IT-314

Lab - 07

Problem Inspection, Debugging and Static Analysis:

Yash Mehta | 202201309

Table of Contents

1. *Questions……..……..……1*
    a. *Magic Number…..……...1*
    b. *Sorting Array……...3*
    c. *Armstrong.……..…5*
    d. *Merge Sort..……...7*
    e. *GCD and LCM……..10*
    f. *Knapsack……..12*
    g. *Matrix Multiplication……14*
    h. *Stack Implementation……..18*
    i. *Tower of Hanoi……19*
    j. *Quadratic Probing……...20*

Questions

1. Magic Number

a. Problem Inspection

- The program contains two issues:
- Issue 1: In the inner `while` loop, the condition should be `while (sum > 0)` instead of `while (sum == 0)`.
- Issue 2: In the inner `while` loop, there are missing semicolons in the following lines: `s = s * (sum / 10)` and `sum = sum % 10`. These should be corrected to: `s = s * (sum / 10);` and `sum = sum % 10;`.
- The inspection category that best suits this code is Category C: Computation Errors, as it involves mistakes in calculations within the `while` loop.
- Program inspection alone may not catch runtime or logical issues that occur during execution.
- The inspection technique is still valuable for spotting and fixing errors related to calculations and computations.

b. Debugging

- The program has two identified errors, as discussed previously.
- To address these issues, you should set a breakpoint at the start of the inner `while` loop to observe its behavior. Additionally, breakpoints can be used to monitor the values of `num` and `s` during execution.
- Below is the corrected version of the executable code:

```java
import java.util.Scanner;

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

        while (num > 9) {
            sum = num;
            int s = 0;

            while (sum > 0) { // Corrected loop condition
                s = s * (sum / 10); // Added missing semicolon
                sum = sum % 10; // Added missing semicolon
            }

            num = s;
        }

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

2. Sorting Array

a. Problem Inspection

- Identified Issues:
- Issue 1: The class name “Ascending Order” contains an extra space and underscore. It should be renamed to “AscendingOrder.”
- Issue 2: The condition in the first nested for loop `for (int i = 0; i <= n; i++);` is incorrect. It should be corrected to `for (int i = 0; i < n; i++)`.
- Issue 3: There’s an unnecessary semicolon after the first nested for loop, which needs to be removed.
- The most suitable categories for program inspection would be Category A: Syntax Errors and Category B: Semantic Errors, as the code contains both syntax and semantic issues.
- While program inspection can detect syntax and some semantic issues, it may not catch logic errors that impact the program's behavior.
- The program inspection method is useful for resolving syntax and semantic issues, but debugging will be required to fix any logical errors.

b. Debugging

- Identified Issues: The program contains two errors, as mentioned earlier.
- To resolve these issues, you should place breakpoints and step through the code, paying particular attention to the class name, the loop conditions, and the extra semicolon.
- The corrected code is provided below:

```
public class SortArrayAscending {
    public static void main(String[] args) {
        int numElements, tempValue;
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of elements for the array: ");
        numElements = scanner.nextInt();

        int[] array = new int[numElements];
        System.out.println("Please input all the elements:");
        for (int i = 0; i < numElements; i++) {
            array[i] = scanner.nextInt();
        }

        // Sorting the array in ascending order
        for (int i = 0; i < numElements; i++) {
            for (int j = i + 1; j < numElements; j++) {
                if (array[i] > array[j]) {
                    tempValue = array[i];
                    array[i] = array[j];
                    array[j] = tempValue;
                }
            }
        }

        // Displaying the sorted array
        System.out.print("Sorted in Ascending Order: ");
        for (int i = 0; i < numElements - 1; i++) {
            System.out.print(array[i] + ", ");
        }
        System.out.print(array[numElements - 1]);
    }
}
```

3. Armstrong

a. Problem Inspection

- Identified Issue: The program has a single error related to remainder calculation, which has been recognized and corrected.
- The most appropriate program inspection category for this issue is Category C: Computation Errors, since the error pertains to the calculation of the remainder, which falls under computation errors.
- Program inspection is not equipped to detect errors typically identified through debugging, such as breakpoints or runtime logic issues.
- Despite this limitation, program inspection remains useful for uncovering and addressing issues related to code structure and computational errors.

b. Debugging

- Identified Issue: The program contains an error in the calculation of the remainder, which was recognized earlier.
- To correct this issue, you should place a breakpoint at the section where the remainder is calculated. This will allow you to step through the code and monitor the values of variables and expressions during execution to verify correctness.
- The following is the corrected version of the executable code:

```
class ArmstrongCheck {
    public static void main(String[] args) {
        int number = Integer.parseInt(args[0]);
        int originalNumber = number; // To compare at the end
        int result = 0, remainder;

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

        if (result == originalNumber) {
            System.out.println(originalNumber + " is an Armstrong Number");
        } else {
            System.out.println(originalNumber + " is not an Armstrong Number");
        }
    }
}
```

4. Merge Sort

a. Problem Inspection

- The only error in the code was a syntax error due to the incorrect use of the `left` and `right` variables.
- The most suitable category was **Category A: Data Reference Error**, as it dealt with the issue of incorrectly referenced `left` and `right` variables.
- In this case, all the errors present in the code were identifiable using the program inspection method.
- Applying the program inspection technique was **challenging** due to the large size of the code. It was difficult to check every item on the checklist while analyzing the entire code.

b. Debugging

- Applying the program inspection technique was challenging due to the large size of the code. It was difficult to check every item on the checklist while analyzing the entire code.
- With 3 break points, inside the loop of calculating the left half, inside the loop of calculating the right half and inside the loop of calculating the merge array, we can find the mistake if there were any, regarding the execution, one iteration at a time.

```
package DebugMergeSort;
import java.util.*;

public class MergeSort {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        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.
    // post: array is in sorted (nondecreasing) order
    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            // split array into two halves
            int[] left = leftHalf(array);
            int[] right = rightHalf(array);

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

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

```
    // 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;
    }

    // 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;
    }
}
```

5. GCD and LCM

a. Problem Inspection

- Identified Errors:
- Error 1: In the `gcd` function, the condition in the `while` loop should be `while (a % b != 0)` rather than `while (a % b == 0)` to correctly compute the GCD.
- Error 2: There is a logical issue in the `lcm` function. The logic for calculating the LCM is flawed and will cause an infinite loop.
- The most suitable program inspection category for this code is Category C: Computation Errors, since both the `gcd` and `lcm` functions contain computation-related errors.
- Program inspection is unable to detect runtime or logical issues such as infinite loops. These issues fall outside the scope of program inspection.
- Despite this, program inspection is a valuable technique for identifying and correcting computation-related errors.

b. Debugging

- Identified Issues: The program contains two errors as previously stated.
- Steps to Fix:
- To resolve Error 1 in the `gcd` function, place a breakpoint at the start of the `while` loop to ensure it operates correctly.
- To address Error 2 in the `lcm` function, the logic for calculating the LCM needs to be reviewed and corrected, as it involves a logical issue.
- The corrected executable code is provided below:

```java
import java.util.Scanner;

public class GCDandLCM {

    static int findGCD(int num1, int num2) {
        int larger, smaller;
        larger = (num1 > num2) ? num1 : num2; // Larger number
        smaller = (num1 < num2) ? num1 : num2; // Smaller number
        while (smaller != 0) { // Corrected loop condition
            int temp = smaller;
            smaller = larger % temp;
            larger = temp;
        }
        return larger;
    }

    static int findLCM(int num1, int num2) {
        return (num1 * num2) / findGCD(num1, num2); // LCM calculation using GCD
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter two numbers: ");
        int num1 = scanner.nextInt();
        int num2 = scanner.nextInt();

        System.out.println("GCD of the two numbers is: " + findGCD(num1, num2));
        System.out.println("LCM of the two numbers is: " + findLCM(num1, num2));

        scanner.close();
    }
}
```

6. Knapsack

a. Problem Inspection

- Identified Issue: The program contains an error in the line: `int option1 = opt[n++][w];`. The variable `n` is being incremented unintentionally. It should be corrected to: `int option1 = opt[n][w];`.
- The most suitable program inspection category for this code is Category C: Computation Errors, as the error involves incorrect computation within loops.
- Program inspection alone cannot detect runtime or logical issues that may occur during execution.
- Despite this limitation, program inspection is useful for identifying and correcting issues related to computation.

