IT 314 Lab 7

Jeet Desai

202201474

Question 1

Reference Code : Code

Part 1:

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

- **Category A: Data Reference Errors**
    - Variables remain unassigned due to an error in the constructor's definition. The __init__ method is misspelled as _init_, which results in the constructor not being executed. Consequently, self.matrix, self.vector, and self.res are left without any initialization.
- **Category C: Computation Errors**
    - In the gauss method, if A[i][i] happens to be zero, there is a potential risk for a division by zero error during the execution of the calculation x[i] /= A[i][i].

- **Category D: Comparison Errors**
    - The diagonal_dominance method's logic may lead to inaccurate outcomes because it doesn't properly handle cases where rows contain duplicate maximum values. This can skew the results of the dominance check.
- **Category E: Control-Flow Errors**
    - The loop in get_upper_permute incorrectly uses the range k in range(i+1, n+1); it should instead be k in range(i+1, n) to avoid going beyond the allowed bounds of the matrix.
- **Category F: Interface Errors**
    - The methods assume a strict input structure (a list of lists for matrices and a list for vectors), but they lack validation mechanisms to catch input formats that don't match expectations.
- **Category G: Input/Output Errors**
    - There is no error handling in place to verify whether the input matrices are square or if the dimensions are appropriate for the operations being performed.
- **Category H: Other Checks**
    - The program requires numpy and matplotlib.pyplot to work correctly, but these libraries are not imported, which would prevent the code from running successfully.

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

Addressing **Category D (Comparison Errors)** and **Category A (Data Reference Errors)** would be the most beneficial in this scenario, as incorrect comparisons and uninitialized variables are key contributors to runtime issues and incorrect program behavior.

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

Logical Errors: While program inspection is useful for catching syntax and runtime mistakes, logical errors can still go unnoticed. For example, algorithms like Jacobi or Gauss-Seidel might perform poorly or fail to converge under certain conditions, which would not be apparent from code inspection alone.

4. **Is the program inspection technique worth applicable?**

Yes, it is highly applicable for finding common coding errors related to data handling, computations, and control flow. It's a valuable practice that improves code quality and robustness, though it should be used in conjunction with dynamic testing techniques, such as unit tests, to catch issues related to program logic and functionality.

Part 2:

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

- **Category A: Data Reference Errors**
    - The constructor is incorrectly defined as _init_ rather than __init__, which prevents the object initialization process when an instance of the Interpolation class is created.
    - The program directly accesses matrix elements in the cubicSpline and piecewise_linear_interpolation methods without confirming that the indices are valid, potentially leading to an IndexError.
    - There is no verification to ensure that the matrix elements are numeric (either integers or floats), which could lead to unexpected behavior if non-numeric data is passed.
- **Category C: Computation Errors**
    - There is a risk of division by zero in the piecewise_linear_interpolation method if two consecutive x-values are identical.
    - The err attribute is declared but not consistently initialized, which could cause inconsistencies in its usage.
- **Category D: Comparison Errors**
    - While the code does not explicitly show floating-point comparison errors, comparisons involving floating-point numbers can be imprecise. Extra care should be taken to ensure precision is managed correctly.
- **Category E: Control-Flow Errors**

    - The loop in the mynewtonint method should be checked to ensure it does not access elements beyond the array's bounds, particularly when n is small.
    - The cubicSpline function does not account for cases where the matrix has fewer than four points, which could cause errors during matrix operations.
- **Category F: Interface Errors**
    - The functions lack docstrings, making it harder for users to understand their purpose, parameters, or expected return values.
    - The program lacks input validation across the methods, leading to potential runtime errors if unexpected or invalid parameters are provided.
- **Category G: Input/Output Errors**
    - The plotting functions do not include checks for valid input data, which could cause the program to fail when attempting to plot with incorrect data.
- **Category H: Other Checks**
    - There are no mechanisms in place to capture or manage warnings that could be generated during compilation or execution, such as unused variables or other issues.

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

Category A (Data Reference Errors) seems most effective in this scenario since initialization and index handling issues are critical to avoid runtime crashes and ensure the program runs smoothly.

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

Runtime Errors: Certain issues, like floating-point inaccuracies or problems caused by unexpected input, may not be detected through static code inspection and would require dynamic analysis or runtime testing to catch.

4. **Is the program inspection technique worth applicable?**

Yes, program inspection is a very useful tool for identifying potential issues early on, ensuring code quality and adherence to good coding practices. Nevertheless, it

should be paired with other testing methods, such as unit tests, to fully cover the spectrum of possible errors and edge cases.

