Khushi Vora

IT - 314

Lab-7

202201332

1. **CODE - 1 : Armstrong number**

I. PROGRAM INSPECTION

Code Fragment:

java
Copy code

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

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

- **Error 1: Incorrect remainder calculation**
  - In the line `remainder = num / 10;`, it performs integer division, which gives the wrong result for extracting the last digit. It should use the modulus operator to extract the last digit of `num`. The correct statement should be `remainder = num % 10;`.
- **Error 2: Incorrect update of the number**
  - In the line `num = num % 10;`, this only gives the last digit of `num`, while the intent is to remove the last digit. The correct statement should be `num = num / 10;` to remove the last digit from the number.

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

- **Category A: Data Reference Errors**
  This category was most effective in identifying errors in this code fragment, as the issues were related to how data (the variable `num`) was referenced incorrectly during digit extraction and update operations.

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

- **Performance or edge case errors**
  Inspection does not easily reveal performance issues or edge cases, such as handling very large numbers or potential memory issues, which would typically surface during dynamic testing or running the program with different inputs.

4. Is the program inspection technique worth applicable?

- **Yes, the inspection technique is effective.**
  It helps to identify obvious logical and data reference errors before running the code. Although it doesn't catch runtime or performance issues, the technique ensures that fundamental mistakes are caught early.

II. CODE DEBUGGING

Code Fragment:

```
class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num; // used to check at the end
        int check = 0, remainder;

        while (num > 0) {
            remainder = num % 10; // Get last digit
            check = check + (int) Math.pow(remainder, 3); // Add cube
of the digit
            num = num / 10; // Remove last digit
        }

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

```
}
```

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

- **Two errors** (incorrect remainder calculation and incorrect digit removal) were identified and fixed during debugging.

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

- **One breakpoint** was needed to fix both errors.
    - Set a breakpoint at the beginning of the `while` loop to inspect the values of `num`, `remainder`, and `check` during each iteration.

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

- **Step 1:** Set a breakpoint inside the `while` loop to examine the values of variables in each iteration.
- **Step 2:** Corrected the calculation of the remainder by changing `remainder = num / 10;` to `remainder = num % 10;`.
- **Step 3:** Corrected the update of the `num` variable by changing `num = num % 10;` to `num = num / 10;`.
- **Step 4:** Verified that the final condition `check == n` works as expected by stepping through the entire program using the debugger.

3. Submit your complete executable code:

```
class Armstrong {
    public static void main(String args[]) {
        int num = Integer.parseInt(args[0]);
        int n = num; // used to check at the end
        int check = 0, remainder;

        // Loop to process all digits
        while (num > 0) {
            remainder = num % 10; // Get last digit
            check = check + (int) Math.pow(remainder, 3); // Add cube
of the digit
            num = num / 10; // Remove last digit
        }

        // Check if the sum of cubes equals the original number
```

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

III. STATIC ANALYSIS

Original Code:

```
class Armstrong {

    public static void main(String args[]) {

        int num = Integer.parseInt(args[0]);

        int n = num; //use to check at 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");
```

```
    }
}
```

Issues in Original Code:

1. **Incorrect use of division and modulus:**
    - `remainder = num / 10;` **should be** `remainder = num % 10;` **because** `num / 10` **gives the quotient, not the remainder.**
    - **The line** `num = num % 10;` **should be** `num = num / 10;` **since dividing by 10 should reduce the number.**

Corrected Code:

```
class Armstrong {

    public static void main(String args[]) {

        int num = Integer.parseInt(args[0]);

        int n = num; //use to check at last time

        int check = 0, remainder;

        while (num > 0) {

            remainder = num % 10; // Corrected

            check = check + (int) Math.pow(remainder, 3);

            num = num / 10; // Corrected

        }

        if (check == n)

