IT314 Lab 9 Report

Keyur Govrani
202101498

Contents

1 Armstrong

A. Program Inspection

1. There is one error in the program, related to the computation of the remainder, and it has been identified and corrected.

2. The most effective category of program inspection for this code is Category C: Computation Errors, as the error pertains to the computation of the remainder, a type of computation error.

3. Program inspection does not identify debugging-related errors. It does not detect issues such as breakpoints or runtime errors like logic errors.

4. The program inspection technique is valuable for identifying and rectifying issues related to code structure and computation errors.

B. Debugging

1. There is one error in the program related to the computation of the remainder, as previously identified.

2. To fix this error, one should set a breakpoint at the point where the remainder is computed to ensure it's calculated correctly. Step through the code to observe the values of variables and expressions during execution.

3. The corrected executable code is as follows:

```
// Armstrong Number
class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num; // used to check at the last time
        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");
    }
}
```

2 GCD and LCM

A. Program Inspection

1. There are two errors in the program:

2. Error 1: In the gcd function, the while loop condition should be `while(a % b != 0)` instead of `while(a % b == 0)` to calculate the GCD correctly.

3. Error 2: In the lcm function, there is a logic error. The logic used to calculate LCM is incorrect and will result in an infinite loop.

4. For this code, the most effective category of program inspection is Category C: Computation Errors, as it contains computation errors in both the gcd and lcm functions.

5. Program inspection is not able to identify runtime issues or logical errors. It can't identify errors like infinite loops.

6. The program inspection technique is worth applying to identify and fix computation-related issues.

B. Debugging

1. There are two errors in the program as mentioned above.

2. To fix these errors:

3. For Error 1 in the gcd function, you need one breakpoint at the beginning of the while loop to verify the correct execution of the loop.

4. For Error 2 in the lcm function, you would need to review the logic for calculating LCM, as it's a logical error.

5. The corrected executable code is as follows:

```
import java.util.Scanner;

public class GCD_LCM {
    static int gcd(int x, int y) {
        int a, b;
        a = (x > y) ? x : y; // a is greater number
        b = (x < y) ? x : y; // b is smaller number
        while (b != 0) { // Fixed the while loop condition
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }

    static int lcm(int x, int y) {
        return (x * y) / gcd(x, y); // Calculate LCM using GCD
    }

    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();
    }
}
```

3 Knapsack

A. Program Inspection

1. There is one error in the program. It is in the following line: `int option1 = opt[n++][w];` The variable `n` is incremented, which is not intended. It should be: `int option1 = opt[n][w];`

2. The category of program inspection that would be most effective for this code is Category C: Computation Errors, as the identified error is related to computation within loops.

3. Program inspection is not able to identify runtime errors or logical errors that might arise during program execution.

4. The program inspection technique is worth applying to identify and fix computation-related issues.

B. Debugging

1. There is one error in the program, as identified above.

2. To fix this error, you would need one breakpoint at the line: `int option1 = opt[n][w];` to ensure `n` and `w` are correctly used without unintended increments.

3. The corrected executable code is as follows:

```
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]; // Fixed the increment here
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w)
                    option2 = profit[n] + opt[n - 1][w - weight[n]];

                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
        }
```

```
        // Rest of the code is fine

        // Print results
        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]);
        }
    }
}
```

4 Magic Number

A. Program Inspection

1. There are two errors in the program:

2. Error 1: In the inner while loop, the condition should be `while (sum > 0)` instead of `while (sum == 0)`.

3. Error 2: Inside the inner while loop, there are missing semicolons in the lines: `s=s*(sum/10);`
`sum=sum%10`
They should be corrected as: `s = s * (sum / 10);`
`sum = sum % 10;`

4. The category of program inspection that would be most effective for this code is Category C: Computation Errors, as it contains computation errors in the while loop.

5. Program inspection is not able to identify runtime issues or logical errors that might arise during program execution.