Part 3:

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

- **Category A: Data Reference Errors**
    - The functions fun and dfun are redundantly defined several times without any clear distinction between the equations they apply to, leading to confusion.
    - The data variable is reused across different iterations but isn't reset properly for each function call, which could lead to unexpected behavior when running multiple root-finding operations in sequence.
- **Category B: Data-Declaration Errors**
    - The variable next is used before it is properly initialized during the first iteration of the loop, potentially leading to NaN values.
    - The DataFrame df is only created after the loop, which could lead to problems if the loop terminates early, such as in cases of immediate convergence.
- **Category C: Computation Errors**
    - The function fpresent = fun(present) should also verify convergence based on |fun(present)| instead of only relying on the difference between next and present.
    - The error calculation error.append(next - present) may not correctly reflect convergence because it compares the last two iterations rather than the iterations used in the root-finding process.
- **Category D: Comparison Errors**
    - The error condition that checks the difference between next and present does not account for situations where present might be very close to the actual root without converging effectively.

    - The convergence criteria depend solely on abs(next - present) > err, ignoring the importance of the function value |fun(next)| in determining convergence.

- **Category E: Control-Flow Errors**
    - The loop could get stuck in an infinite loop if dfun(present) is zero (e.g., a vertical tangent) or if the initial guess is too far from the root.
    - There are no safeguards in place to break out of the loop after a certain number of iterations or to prevent division by zero during the calculation next = present - (fpresent / dfpresent).
- **Category F: Input/Output Errors**
    - The code lacks any form of logging or output during iterations, making it difficult to track how the algorithm is progressing over time.
    - The plot titles are not informative enough to clarify which function or root is being plotted, which may lead to confusion when analyzing the results.
- **Category G: Other Checks**
    - The code does not handle cases where the function may lack a root within the provided domain or where the derivative could cause undefined behavior.
    - Multiple plots are generated without clearing the data from previous plots, which could result in cluttered visualizations when testing multiple functions.

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

Focusing on **Category A (Data Reference Errors)** is crucial, as improper handling of inputs can cause runtime crashes and make the program difficult to debug or use effectively.

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

Non-obvious Logical Errors: These errors, like incorrect root convergence or numerical instability, often don't manifest until the program is executed with specific input values, making them hard to spot during inspection.

4. **Is the program inspection technique worth applicable?**

Yes, it is valuable for finding various types of errors and improving the structure and maintainability of the code. It is particularly useful for enhancing code readability and identifying potential pitfalls, especially in more complex numerical methods or algorithms.

Part 4:

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

- **Category A: Data Reference Errors**
    - The input matrix is assumed to be a 2D array, but the code doesn't check for this or handle incorrect input structures, which could cause errors during execution.
    - Variables like coef and poly_i are used multiple times in different scopes, which can create confusion about their purpose and may lead to unexpected outcomes.
- **Category B: Data-Declaration Errors**
    - In the plotting function plot_fun, there's a lack of safeguards in case y is empty or improperly initialized. This could result in errors when the program tries to plot.
    - There is no validation for matrix inversion before calling np.linalg.inv(ATA), meaning if the matrix is singular, the program will crash.
- **Category C: Computation Errors**
    - The line coef = coef[::-1] reverses the coefficients, but the polynomial function expects the coefficients to be in descending order. This mismatch could result in incorrect polynomial behavior.
    - During coefficient calculation, the same coef variable is overwritten for each polynomial degree, leading to ambiguity over which set of coefficients corresponds to which polynomial.
- **Category D: Comparison Errors**

    - The error tolerance err = 1e-3 is hardcoded into the function, which might not be suitable for all datasets. There's no dynamic adjustment to account for the input data's range.
    - When plotting multiple polynomial fits, the legend does not ensure distinct labeling for each polynomial, making it unclear which line corresponds to which function.
- **Category E: Control-Flow Errors**
    - The plotting function might enter an infinite loop if it processes incorrectly formatted data, especially when no points are available to plot.
    - The leastSquareErrorPolynomial function doesn't include an exit condition for poorly conditioned matrices or overly high polynomial degrees relative to the number of data points, potentially causing excessive computations or errors.
- **Category F: Input/Output Errors**
    - The program doesn't provide feedback during polynomial fitting, making it hard for the user to know whether the process is running or completed.
    - The variable name poly_i could cause confusion because it suggests a single polynomial, whereas it actually stores a polynomial object. A more descriptive name would make the code easier to understand.
- **Category G: Other Checks**
    - The function doesn't handle cases where all y values are identical, leading to a constant polynomial. This could be misleading for users expecting more varied results.
    - There are no unit tests or assertions to validate input parameters, which makes it difficult to ensure that the functions perform as expected across a range of scenarios.
- **Category H: General Code Quality**
    - The plotting code includes redundant sections that could be refactored into reusable functions, making the program more efficient and easier to maintain.

- The functions lack documentation, making it harder for users to understand the purpose and expected behavior, especially when revisiting the code after some time.

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

Category C (Computation Errors) is the most important in this case because ensuring accurate calculations is essential for numerical methods like polynomial fitting, where any minor error can lead to incorrect results.

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

Data-Specific Errors: Some errors related to specific edge cases in the input data, like having identical y values, may not become apparent during static inspection and only manifest when the program is run with certain datasets.