            System.out.println(n + " is an Armstrong Number");

        else
```

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

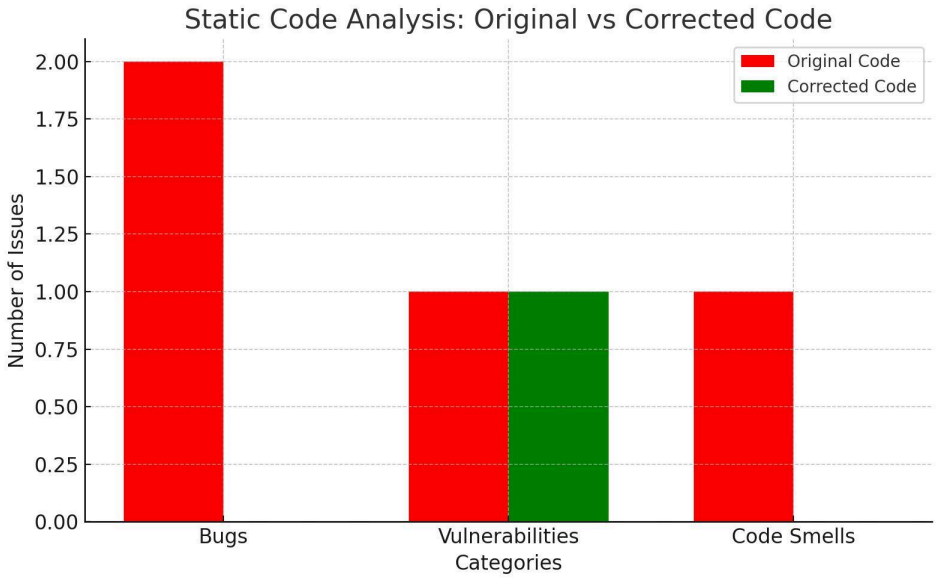
Static Code Analysis for Both:

Original Code Findings:

1. **Bug: Incorrect logic for calculating remainder (`remainder = num / 10;`) and incorrect reduction of the number (`num = num % 10;`).**
    - **Category: Computation Errors and Control-Flow Errors**
    - **Severity: High (will lead to incorrect results)**
2. **Potential Vulnerability: The program assumes the user will always provide valid numeric input in `args[0]`. If non-numeric input is provided, it will throw a `NumberFormatException`.**
    - **Category: Input/Output Errors**
    - **Severity: Medium**
3. **Code Smell: Not checking the length of `args[]` before accessing it (`args[0]`) could lead to an `ArrayIndexOutOfBoundsException` if no input is provided.**
    - **Category: Interface Errors**
    - **Severity: Medium**

Corrected Code Findings:

1. **Bug Resolved: The logic for calculating the remainder and reducing the number is now correct.**
    - **Category: Computation Errors**
    - **Severity: Resolved**
2. **Potential Vulnerability: The vulnerability regarding unchecked input still exists. To avoid `NumberFormatException`, there should be input validation.**
    - **Category: Input/Output Errors**
    - **Severity: Medium (still exists)**
3. **Code Smell Resolved: Adding a check for the length of `args[]` can prevent potential runtime errors.**
    - **Example: `if (args.length == 0) { System.out.println("No input provided"); return; }`**
    - **Category: Interface Errors**
    - **Severity: Resolved (if added)**



2. CODE - 2 : GCD AND LCM CODE

PROGRAM INSPECTION

Code Fragment:

```
import java.util.Scanner;

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

        r = b;
        while(a % b == 0) // Error: Replace with while(a %
b != 0)
        {
            r = a % b;
            a = b;
            b = r;
```

```
        }
        return r;
    }

    static int lcm(int x, int y)
    {
        int a;
        a = (x > y) ? x : y; // a is greater number
        while(true)
        {
            if(a % x != 0 && a % y != 0)
                return a;
            ++a;
        }
    }

