n, p, q, sum = 0, c, d, k;

        Scanner in = new Scanner(System.in);
        System.out.println("Enter the number of rows and columns of first matrix");
        m = in.nextInt();
        n = in.nextInt();

        int first[][] = new int[m][n];

        System.out.println("Enter the elements of first matrix");

        for (c = 0; c < m; c++)
            for (d = 0; d < n; d++)
                first[c][d] = in.nextInt();

        System.out.println("Enter the number of rows and columns of second matrix");
        p = in.nextInt();
        q = in.nextInt();

        if (n != p)
            System.out.println("Matrices with entered orders can't be multiplied with each other.");
        else {
            int second[][] = new int[p][q];
            int multiply[][] = new int[m][q];

            System.out.println("Enter the elements of second matrix");

            for (c = 0; c < p; c++)
                for (d = 0; d < q; d++)
                    second[c][d] = in.nextInt();

            for (c = 0; c < m; c++) {
                for (d = 0; d < q; d++) {
                    for (k = 0; k < n; k++) {  // Fix: iterate over columns of the first matrix (n)
                        sum = sum + first[c][k] * second[k][d];  // Fix: correct indexing
                    }

                    multiply[c][d] = sum;
                    sum = 0;
                }
            }

            System.out.println("Product of entered matrices:-");

            for (c = 0; c < m; c++) {
                for (d = 0; d < q; d++)
                    System.out.print(multiply[c][d] + "\t");

                System.out.print("\n");
            }
        }
        in.close();  // Close the scanner to prevent resource leaks
    }
}
```

Code 7 : quadretic probing

1.How many errors are there in the program? Mention the errors you have identified.

Errors:

- Indexing errors in matrix multiplication:
  In the innermost loop of the matrix multiplication logic, the expressions `first[c-1][c-k]` and `second[k-1][k-d]` are incorrect. Matrix indices should not be decremented like this (`c-1` or `k-1`), as it results in invalid access to negative indices. The correct expressions should be `first[c][k]` and `second[k][d]` to access the appropriate matrix elements for multiplication.
- Incorrect loop range for multiplication (variable `k`):
  The loop variable `k` should iterate over the number of columns in the first matrix (`n`) instead of `p` (which refers to the number of rows in the second matrix). The common dimension between the two matrices is the number of columns in the first matrix and the number of rows in the second matrix.

2.How many breakpoints you need to fix those errors?

a. What are the steps you have taken to fix the error you identified in the code fragment?

Breakpoints required:

- Breakpoint 1: At the start of the matrix multiplication loop, to check the intermediate values being calculated for `sum`, `first`, and `second`.
- Breakpoint 2: After calculating each element of the result matrix (the `multiply` matrix), to verify that the correct product is being stored.

Steps to fix the errors:

- **Fix matrix indexing:**
  Correct the matrix access expressions by replacing `first[c-1][c-k]` with `first[c][k]` and `second[k-1][k-d]` with `second[k][d]` to ensure that the appropriate matrix elements are used for multiplication.
- **Fix the loop range for k:**
  Modify the loop to iterate over the number of columns in the first matrix (n), since this is the dimension shared between both matrices for multiplication.

3. Submit your complete executable code.

```java
public class MatrixMultiplication {
    public static void main(String[] args) {
        int r1 = 2, c1 = 3; // First matrix dimensions (2x3)
        int r2 = 3, c2 = 2; // Second matrix dimensions (3x2)

        // First matrix
        int[][] first = {
            {1, 2, 3},
            {4, 5, 6}
        };

        // Second matrix
        int[][] second = {
            {7, 8},
            {9, 10},
            {11, 12}
        };

        // Result matrix (r1 x c2)
        int[][] multiply = new int[r1][c2];

        // Multiplication logic
        for (int c = 0; c < r1; c++) {
            for (int d = 0; d < c2; d++) {
                int sum = 0;
                for (int k = 0; k < c1; k++) { // Fix the loop to iterate over columns of the first matrix
                    sum += first[c][k] * second[k][d]; // Corrected matrix access
                }
                multiply[c][d] = sum; // Store the product in the result matrix
            }
        }