6. The program inspection technique is worth applying to identify and fix computation-related issues.

B. Debugging

1. There are two errors in the program, as identified above.

2. To fix these errors, you would need one breakpoint at the beginning of the inner while loop to verify the execution of the loop. You can also use breakpoints to check the values of `num` and `s` during execution.

3. The corrected executable code is as follows:

```
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) { // Fixed the condition here
                s = s * (sum / 10);
                sum = sum % 10; // Fixed the 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.");
        }
    }
}
```

5 Merge Sort

A. Program Inspection

1. There are several errors in the program:

2. Error 1: In the `mergeSort` method, the lines `int[] left = leftHalf(array+1);` and `int[] right = rightHalf(array-1);` should be corrected. It seems like an attempt to split the array, but it's not done correctly.

3. Error 2: The `leftHalf` and `rightHalf` methods are incorrect. They should return the correct halves of the array.

4. Error 3: The `merge` method should have `left` and `right` arrays as inputs, not `left++` and `right--`.

5. The category of program inspection that would be most effective for this code is Category C: Computation Errors, as there are computation-related issues in the code.

6. Program inspection cannot identify runtime issues or logical errors that might arise during program execution.

7. The program inspection technique is worth applying to identify and fix computation-related issues.

B. Debugging

1. There are multiple errors in the program, as identified above.

2. To fix these errors, you would need to set breakpoints to examine the values of `left`, `right`, and `array` during execution. You can also use breakpoints to check the values of `i1` and `i2` inside the `merge` method.

3. The corrected executable code is as follows:

```
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));
    }

    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            int[] left = leftHalf(array);
            int[] right = rightHalf(array);
            mergeSort(left);
            mergeSort(right);
            merge(array, left, right);
        }
    }

    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;
    }

    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;
    }

    public static void merge(int[] result, int[] left, int[] right) {
        int i1 = 0;
        int i2 = 0;
        for (int i = 0; i < result.length; i++) {
            if (i2 >= right.length || (i1 < left.length && left[i1] <= right[i2])) {
                result[i] = left[i1];
                i1++;
            } else {
                result[i] = right[i2];
                i2++;
            }
        }
    }
}
```

6 Multiply Matrices

A. Program Inspection

1. There are several errors in the program:

2. Error 1: In the nested loops for matrix multiplication, the loop indices should start from 0, not -1.

3. Error 2: The error message when the matrix dimensions are incompatible should print "Matrices with entered orders can't be multiplied with each other," not "Matrices with entered orders can't be multiplied with each other."

4. The category of program inspection that would be most effective for this code is Category C: Computation Errors, as there are computation-related issues in the code.

5. Program inspection cannot identify runtime issues or logical errors that might arise during program execution.

6. The program inspection technique is worth applying to identify and fix computation-related issues.

B. Debugging

1. There are multiple errors in the program, as identified above.

2. To fix these errors, you would need to set breakpoints to examine the values of `c`, `d`, `k`, and `sum` during execution. You should pay particular attention to the nested loops where the matrix multiplication occurs.

3. The corrected executable code is as follows:

```
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 the first matrix");
        m = in.nextInt();
        n = in.nextInt();

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

        System.out.println("Enter the elements of the 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 the 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 the 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 < p; k++) {
                        sum = sum + first[c][k] * second[k][d];
                    }

                    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");
            }
        }
    }
}
```

7 Quadratic Probing

A. Program Inspection

1. There are multiple errors in the program:

2. Error 1: The insert method has a typo in the line i + = (i + h / h–)

3. Error 2: In the remove method, there is a logic error in the loop to rehash keys. It should be i = (i + h * h++)

4. Error 3: In the get method, there is a logic error in the loop to find the key. It should be i = (i + h * h++)

5. The category of program inspection that would be most effective for this code is Category A: Syntax Errors and Category B: Semantic Errors, as there are both syntax errors and semantic issues in the code.