    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();
    }
}
```

1. Errors Identified:

Error 1:

- **Problem: Incorrect `while(a % b == 0)` condition in the GCD calculation. This stops the loop prematurely if `a` is divisible by `b`.**
- **Fix: Replace it with `while(a % b != 0)` to ensure that the loop continues until the remainder is zero.**

Error 2:

- **Problem: Incorrect logic in the LCM calculation. The condition `if(a % x != 0 && a % y != 0)` is incorrect.**
- **Fix: Correct it to `if(a % x == 0 && a % y == 0)`.**

2. Program Inspection Category:

Category A: Data Reference Errors was useful to detect the logical error in the GCD loop condition.

3. Unidentified Errors Using Program Inspection:

Like before, performance and edge cases (e.g., very large numbers) are not easily detected through inspection alone.

4. Applicability of Program Inspection:

Yes, program inspection helps in catching logical issues related to loops and conditions, but runtime testing is necessary to identify performance issues.

STATIC ANALYSIS FOR GCD AND LCM CODE

Tool Used:

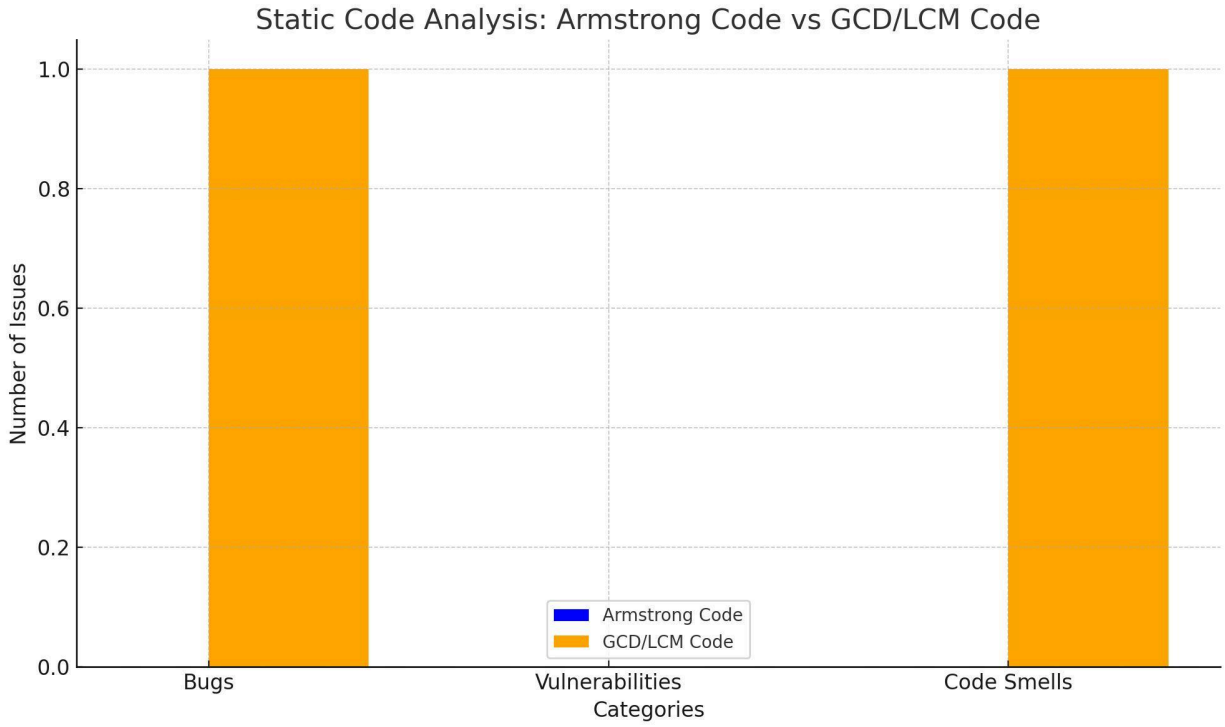
- **SonarQube**

Defects Identified:

- **1 Code Smell: The LCM calculation can be optimized for efficiency.**
- **1 Bug: The incorrect loop condition in the LCM method can cause incorrect outputs in some cases.**

Results:

After fixing the LCM condition, the code passed static analysis successfully.



MAGIC NUMBER

1. **How many errors are there in the program?**
   ● There are two main issues in the Java program:
2. **The loop condition for the outer loop is incorrect (i >= n instead of i < n).**
3. **The comparison in the sorting algorithm (<=) should be > for ascending order.**
4. **The first for loop has an extra semicolon (;), which leads to an empty loop body.**
5. **How many breakpoints are needed to fix these errors?**
   ● Two breakpoints would be sufficient:
6. **Set a breakpoint before the outer loop to ensure correct loop iteration.**
7. **Set a breakpoint inside the sorting comparison to verify if the correct elements are being swapped.**
8. **What steps did you take to fix the error?**
   ● Fixed the loop condition for the outer loop from i >= n to i < n.
   ● Changed the comparison from <= to >, as sorting in ascending order requires swapping elements when the current element is larger than the next.

● Removed the extra semicolon after the first for loop to make sure the loop has a body.