b. Debugging

- Identified Issue: The program contains one error, as mentioned earlier.
- To resolve this issue, set a breakpoint at the line `int option1 = opt[n][w];` to ensure that both `n` and `w` are used correctly without unintended increments.
- The following is the corrected version of the executable code:

```
public class KnapsackProblem {
    public static void main(String[] args) {
        int numItems = Integer.parseInt(args[0]); // Total number of items
        int maxWeight = Integer.parseInt(args[1]); // Maximum capacity of the knapsack
        int[] profits = new int[numItems + 1];
        int[] weights = new int[numItems + 1];

        // Generate random values for profits and weights of items 1 to numItems
        for (int i = 1; i <= numItems; i++) {
            profits[i] = (int) (Math.random() * 1000);
            weights[i] = (int) (Math.random() * maxWeight);
        }

        int[][] optimalSolution = new int[numItems + 1][maxWeight + 1];
        boolean[][] solutionChosen = new boolean[numItems + 1][maxWeight + 1];

        for (int i = 1; i <= numItems; i++) {
            for (int w = 1; w <= maxWeight; w++) {
                int excludeItem = optimalSolution[i - 1][w]; // Corrected no increment here
                int includeItem = Integer.MIN_VALUE;
                if (weights[i] <= w) {
                    includeItem = profits[i] + optimalSolution[i - 1][w - weights[i]];
                }
                optimalSolution[i][w] = Math.max(excludeItem, includeItem);
                solutionChosen[i][w] = (includeItem > excludeItem);
            }
        }

        // Output the result
        System.out.println("Item" + "\t" + "Profit" + "\t" + "Weight" + "\t" + "Selected");
        for (int i = 1; i <= numItems; i++) {
            System.out.println(i + "\t" + profits[i] + "\t" + weights[i] + "\t" + solutionChosen[i][maxWeight]);
        }
    }
```

7. Matrix Multiplication

a. Problem Inspection

- Identified Errors:
- Error 1: In the matrix multiplication nested loops, the indices should begin from `0`, not `-1`.
- Error 2: The error message for incompatible matrix dimensions should be: "Matrices with the entered dimensions can't be multiplied with each other," rather than the incorrect message currently printed.
- The most appropriate category of program inspection for this issue is Category C: Computation Errors, as the problems stem from computational issues.
- Program inspection alone is not able to detect runtime or logical errors that could occur when the program is executed.
- Nonetheless, program inspection remains useful for identifying and correcting computation-related mistakes.

b. Debugging

- Identified Issues: The program contains multiple errors, as previously mentioned.
- To resolve these issues, breakpoints should be set to monitor the values of `c`, `d`, `k`, and `sum` during the program's execution. Special attention should be given to the nested loops that handle the matrix multiplication.
- The following is the corrected version of the executable code:

```
import java.util.Scanner;

public class MatrixMultiplication {

    public static void main(String[] args) {
        int rows1, cols1, rows2, cols2, sum = 0, i, j, k;
        Scanner scanner = new Scanner(System.in);

        // Input dimensions for the first matrix
        System.out.println("Enter the number of rows and columns for the first matrix:");
        rows1 = scanner.nextInt();
        cols1 = scanner.nextInt();

        int[][] matrix1 = new int[rows1][cols1];
        System.out.println("Enter the elements of the first matrix:");
        for (i = 0; i < rows1; i++) {
            for (j = 0; j < cols1; j++) {
                matrix1[i][j] = scanner.nextInt();
            }
        }

        // Input dimensions for the second matrix
        System.out.println("Enter the number of rows and columns for the second matrix:");
        rows2 = scanner.nextInt();
        cols2 = scanner.nextInt();

        // Check matrix multiplication condition
        if (cols1 != rows2) {
            System.out.println("Matrices with the entered dimensions cannot be multiplied.");
        } else {
            int[][] matrix2 = new int[rows2][cols2];
            int[][] productMatrix = new int[rows1][cols2];
```

```
            // Input the elements of the second matrix
            System.out.println("Enter the elements of the second matrix:");
            for (i = 0; i < rows2; i++) {
                for (j = 0; j < cols2; j++) {
                    matrix2[i][j] = scanner.nextInt();
                }
            }

            // Perform matrix multiplication
            for (i = 0; i < rows1; i++) {
                for (j = 0; j < cols2; j++) {
                    for (k = 0; k < cols1; k++) {
                        sum += matrix1[i][k] * matrix2[k][j];
                    }
                    productMatrix[i][j] = sum;
                    sum = 0;
                }
            }

            // Print the resulting matrix
            System.out.println("Product of the entered matrices:");
            for (i = 0; i < rows1; i++) {
                for (j = 0; j < cols2; j++) {
                    System.out.print(productMatrix[i][j] + "\t");
                }
                System.out.println();
            }
        }

        scanner.close();
    }
}
```