        // Printing the result
        System.out.println("Product of the matrices:");
        for (int[] row : multiply) {
            for (int element : row) {
                System.out.print(element + " ");
            }
            System.out.println();
        }
    }
}
```

Code 8 : sorting array

1.How many errors are there in the program? Mention the errors you have identified.

Errors:

- **Error 1:** In the outer `for` loop for sorting, the condition is `i >= n`, which is incorrect. The loop should iterate while `i < n`, so it should be `i < n`.
- **Error 2:** There is a **semicolon** `;` after the outer `for` loop: `for (int i = 0; i >= n; i++);`. This semicolon ends the loop prematurely and should be removed.
- **Error 3:** The condition for swapping the elements in the nested loop is incorrect: `if (a[i] <= a[j])`. This is checking for elements that are already in ascending order, but we want to swap if the current element is greater, so it should be `if (a[i] > a[j])`.
- **Error 4:** In the output loop printing the sorted array, the loop condition should be `i < n`, not `i < n - 1`. The current condition skips printing the last element correctly.

2.How many breakpoints you need to fix those errors?

a. What are the steps you have taken to fix the error you identified in the code fragment?

Breakpoints required:

- Breakpoint 1: Set a breakpoint at the start of the outer loop to check if the loop is iterating correctly and if the array is being sorted properly.
- Breakpoint 2: Set a breakpoint in the nested loop to verify that the elements are being compared and swapped as needed.

Steps to fix the errors:

- Fix the outer loop condition: Change `i >= n` to `i < n` so the loop iterates through the array correctly.

- Remove the unnecessary semicolon after the outer `for` loop to prevent it from ending prematurely.
- Fix the condition for swapping: Change the condition `if (a[i] <= a[j])` to `if (a[i] > a[j])` so that elements are swapped when the current element is larger than the next, ensuring ascending order.
- Fix the print loop: Ensure that the print loop prints all elements, including the last one, by changing the condition to `i < n`.

3. Submit your complete executable code.

```java
// Sorting the array in ascending order
import java.util.Scanner;

public class AscendingOrder {
    public static void main(String[] args) {
        int n, temp;
        Scanner s = new Scanner(System.in);

        // Input: number of elements
        System.out.print("Enter no. of elements you want in array: ");
        n = s.nextInt();

        // Create the array
        int a[] = new int[n];
        System.out.println("Enter all the elements:");

        // Input: elements of the array
        for (int i = 0; i < n; i++) {
            a[i] = s.nextInt();
        }

        // Sort the array in ascending order
        for (int i = 0; i < n; i++) { // Fixed: changed from i >= n to i < n and removed the semicolon
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) { // Fixed: changed <= to >
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }

        // Output: sorted array in ascending order
        System.out.print("Ascending Order: ");
        for (int i = 0; i < n - 1; i++) {
            System.out.print(a[i] + ", ");
        }
        System.out.print(a[n - 1]); // Fixed: prints the last element correctly without a trailing comma
    }
}
```

Code 9 : stack implementation

1.How many errors are there in the program? Mention the errors you have identified.

Errors:

- **Error 1:** In the `push` method, the `top--` statement is incorrect. Instead of decrementing, it should increment (`top++`) before assigning the value to `stack[top]`. This ensures that elements are added to the correct position in the stack.
- **Error 2:** In the `display` method, the loop condition `i > top` is incorrect. It should be `i <= top` because we need to display all the elements from `0` to `top`.

2.How many breakpoints you need to fix those errors?

a. What are the steps you have taken to fix the error you identified in the code fragment?

Breakpoints required:

- **Breakpoint 1:** Set a breakpoint in the `push` method to check if the values are being pushed correctly into the stack.
- **Breakpoint 2:** Set a breakpoint in the `display` method to ensure that the stack elements are being displayed correctly.

Steps to fix the errors:

- Fix the `push` method: Change `top--` to `top++` in the `push` method so that elements are added to the stack from the bottom upwards, not downwards.
- Fix the loop condition in the `display` method: Modify the loop condition from `i > top` to `i <= top` to display all elements from the bottom of the stack up to the top.

3. Submit your complete executable code.