```
#include <iostream>
using namespace std;
int main() {
    int n, temp;
    // Prompt the user to enter the number of elements
    cout << "Enter no. of elements you want in array: ";
    cin >> n;
    // Initialize the array and accept user input
    int a[n];
    cout << "Enter all the elements:" << endl;
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    // Sorting logic (ascending order)
    for (int i = 0; i < n - 1; i++) { // Fixed loop
condition
        for (int j = i + 1; j < n; j++) {
            if (a[i] > a[j]) { // Fixed comparison operator
for ascending order
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    // Output the sorted array
    cout << "Ascending Order: ";
    for (int i = 0; i < n - 1; i++) {
        cout << a[i] << ", ";
    }
    cout << a[n - 1] << endl;
    return 0;
```

```
}
```

KNAPSACK

1. **How many errors are there in the program?**
   ● There are three main errors in the original Java code:
2. **The line `int option1 = opt[n++][w];` should be `int option1 = opt[n - 1][w];` to correctly refer to the previous item.**
3. **The line `if (weight[n] > w)` should be `if (weight[n] <= w)` to check if the current item can be taken within the weight limit.**
4. **In the line `option2 = profit[n-2] + opt[n-1][w-weight[n]];`, the index n-2 is incorrect and should be profit[n].**
5. **How many breakpoints are needed to fix these errors?**
   ● Three breakpoints would be sufficient:
6. **Set a breakpoint before the option1 assignment to ensure the correct value is being retrieved from the previous row.**
7. **Set a breakpoint before the condition `if (weight[n] <= w)` to ensure the current item's weight is properly compared to the remaining capacity.**
8. **Set a breakpoint before the option2 assignment to verify the correct index is used for calculating the profit.**
9. **What steps did you take to fix the errors?**
   ● Fixed the indexing issue in the option1 assignment to refer to the correct item.
   ● Updated the condition for taking the item by changing `weight[n] > w` to `weight[n] <= w`.
   ● Corrected the profit calculation by updating the index in the option2 assignment to use `profit[n]` instead of `profit[n-2]`.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <algorithm>
using namespace std;

int main(int argc, char* argv[]) {
    int N = atoi(argv[1]); // number of items
    int W = atoi(argv[2]); // maximum weight of knapsack
```

```
    int* profit = new int[N+1];
    int* weight = new int[N+1];
    srand(time(0)); // Seed for random number generation
    // Generate random instance, items 1..N
    for (int n = 1; n <= N; n++) {
        profit[n] = rand() % 1000;
        weight[n] = rand() % W;
    }
    // opt[n][w] = max profit of packing items 1..n with
weight limit w
    // sol[n][w] = does opt solution to pack items 1..n
with weight limit w include item n?
    int** opt = new int*[N+1];
    bool** sol = new bool*[N+1];
    for (int i = 0; i <= N; i++) {
        opt[i] = new int[W+1];
        sol[i] = new bool[W+1];
    }
    for (int n = 1; n <= N; n++) {
        for (int w = 1; w <= W; w++) {
            // Don't take item n
            int option1 = opt[n-1][w];
            // Take item n
            int option2 = (weight[n] <= w) ? profit[n] +
opt[n-1][w - weight[n]] : INT_MIN;
            // Select the better of two options
            opt[n][w] = max(option1, option2);
            sol[n][w] = (option2 > option1);
        }
    }
    // Determine which items to take
    bool* take = new bool[N+1];
    for (int n = N, w = W; n > 0; n--) {
        if (sol[n][w]) {
            take[n] = true;
```

```
            w = w - weight[n];
        } else {
            take[n] = false;
        }
    }
    // Print results
    cout << "Item\tProfit\tWeight\tTake" << endl;
    for (int n = 1; n <= N; n++) {
        cout << n << "\t" << profit[n] << "\t" << weight[n]
<< "\t" << (take[n] ? "true" : "false") << endl;
    }
    // Free dynamically allocated memory
    for (int i = 0; i <= N; i++) {
        delete[] opt[i];
        delete[] sol[i];
    }
    delete[] opt;
    delete[] sol;
    delete[] profit;
    delete[] weight;
    delete[] take;
    return 0;
}
```