6. The program inspection technique is worth applying to identify and fix these errors, but it may not identify logical errors that affect the program's behavior.

B. Debugging

1. There are three errors in the program, as identified above.

2. To fix these errors, you would need to set breakpoints and step through the code while examining variables like `i`, `h`, `tmp1`, and `tmp2`. You should pay attention to the logic of the insert, remove, and get methods.

3. The corrected executable code is as follows:

```
import java.util.Scanner;

class QuadraticProbingHashTable {
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

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

    public void makeEmpty() {
        currentSize = 0;
        keys = new String[maxSize];
        vals = a String[maxSize];
    }

    public int getSize() {
        return currentSize;
    }

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

    public boolean isEmpty() {
        return getSize() == 0;
    }

    public boolean contains(String key) {
        return get(key) != null;
    }

    private int hash(String key) {
        return key.hashCode() % maxSize;
    }
```

```
public void insert(String key, String val) {
    int tmp = hash(key);
    int i = tmp, h = 1;
    do {
        if (keys[i] == null) {
            keys[i] = key;
            vals[i] = val;
            currentSize++;
            return;
        }
        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        }
        i += (h * h++) % maxSize;
    } while (i != tmp);
}

public String get(String key) {
    int i = hash(key), h = 1;
    while (keys[i] != null) {
        if (keys[i].equals(key))
            return vals[i];
        i = (i + h * h++) % maxSize;
    }
    return null;
}

public void remove(String key) {
    if (!contains(key))
        return;

    int i = hash(key), h = 1;
    while (!key.equals(keys[i]))
        i = (i + h * h++) % maxSize;

    keys[i] = vals[i] = null;

    for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) % maxSize)
    {
        String tmp1 = keys[i], tmp2 = vals[i];
        keys[i] = vals[i] = null;
        currentSize--;
        insert(tmp1, tmp2);
    }
    currentSize--;
}

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

```
                System.out.println(keys[i] + " " + vals[i]);
            System.out.println();
        }
    }

    public class QuadraticProbingHashTableTest {
        public static void main(String[] args) {
            Scanner scan = new Scanner(System.in);
            System.out.println("Hash Table Test\n\n");
            System.println("Enter size");

            QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt());

            char ch;

            do {
                System.out.println("\nHash Table Operations\n");
                System.out.println("1. insert");
                System.out.println("2. remove");
                System.out.println("3. get");
                System.out.println("4. clear");
                System.out.println("5. size");

                int choice = scan.nextInt();

                switch (choice) {
                    case 1:
                        System.out.println("Enter key and value");
                        qpht.insert(scan.next(), scan.next());
                        break;
                    case 2:
                        System.out.println("Enter key");
                        qpht.remove(scan.next());
                        break;
                    case 3:
                        System.out.println("Enter key");
                        System.out.println("Value = " + qpht.get(scan.next()));
                        break;
                    case 4:
                        qpht.makeEmpty();
                        System.out.println("Hash Table Cleared\n");
                        break;
                    case 5:
                        System.out.println("Size = " + qpht.getSize());
                        break;
                    default:
                        System.out.println("Wrong Entry\n");
                        break;
                }
                qpht.printHashTable();
                System.out.println("\nDo you want to continue (Type y or n) \n");
                ch = scan.next().charAt(0);
```

```
        } while (ch == 'Y' || ch == 'y');
    }
}
```

8 Sorting Array

A. Program Inspection

1. Errors identified:

2. Error 1: The class name "Ascending _Order" contains an extra space and an underscore. The class name should be corrected to "AscendingOrder."

3. Error 2: The first nested for loop has an incorrect loop condition for (int i = 0; i ¿= n; i++);, which should be modified to for (int i = 0; i ¡ n; i++).

4. Error 3: There is an extra semicolon (;) after the first nested for loop, which should be removed.

5. The most effective category of program inspection would be Category A: Syntax Errors and Category B: Semantic Errors, as there are both syntax errors and semantic issues in the code.

6. Program inspection alone can identify and fix syntax errors and some semantic issues. However, it may not detect logic errors that affect the program's behavior.

7. The program inspection technique is worth applying to fix the syntax and semantic errors, but debugging is required to address logic errors.

B. Debugging

1. There are two errors in the program as identified above.

2. To fix these errors, you need to set breakpoints and step through the code. You should focus on the class name, the loop conditions, and the unnecessary semicolon.

3. The corrected executable code is as follows:

```
import java.util.Scanner;

public class AscendingOrder {
    public static void main(String[] args) {
        int n, temp;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the number of elements you want in the array: ");
        n = s.nextInt();
        int a[] = new int[n];
        System.out.println("Enter all the elements:");
        for (int i = 0; i < n; i++) {
            a[i] = s.nextInt();
        }
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) {
                    temp = a[i];
                    a[i] = a[j];
```

```
                    a[j] = temp;
                }
            }
        }
        System.out.print("Ascending Order: ");
        for (int i = 0; i < n - 1; i++) {
            System.out.print(a[i] + ", ");
        }
        System.out.print(a[n - 1]);
    }
}
```

9 Stack Implementation

A. Program Inspection

1. Errors identified:

2. Error 1: The push method has a decrement operation on the top variable (top–) instead of an increment operation. It should be corrected to top++ to push values correctly.

3. Error 2: The display method has an incorrect loop condition in for(int i=0; i ¿ top; i++). The loop condition should be for (int i = 0; i ¡= top; i++) to correctly display the elements.

4. Error 3: The pop method is missing in the StackMethods class. It should be added to provide a complete stack implementation.

5. The most effective category of program inspection would be Category A: Syntax Errors, as there are syntax errors in the code. In addition, Category B: Semantic Errors can help identify logic and functionality issues.

6. The program inspection technique is worth applying to identify and fix syntax errors, but additional inspection is needed to ensure the logic and functionality are correct.

B. Debugging

1. There are three errors in the program, as identified above.

2. To fix these errors, you would need to set breakpoints and step through the code, focusing on the push, pop, and display methods. Correct the push and display methods and add the missing pop method to provide a complete stack implementation.

3. The corrected executable code is as follows:

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

    public StackMethods(int arraySize) {
        size = arraySize;
        stack = new int[size];
        top = -1;
    }
```

```
    public void push(int value) {
        if (top == size - 1) {
            System.out.println("Stack is full, can't push a value");
        } else {
            top++;
            stack[top] = value;
        }
    }

    public void pop() {
        if (!isEmpty()) {
            top--;
        } else {
            System.out.println("Can't pop...stack is empty");
        }
    }

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

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

10 Tower of Hanoi

A. Program Inspection

1. Errors identified:

2. Error 1: In the line doTowers(topN ++, inter–, from+1, to+1), there are errors in the increment and decrement operators. It should be corrected to doTowers(topN - 1, inter, from, to).

3. The most effective category of program inspection would be Category B: Semantic Errors because the errors in the code are related to logic and function.

4. The program inspection technique is worth applying to identify and fix semantic errors in the code.

B. Debugging

1. There is one error in the program, as identified above.

2. To fix this error, you need to replace the line:

```
doTowers(topN ++, inter--, from+1, to+1);
```

3. with the correct version:

```
doTowers(topN - 1, inter, from, to);
```

4. The corrected executable code is as follows:

```
public class MainClass {
   public static void main(String[] args) {
      int nDisks = 3;
      doTowers(nDisks, 'A', 'B', 'C');
   }

   public static void doTowers(int topN, char from, char inter, char to) {
      if (topN == 1) {
         System.out.println("Disk 1 from " + from + " to " + to);
      } else {
         doTowers(topN - 1, from, to, inter);
         System.out.println("Disk " + topN + " from " + from + " to " + to);
         doTowers(topN - 1, inter, from, to);
      }
   }
}
```