8. Stack Implementation

a. Problem Inspection

- Identified Issues:
- Issue 1: In the `push` method, the `top` variable is decremented (`top--`) instead of being incremented. It should be changed to `top++` to correctly push values onto the stack.
- Issue 2: The `display` method has an incorrect loop condition in `for(int i = 0; i < top; i++)`. It should be `for (int i = 0; i <= top; i++)` to display the elements properly.
- Issue 3: The `pop` method is missing from the `StackMethods` class. This method should be added to implement a complete stack functionality.
- The best category of program inspection for this case is Category A: Syntax Errors, as there are multiple syntax issues present. Additionally, Category B: Semantic Errors can help identify issues with the logic and functionality of the code.
- Program inspection is useful for finding and fixing syntax errors, but further inspection is necessary to ensure that the logic and functionality are implemented correctly.

b. Debugging

- Identified Issues: The program contains three errors, as mentioned previously.
- To resolve these issues, you should set breakpoints and step through the code, paying particular attention to the `push`, `pop`, and `display` methods. The `push` and `display` methods need to be corrected, and the missing `pop` method should be added to complete the stack implementation.
- The following is the corrected version of the executable code:

```
public class StackOperations {
    private int top;
    private int capacity;
    private int[] stackArray;

    public StackOperations(int arrayCapacity) {
        capacity = arrayCapacity;
        stackArray = new int[capacity];
        top = -1;
    }

    public void push(int value) {
        if (top == capacity - 1) {
            System.out.println("Stack overflow, cannot push the value");
        } else {
            top++;
            stackArray[top] = value;
        }
    }

    public void pop() {
        if (!isStackEmpty()) {
            top--;
        } else {
            System.out.println("Cannot pop...stack is empty");
        }
    }

    public boolean isStackEmpty() {
        return top == -1;
    }
```

```
    public void displayStack() {
        for (int i = 0; i <= top; i++) {
            System.out.print(stackArray[i] + " ");
        }
        System.out.println();
    }
}
```

9. Tower of Hanoi

a. Problem Inspection

- Identified Issues:
- Issue 1: In the line `doTowers(topN++, inter--, from+1, to+1)`, there are incorrect usage of increment and decrement operators. It should be corrected to `doTowers(topN - 1, inter, from, to)` to fix the logic.
- The most suitable category for program inspection in this case is Category B: Semantic Errors, as the errors relate to the logic and function of the code.
- Applying program inspection is beneficial for identifying and correcting semantic issues in the code.

b. Debugging

- Identified Issue: The program contains a single error, as previously mentioned.
- To resolve this issue, you need to replace the following line: `doTowers(topN++, inter--, from+1, to+1);`
- with the corrected version:

```
public class TowerOfHanoi {
    public static void main(String[] args) {
        int totalDisks = 3;
        solveTowers(totalDisks, 'A', 'B', 'C');
    }

    public static void solveTowers(int numDisks, char source, char auxiliary, char destination) {
        if (numDisks == 1) {
            System.out.println("Move disk 1 from " + source + " to " + destination);
        } else {
            solveTowers(numDisks - 1, source, destination, auxiliary);
            System.out.println("Move disk " + numDisks + " from " + source + " to " + destination);
            solveTowers(numDisks - 1, auxiliary, source, destination);
        }
    }
}
```

10. Quadratic Probing

a. Problem Inspection

- Identified Issues:
- Error 1: There is a typo in the `insert` method on the line `i += (i + h / h--)`, which needs to be corrected.
- Error 2: In the `remove` method, there is a logic error in the loop for rehashing keys. The line should be changed to `i = (i + h * h++)`.
- Error 3: A similar logic error exists in the `get` method, where the line for finding the key should also be corrected to `i = (i + h * h++)`.
- The most suitable program inspection categories for this code are Category A: Syntax Errors and Category B: Semantic Errors, as there are both syntax mistakes and logical issues.
- Program inspection is useful for identifying and correcting these types of errors, but it may not always detect more subtle logical issues that impact the overall behavior of the program.