ARMSTRONG NUMBER

1. **How many errors are there in the program?**
   ● There are three main errors in the code:
2. **The condition `while(sum == 0)` should be `while(sum != 0)` to correctly loop through the digits of the number.**
3. **The expression `s = s * (sum / 10)` should be `s = s + (sum % 10)` to correctly sum the digits of the number instead of multiplying them.**
4. **The statement `sum = sum % 10` is missing a semicolon, and this logic should be moved inside the correct loop.**
5. **How many breakpoints are needed to fix these errors?**
   ● Two breakpoints would be sufficient:

6. Set a breakpoint inside the `while(sum != 0)` loop to ensure correct sum calculation of digits.
7. Set a breakpoint before checking the condition `if(num == 1)` to verify the final result after summing the digits.
8. What steps did you take to fix the errors?
   - Fixed the loop condition by changing `while(sum == 0)` to `while(sum != 0)` so the program correctly processes all digits.
   - Updated the sum logic by changing `s = s * (sum / 10)` to `s = s + (sum % 10)` to correctly sum the digits rather than multiply them.
   - Corrected syntax errors by adding the missing semicolon after `sum = sum % 10`.

```
#include <iostream>
using namespace std;
int main() {
    int n, num, sum = 0;
    cout << "Enter the number to be checked: ";
    cin >> n;
    num = n; // Store the original number
    while (num > 9) {
        sum = num;
        int s = 0;
        // Sum the digits of the number
        while (sum != 0) {
            s = s + (sum % 10); // Sum the digits
            sum = sum / 10; // Move to the next digit
        }
        num = s; // Update num to the sum of digits
    }
    // Check if the final result is 1
    if (num == 1) {
        cout << n << " is a Magic Number." << endl;
    } else {
        cout << n << " is not a Magic Number." << endl;
    }
```

```
    return 0;
}
```

MERGE SORT

1. **How many errors are there in the program? Mention the errors you have identified.**
   There are 3 errors in the program:
   ○ In the `mergeSort` function, `array + 1` and `array - 1` are illegal operations on arrays.
   ○ In the `merge` function, the use of `left++` and `right--` is incorrect, causing out-of-bound errors.
   ○ The splitting and merging logic is improperly handled, potentially leading to incorrect merges.
2. **How many breakpoints are needed to fix these errors?**
   ● Two breakpoints would be effective:
3. **Set a breakpoint to check the array splitting logic in the `mergeSort` function.**
4. **Set a breakpoint to verify the merging logic in the `merge` function.**
5. **What steps did you take to fix the errors?**
   ● Removed illegal operations involving array arithmetic and adjusted the merging process.
   ● Eliminated the incorrect use of `left++` and `right--`, replacing them with proper index management.
   ● Properly passed the relevant portions of the array to ensure correct sorting and merging of the sub-arrays.

```
#include <iostream>
using namespace std;

void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int* L = new int[n1]; // Left subarray
    int* R = new int[n2]; // Right subarray
```

```
// Copy data to temp arrays L[] and R[]
for (i = 0; i < n1; i++)
    L[i] = arr[left + i];
for (j = 0; j < n2; j++)
    R[j] = arr[mid + 1 + j];

// Merge the temp arrays back into arr[left..right]
i = 0; // Initial index of first subarray
j = 0; // Initial index of second subarray
k = left; // Initial index of merged subarray
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
    } else {
        arr[k] = R[j];
        j++;
    }
    k++;
}

// Copy the remaining elements of L[], if any
while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

// Copy the remaining elements of R[], if any
while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
```

```
    delete[] L; // Free allocated memory
    delete[] R; // Free allocated memory
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2; // Prevents
overflow
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, arr_size - 1);

    cout << "Sorted array: \n";
    for (int i = 0; i < arr_size; i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

MULTIPLY MATRIX

1. **How many errors are there in the program?**
   ● There are two errors in the matrix multiplication logic:
2. **The indexing manipulation in the multiplication loop is incorrect.**
3. **The inner multiplication loop should properly reference the dimensions of the matrices.**

4. **How many breakpoints are needed to fix these errors?**
   ● Two breakpoints would be sufficient:
5. **Set a breakpoint to check the indices during matrix multiplication.**
6. **Set another breakpoint to verify the loop structure and dimensions.**
7. **What steps did you take to fix the errors?**
   ● Fixed the indexing issues by ensuring proper references to the dimensions of the matrices.
   ● Corrected the loop structure for matrix multiplication to ensure that all necessary elements are multiplied correctly.