```java
// Stack implementation in Java
import java.util.Arrays;

public class StackMethods {
    private int top;
    int size;
    int[] stack;

    // Constructor to initialize the stack
    public StackMethods(int arraySize) {
        size = arraySize;
        stack = new int[size];
        top = -1;
    }

    // Push method to add an element to the stack
    public void push(int value) {
        if (top == size - 1) {
            System.out.println("Stack is full, can't push a value");
        } else {
            top++; // Fixed: Increment top to add elements correctly
            stack[top] = value;
        }
    }

    // Pop method to remove an element from the stack
    public void pop() {
        if (!isEmpty()) {
            top--; // Decrement top to remove the top element
        } else {
            System.out.println("Can't pop...stack is empty");
        }
    }

    // Method to check if the stack is empty
    public boolean isEmpty() {
        return top == -1;
    }

    // Display method to show elements of the stack
    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
        } else {
            for (int i = 0; i <= top; i++) { // Fixed: Changed condition to i <= top
                System.out.print(stack[i] + " ");
            }
            System.out.println();
        }
    }
}

public class StackReviseDemo {
    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);

        // Pushing elements onto the stack
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);

        // Display stack elements
        newStack.display();

        // Popping elements from the stack
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();

        // Display stack after popping
        newStack.display();
    }
}
```

Code 10 : tower of hanoi

1.How many errors are there in the program? Mention the errors you have identified.

Errors:

- Error 1: In the `doTowers` method, the call to `doTowers(topN++, inter--, from+1, to+1)` has logical and syntax issues. The `++` and `--` operators should not be used in recursive calls because they increment/decrement the values, which changes them in ways we do not intend. The correct recursive call should just pass the values as they are (`topN`, `inter`, `from`, and `to`).
- Error 2: The parameters `from+1`, `to+1`, `inter--` are incorrect. The character values ('A', 'B', 'C') are passed as characters representing the rods, and these should not be incremented or decremented. The characters need to be passed as is.

2.How many breakpoints you need to fix those errors?

a. What are the steps you have taken to fix the error you identified in the code fragment?

Breakpoints required:

- Breakpoint 1: Set a breakpoint in the recursive call to `doTowers` to verify the correct values of the parameters passed during the recursion.
- Breakpoint 2: Set a breakpoint before the base case (`if (topN == 1)`) to ensure that the recursive process is terminating correctly.

Steps to fix the errors:

- **Fix the recursive call:**
  Remove the `++` and `--` operators and the character arithmetic from the

recursive call. Simply pass the arguments `topN - 1`, `from`, `inter`, and `to` as they are.

- **Ensure correct base and recursive steps:**
  The recursive calls and base case should remain logically consistent for solving the Tower of Hanoi problem, so no arithmetic should be done on the rods (`from`, `to`, `inter`).

3. Submit your complete executable code.

```java
// Tower of Hanoi implementation in Java
public class MainClass {
    public static void main(String[] args) {
        int nDisks = 3; // Number of disks
        doTowers(nDisks, 'A', 'B', 'C'); // Solve Tower of Hanoi
    }

    // Recursive method to solve Tower of Hanoi
    public static void doTowers(int topN, char from, char inter, char to) {
        if (topN == 1) { // Base case: only one disk
            System.out.println("Disk 1 from " + from + " to " + to);
        } else {
            // Move topN-1 disks from 'from' to 'inter' using 'to' as intermediate
            doTowers(topN - 1, from, to, inter);

            // Move the nth disk from 'from' to 'to'
            System.out.println("Disk " + topN + " from " + from + " to " + to);

            // Move the topN-1 disks from 'inter' to 'to' using 'from' as intermediate
            doTowers(topN - 1, inter, from, to); // Fixed: Removed incorrect increment and decrement operations
        }
    }
}

```