b. Debugging

- Identified Issues: The program contains three errors, as previously discussed.
- To address these issues, you should set breakpoints and step through the code, closely monitoring variables such as `i`, `h`, `tmp1`, and `tmp2`. Special attention should be given to the logic within the `insert`, `remove`, and `get` methods.
- The following is the corrected version of the executable code:

```
import java.util.Scanner;

class QuadraticProbingHashTable {
    private int currentSize, maxCapacity;
    private String[] keys;
    private String[] values;

    public QuadraticProbingHashTable(int capacity) {
        currentSize = 0;
        maxCapacity = capacity;
        keys = new String[maxCapacity];
        values = new String[maxCapacity];
    }

    public void clearTable() {
        currentSize = 0;
        keys = new String[maxCapacity];
        values = new String[maxCapacity];
    }

    public int getSize() {
        return currentSize;
    }

    public boolean isFull() {
        return currentSize == maxCapacity;
    }

    public boolean isEmpty() {
        return currentSize == 0;
    }
```

```
public boolean containsKey(String key) {
    return getValue(key) != null;
}

private int getHash(String key) {
    return key.hashCode() % maxCapacity;
}

public void insert(String key, String value) {
    int index = getHash(key);
    int i = index, h = 1;
    do {
        if (keys[i] == null) {
            keys[i] = key;
            values[i] = value;
            currentSize++;
            return;
        }
        if (keys[i].equals(key)) {
            values[i] = value;
            return;
        }
        i = (i + h * h++) % maxCapacity;
    } while (i != index);
}
```

```
    public String getValue(String key) {
        int i = getHash(key), h = 1;
        while (keys[i] != null) {
            if (keys[i].equals(key))
                return values[i];
            i = (i + h * h++) % maxCapacity;
        }
        return null;
    }

    public void remove(String key) {
        if (!containsKey(key))
            return;
        int i = getHash(key), h = 1;
        while (!key.equals(keys[i]))
            i = (i + h * h++) % maxCapacity;
        keys[i] = values[i] = null;

        for (i = (i + h * h++) % maxCapacity; keys[i] != null; i = (i + h * h++) % maxCapacity) {
            String tempKey = keys[i], tempValue = values[i];
            keys[i] = values[i] = null;
            currentSize--;
            insert(tempKey, tempValue);
        }
        currentSize--;
    }
```

```
    public void printTable() {
        System.out.println("\nHash Table Contents: ");
        for (int i = 0; i < maxCapacity; i++)
            if (keys[i] != null)
                System.out.println(keys[i] + " " + values[i]);
        System.out.println();
    }
}

public class QuadraticProbingHashTableDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Hash Table Demonstration\n");

        System.out.println("Enter size of the hash table:");
        QuadraticProbingHashTable hashTable = new QuadraticProbingHashTable(scanner.nextInt());

        char choice;
        do {
            System.out.println("\nHash Table Operations:");
            System.out.println("1. Insert");
            System.out.println("2. Remove");
            System.out.println("3. Get Value");
            System.out.println("4. Clear Table");
            System.out.println("5. Get Size");
            int operation = scanner.nextInt();
```

```
            switch (operation) {
                case 1:
                    System.out.println("Enter key and value:");
                    hashTable.insert(scanner.next(), scanner.next());
                    break;
                case 2:
                    System.out.println("Enter key to remove:");
                    hashTable.remove(scanner.next());
                    break;
                case 3:
                    System.out.println("Enter key to retrieve value:");
                    System.out.println("Value = " + hashTable.getValue(scanner.next()));
                    break;
                case 4:
                    hashTable.clearTable();
                    System.out.println("Hash Table Cleared.");
                    break;
                case 5:
                    System.out.println("Current size = " + hashTable.getSize());
                    break;
                default:
                    System.out.println("Invalid option.");
                    break;
            }

            hashTable.printTable();
            System.out.println("\nWould you like to continue? (Type y or n)");
            choice = scanner.next().charAt(0);
        } while (choice == 'y' || choice == 'Y');
    }
}
```