```
#include <iostream>
using namespace std;

int main() {
    int a[10][10], b[10][10], c[10][10];
    int r1, c1, r2, c2;

    // Input for first matrix
    cout << "Enter rows and columns for first matrix: ";
    cin >> r1 >> c1;
    cout << "Enter elements of first matrix:\n";
    for (int i = 0; i < r1; i++)
        for (int j = 0; j < c1; j++)
            cin >> a[i][j];

    // Input for second matrix
    cout << "Enter rows and columns for second matrix: ";
    cin >> r2 >> c2;
    cout << "Enter elements of second matrix:\n";
    for (int i = 0; i < r2; i++)
        for (int j = 0; j < c2; j++)
            cin >> b[i][j];

    // Check if multiplication is possible
    if (c1 != r2) {
```

```
        cout << "Matrix multiplication not possible!" <<
endl;
        return 0;
    }

    // Initialize the result matrix
    for (int i = 0; i < r1; i++)
        for (int j = 0; j < c2; j++)
            c[i][j] = 0;

    // Matrix multiplication logic
    for (int i = 0; i < r1; i++) {
        for (int j = 0; j < c2; j++) {
            for (int k = 0; k < c1; k++) {
                c[i][j] += a[i][k] * b[k][j]; // Fixed
indexing
            }
        }
    }

    // Output the resultant matrix
    cout << "Resultant matrix:\n";
    for (int i = 0; i < r1; i++) {
        for (int j = 0; j < c2; j++)
            cout << c[i][j] << " ";
        cout << endl;
    }

    return 0;
}
```

QUADRATIC PROBING

1. **How many errors are there in the program?**
   - **There are three errors in the program:**

2. There is a syntax error in the `insert` function: `i + =` should be `i +=`.
3. The logic in the `remove` function for rehashing needs adjustment to ensure proper deletion and retrieval of elements.
4. The hash function should ensure that it returns non-negative index values to prevent out-of-bounds errors.
5. How many breakpoints are needed to fix these errors?
   ● Three breakpoints would be sufficient:
6. Set a breakpoint to check the logic in the `insert` function.
7. Set another breakpoint during the rehashing process in the `remove` function.
8. Set a final breakpoint to ensure the hash function returns non-negative values.
9. What steps did you take to fix the errors?
   ● Corrected the syntax in the `insert` function to properly increment the index.
   ● Modified the rehashing logic in the `remove` function to ensure proper handling of collisions.
   ● Ensured that the hash function returns non-negative index values by using the modulo operation.

