IT314

Software Engineering

LAB – 7

Name: Smit Fefar

Id: 202201253

Code Debugging and Program Inspection

Code-1: Armstrong

A . Program Inspection:

1. There is one error related to the computation of the remainder.

2. 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 runtime errors.

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

B. Debugging

1. Error in the program related to the computation of the remainder.

2. To fix this error, one should set a breakpoint at the point where the remainder is computed to ensure its calculated correctly.

3. The corrected executable code:

```
// 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 and LCM

A. Program Inspection:

1. There are two errors in the program:

    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.

    Error 2: The logic used to calculate LCM is incorrect and will result in an infinite loop.

2. Category C: Computation Errors.

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

4. 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: we nned to keep breakpoint at the start of while loop in gcd function and should check the logic of the lcm function.

3. The corrected executable code:

```
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) {
                int temp = b;
                b = a % b;
```

```
            a = temp;
            }
        return a;
}
static int lcm(int x, int y) {
        return (x * y) / gcd(x, y);
}
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

A. Program Inspection:

1. There is one error in the program. It is in the following line: int option1 = opt[n++][w]; incrementing n is not intended. It should be: int option1 = opt[n][w];

2. Category C: Computation Errors.

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

4. 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:

```
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];
    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);
    }
}
```

```
        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

A. Program Inspection

1. There are two errors in the program:

2. Error-1: condition should be while (sum > 0). And 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;

3. Category C: Computation Errors.

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

B. Debugging

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

2. To fix these errors, we should 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:

```
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.");
        }
    }
}
```

Code-5: Merge Sort

A. Program Inspection:

1. There are several errors in the program:

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.

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

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

2. Category C: Computation Errors, as there are computation-related issues in the code.

3. Program inspection cannot identify runtime issues 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 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:

```
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++;
                }
        }
      }
}
```

Code6: Multiply Matrices

A. Program Inspection:

1. There are a few mistakes in the program that need fixing.

Mistake 1: In the matrix multiplication, the loop counters should start from 0 instead of -1.

Mistake 2: When the matrix sizes don’ t match for multiplication, the error message should say, "Matrices with the given sizes cannot be multiplied," but it currently prints the same message twice.

2. This problem falls under Computation Errors because the issues relate to how the program handles calculations.

3. Program reviews like this can’ t catch runtime errors or logical problems that might happen when the program runs.

B. Debugging:

1. As mentioned earlier, there are several errors in the program.

2. To resolve them, you should set breakpoints to check the values of c, d, k, and sum while the program runs. Pay close attention to the nested loops where the matrix multiplication happens, as that’ s where the main issues are.

3. The corrected executable code:

```
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");
                }
            }
        }
}
```

Code-7: Quadratic Probing

A. Program Inspection

1. There are multiple errors in the program:

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

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

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++)

2. Category A: Syntax Errors and Category B: Semantic Errors.

3. 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 behaviour.

B. Debugging

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

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

3. The corrected executable code:

```
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');
      }
```

}

Code-8: Sorting Array

A. Program Inspection:

1. Errors identified:

Error 1: The class name " Ascending Order" contains an extra space and an underscore.

Error 2: The first nested for loop has an incorrect loop condition.

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

2. Category A: Syntax Errors and Category B: Semantic Errors.

3. 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.

4. 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 should check the names of variables and class of code.

3. The corrected executable code:

```
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]);
        }
}
```

Code-9: Stack Implementation

A. Program Inspection

1. Errors identified:

Error 1: The push method has a decrement operation on the top variable (top–) instead of an increment operation.

Error 2: The display method has an incorrect loop condition in for.

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

2. Category A and B.

3. 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:

```
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();
    }
}
```

Code-10 Tower of Hanoi

A. Program Inspection:

1. Errors identified:

    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).

2. Category B.

3. 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); with the correct version: doTowers(topN - 1, inter, from, to);

3. The corrected executable code:

```
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);
            }
        }
}
```

Program Inspection/Debugging For Long-code from GitHub

We need to identify errors from the following error checklist.

1. Data referencing Errors
2. Data declaration Errors
3. Computation Errors
4. Comparison Errors
5. Control Flow errors
6. Interface errors
7. Input/Output Errors
8. Other Checks

Github Link:

https://github.com/expressjs/express/blob/master/test/app.router.js

Following are the observations:

For first some part of code:

1. Data Referencing Errors

You might run into situations where you're trying to access a variable that doesn’ t exist yet, which can throw a ReferenceError. For instance, if you're trying to get req.params.id, it could be undefined if the route you set up doesn't match the incoming request.

2. Data Declaration Errors

When using middleware, if you expect certain request parameters and they aren’ t there, things can get messy. If you try to access req.params without properly defining your route, you might end up with unexpected results.

3. Computation Errors

These errors pop up when you're trying to perform operations on the wrong types of data. For example, if you accidentally use a string or an undefined value in a mathematical operation within your middleware, it could lead to a TypeError. It's crucial to make sure the data types you’ re working with are compatible.

4. Comparison Errors

Sometimes, your conditions may not be formed correctly, leading to logic errors. In your code, if you don’ t handle cases where req.method isn’ t what you expect, you might accidentally skip over important logic, causing your app to behave unexpectedly.

5. Control Flow Errors

Control flow errors usually happen when you forget to call next() in your middleware. This can leave your requests hanging. If you have a middleware function in app.use and you don’ t call next() after processing, your application might just stop responding.

6. Interface Errors

If your route is set up to expect certain parameters and they don’ t come through, you can run into all sorts of unexpected behavior. For example, if you set up a route like app.get('/:name', ...) but don’ t access it correctly, you might end up with missing or incorrect req.params, which can lead to confusion.

7. Input/Output Errors

These errors occur when you don’ t handle responses properly, which can lead to hanging requests or incorrect status codes. If you forget to send a response in one of your handler functions, clients will be left waiting indefinitely.

8. Other Checks

Lastly, if you don’ t validate incoming request data correctly or mismanage assertions, it can lead to runtime errors. Using assert.throws improperly can result in unhandled rejections, making it hard to know what went wrong.

For the remaining part of code:

1. Data Referencing Errors

Undefined Variables: In cases where route parameters are not provided, such as req.params.user or req.params.format, they might be undefined. For example, if you call request(app).get('/user/'), it will result in req.params.user being undefined, leading to

unexpected results when you try to use it in the response.

2. Data Declaration Errors

Missing Route Parameters: If a request doesn't match a route that expects parameters, it can lead to problems. For instance, if you set up a route like app.get('/user/:user'), and you call request(app).get('/user/tj/edit'), the request will not match, resulting in a 404 Not Found error.

3. Computation Errors

TypeErrors: If you expect certain data types but get something else, you can run into issues. For instance, if req.params is accessed without checking if the

parameters exist or are of the expected type, trying to perform string operations or concatenations might cause a TypeError.

4. Comparison Errors

Logic Errors in Conditionals: If you have conditionals that incorrectly check values, it can lead to wrong paths being taken in your logic. For example, if you check if (req.method === 'GET') without properly validating, it might bypass certain logic, leading to unintended responses or error codes.

5. Control Flow Errors

Next Function Misuse: Forgetting to call next() in your middleware can lead to hanging requests. For example,

in the first app.get('/user/', ...) route, if you were to forget to call next() after processing, the request could hang indefinitely.

6. Interface Errors

Mismatch in Expected Parameters: If you set up a route expecting parameters but don't provide them correctly, it can lead to unexpected behavior. For example, using app.get('/user/:user') and then requesting request(app).get('/user/') will not provide a user, leading to issues with the response handling.

7. Input/Output Errors

Response Handling Errors: If you fail to send a response in some conditions, clients can end up waiting

indefinitely. For instance, if you have a route that ends without sending a response or calling res.end() or res.send(), it will result in clients not receiving the expected output.

8. Other Checks

Assertions and Validations: If you’ re not correctly validating incoming request data, it could lead to runtime errors. For example, if you're checking for specific request parameters but forget to validate their existence or format, it may lead to errors down the line.