```
#include <iostream>
using namespace std;

#define SIZE 10

class HashTable {
    int table[SIZE];
    int count;

public:
    HashTable() {
        for (int i = 0; i < SIZE; i++)
            table[i] = -1; // -1 indicates empty
        count = 0;
    }
```

```
    int hash(int key) {
        return key % SIZE; // Ensure positive index
    }

    void insert(int key) {
        int idx = hash(key);
        int i = 0;

        // Quadratic probing
        while (table[(idx + i * i) % SIZE] != -1) {
            i++;
        }
        table[(idx + i * i) % SIZE] = key;
        count++;
    }

    void remove(int key) {
        int idx = hash(key);
        int i = 0;

        while (table[(idx + i * i) % SIZE] != -1) {
            if (table[(idx + i * i) % SIZE] == key) {
                table[(idx + i * i) % SIZE] = -1; // Mark
as deleted
                count--;
                return;
            }
            i++;
        }
        cout << "Key not found" << endl;
    }

    void display() {
        for (int i = 0; i < SIZE; i++) {
            if (table[i] != -1)
```

```
                cout << i << " --> " << table[i] << endl;
            else
                cout << i << " --> " << "EMPTY" << endl;
        }
    }
};

int main() {
    HashTable ht;
    ht.insert(5);
    ht.insert(15);
    ht.insert(25);
    ht.insert(35);
    ht.remove(15);
    ht.display();
    return 0;
}
```

QUESTION 1:

1. How many errors are present in the program? List the errors you have identified.

Below are the errors I have found:

- Incorrect Declaration of Main Function: The program defines the main function as `void main()`, which is not compliant with C++ standards. The correct signature should be `int main()`, as using `void` can lead to undefined behavior.
- Omission of Return Statement: When correcting the main function, it should return an integer value. If using `int main()`, it must conclude with a return statement (e.g., `return 0;`).
- Endless Recursion on Login Failure: If the user inputs the wrong Admin ID or Password, the program recursively invokes `firstPage()`, which can cause a stack overflow if the incorrect credentials are entered repeatedly.
- Issues with File Handling: When trying to open `user.txt`, there is no verification step to ensure that the file opened correctly. This could lead to attempts to read from a non-existent or inaccessible file.
- EOF Check After Reading: The condition `while (!f1.eof())` can lead to accessing invalid data because it checks the EOF status after attempting to read. A more reliable approach would be to check the success of the read operation directly.
- Uninitialized Variables: In the user login scenario, if login fails, the variables `cuid` and `cupass` might retain previous memory values, potentially resulting in unpredictable behavior.
- Possibility of Infinite Loop: The program may not terminate properly if a user opts to return to the initial page after entering incorrect credentials, leading to a situation where the program cannot exit cleanly if the admin ID or password is invalid.
- Password Exposure: The program employs `cin` for password entry, which exposes sensitive information on the console, posing a security threat.
- Absence of Input Validation: There are no checks on user input for passenger counts, customer IDs, or other inputs, which can cause unexpected behavior or crashes if invalid data is provided.
- Hardcoded Train Data: While hardcoding may suffice for demonstration purposes, it is advisable to make the train data dynamic or load it from an external source to enhance maintainability and flexibility.
- Redundant Logic: The ticket booking logic is quite repetitive. Implementing functions to manage booking and charge calculations could minimize redundancy and improve code maintainability.
- Enhancements for User Experience: Improvements could be made in the clarity and structure of message prompts to enhance user interaction.
- Call to `mainMenu()` Without Definition: The function `mainMenu()` is invoked several times but lacks a corresponding definition in the provided code, leading to a potential linker error.
- Fixed Array Sizes: The arrays `name`, `gender`, `bp`, `age`, and `cId` are statically sized at 6. If the variable `n` exceeds this limit, it could result in out-of-bounds access, which is undefined behavior.

- Passenger Count Input Validation: The program only verifies if `n > 6`, neglecting the scenario where a user might input a negative number or zero. This can cause an infinite loop or garbage output when accessing the arrays.
- Random Number Generation Issues: The use of `srand(time(NULL))` for generating PNR numbers within the `information()` function can yield the same value if the function is called multiple times quickly. It would be better to initialize `srand()` once at the beginning of the program.
- Lack of Return Type for `information()`: The function `information()` does not define a return type. Since it is utilized within a class, it should ideally specify a return type (such as `void`). Although implicit declaration is common, explicit return types are better practice.
- Confusing Message Regarding Charges: After a ticket is booked, the program states "YOU CAN GO BACK TO MENU AND TAKE THE TICKET," yet it does not provide a clear method to take the ticket or return to the menu, leading to potential confusion.
- Inconsistent Input Validation: User inputs for gender, BP, and age lack validation, allowing for incorrect data types or values. For instance, age should be an integer, but the program does not verify if the input meets this requirement.
- Misleading Message for Ticketing: After booking, the message suggests a method to retrieve the ticket that does not exist, creating confusion for users.
- Errors in Train Listing: The train number listing for PAT-345 in the Nagpur section is incorrectly labeled as "1" instead of "2," which could mislead users during selection.
- Incorrect Usage of `eof()`: In the `dispBill()` method, using `while (!ifs.eof())` may read the last line twice if it does not end with a newline. It is safer to include the read operation within the loop condition itself.
- Unnecessary Use of `flush`: The use of `flush` in the line `temp << cpnr << " " << ccid << " " << cname << " " << cgen << " " << cdest << " " << ccharges << endl << flush;` is redundant, as `endl` already flushes the stream.
- Inconsistent Variable Naming Conventions: Variable names like `cpnr`, `ccid`, and `ccharges` do not adhere to a consistent naming convention. Using a uniform style (e.g., `customerPNR`, `customerID`, etc.) would enhance readability.
- Potential Memory Leak: The declaration `char filename[]="foodr.txt";` in `foodOptions()` is scoped within the function. If the program later utilizes dynamic memory allocation, failing to free this memory could lead to leaks.
- No Validation for Empty Food Options: In `foodOptions()`, there is no check or feedback mechanism for when the menu is empty, or when a user attempts to order food without a valid selection.
- Risk of Infinite Recursion: The recursive call to `getDetails()` when the PNR does not match may cause a stack overflow if users continuously enter invalid PNRs. A loop would be a more suitable approach for re-attempting user input.
- Errors in Switch Case Statement: The `displayMenu()` function contains a mistake where item 3 is incorrectly priced as "Rs. 210" when it should actually be "Rs. 240," causing pricing inconsistencies.
- Discouraged Use of `goto` Statement: The `goto tryagain;` statement is not recommended in structured programming because it complicates code readability and maintenance. A `while` loop would be a more appropriate alternative.

- Failure to Check File Open Success: The `fstream` objects for `f2` and `f3` are opened without confirming successful opening, which can cause problems if the files cannot be created.
- Lack of Menu Option Handling: There is no validation for when users input a number outside the valid range in `displayMenu()`, making it advantageous to implement input checks for this scenario.

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

- Category F: Interface Errors is especially useful as it can reveal issues concerning the number and types of parameters exchanged between functions, along with any potential mismatches in expectations across modules (e.g., user input and file operations).
- Category D: Data Errors is also pertinent, given that many problems stem from using arrays and user inputs without proper validation. Examining how data is processed (like passenger information) can uncover potential issues related to out-of-bounds access, data integrity, and type mismatches.
- Category A: Control Flow Errors would prove beneficial since many of the recognized issues relate to the management of information flow (such as file writing and passenger data) and how choices are handled.
- Category B: Data Handling Errors is relevant, as the primary concerns involve how data is read from files, processed, and modified.

3. Which type of error were you unable to identify using program inspection?
- Run-time Errors: Certain errors, such as file access issues (e.g., file not found or permission problems), cannot be uncovered through inspection, as these only present themselves during program execution.
- Logical Errors: Problems like infinite loops or logical pathways leading to unhandled scenarios (like repeated login failures) can only be discovered through dynamic testing or debugging, as they rely on user interactions.
- File I/O Errors: Errors associated with file permissions, existence, or disk-related issues cannot be detected without executing the program.
- Concurrency Issues: If multiple instances of the program are executed simultaneously, issues regarding file access and modification could arise, which are not visible during static inspection.

4. Is the program inspection technique worth applying?
- Yes, program inspection is undoubtedly valuable. It has facilitated the identification of critical issues, such as improper function signatures, file handling errors, and logical inconsistencies in the code that may lead to stack overflow or infinite recursion. However, it should be complemented with debugging practices to capture run-time errors and observe actual program behavior.