4. **Is the program inspection technique worth applicable?**

Yes, program inspection is a useful strategy. It allows for identifying a broad range of issues, especially those related to data handling and computational accuracy. This technique is particularly beneficial for enhancing code structure and reliability, although it should be supplemented with testing methods like unit testing and validation checks to ensure robustness in various scenarios.

Question 2

1) Armstrong Number:

Original Code :

```
//Armstrong Number
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 a Armstrong Number");
    }
```

I. Errors in the code:

1. **Line 9: remainder = num / 10;**
    - This line is mistakenly dividing the number by 10 to extract the last digit, which results in the quotient, not the remainder. It should use num % 10 to get the actual remainder (the last digit).
2. **Line 12: num = num % 10;**
    - The operation in this line intends to remove the last digit but is using the modulus operator incorrectly. The correct operation is num = num / 10; to drop the last digit from the number.
3. **Missing closing bracket:**
    - The code should include a closing bracket to properly finish the method.

II. Breakpoints:

- **Initial values:** Check the variables num, check, and remainder at the start.
- **Value updates:** Set a breakpoint to observe how remainder changes after each division.
- **Progression of num:** Track how num is modified after each iteration.

III. Steps to fix the errors:

1. **Correct operations:**

- Modify remainder = num / 10 to remainder = num % 10 on line 9.
- Change num = num % 10 to num = num / 10 on line 12.

IV. FIXED CODE

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

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

      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:

Original Code :

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 it 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();
    }
}
```

I. Errors in the code:

1. **GCD Calculation (Line 13):**
    - The condition while(a % b == 0) is flawed. It should be while(a % b != 0) to ensure the loop continues until the remainder is zero. The current logic may cause the loop to run infinitely when a % b == 0.
2. **LCM Calculation (Line 24):**
    - The condition if(a % x != 0 && a % y != 0) is incorrect. It will return a value when a is not divisible by either x or y, but the intention is to find a number divisible by both. It should be if(a % x == 0 && a % y == 0).

II. Breakpoints:

- **Line 13:** Check the logic within the GCD loop.
- **Line 24:** Verify the condition for LCM computation.
- **Line 31:** Confirm the correctness of the final GCD and LCM values.

III. Steps to fix the errors:

1. **Fix the GCD calculation** by updating the while loop condition.
2. **Correct the LCM condition** in the if statement.

IV. FIXED CODE:

```
public class GCD_LCM {


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


      while (a % b != 0) { // Continue until remainder is 0
         r = a % b;
         a = b;
         b = r;
      }
      return r; // The GCD is the last non-zero remainder
   }


   static int lcm(int x, int y) {
      int a = (x > y) ? x : y;
      while (true) {
         if (a % x == 0 && a % y == 0) // The least common multiple is divisible by both
            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();
  }
}
```

3) Knapsack:

Original 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];

      // 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);
    }

    // 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][W+1];
    boolean[][] sol = new boolean[N+1][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++][w];

        // take item n
        int option2 = Integer.MIN_VALUE;
        if (weight[n] > w) option2 = profit[n-2] + opt[n-1][w-weight[n]];

        // select better of two options
        opt[n][w] = Math.max(option1, option2);
        sol[n][w] = (option2 > option1);
      }
    }

    // determine which items to take
    boolean[] take = new boolean[N+1];
    for (int n = N, w = W; n > 0; n--) {
      if (sol[n][w]) { take[n] = true;  w = w - weight[n]; }
      else           { take[n] = false;                    }
    }

    // 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]);
        }
    }
}
```

I. Errors in the code:

1. **Line 20: int option1 = opt[n++][w];**
    - The post-increment operator n++ causes the index n to increase prematurely, which will result in an out-of-bounds error. Replace it with opt[n][w] to avoid this issue.
2. **Line 24: option2 = profit[n-2] + opt[n-1][w-weight[n]];**
    - The reference to profit[n-2] is incorrect. It should simply use profit[n] because we are working with the current item n.
3. **Line 32: Update logic for take[n].**
    - The logic inside if (sol[n][w]) is incorrect when updating the weight w = w - weight[n]. This can cause weight to become negative, leading to an out-of-bounds error.

II. Breakpoints:

- **Line 20:** Check how option1 is being calculated.
- **Line 24:** Confirm the logic behind option2.
- **Line 32:** Ensure the items are correctly selected in the loop.

III. Steps to fix the errors:

1. **Fix the increment error** by removing n++ in line 20.
2. **Correct the profit reference** in line 24 by changing profit[n-2] to profit[n].
3. **Adjust the weight logic** in line 32 to prevent errors.

IV. FIXED CODE:

```
public class Knapsack {
   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      int W = Integer.parseInt(args[1]);

      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][w]; // No increment
            int option2 = Integer.MIN_VALUE;

            if (weight[n] <= w)
               option2 = profit[n] + opt[n - 1][w - weight[n]]; // Corrected profit index

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

```
            sol[n][w] = (option2 > option1);
        }
      }

      boolean[] take = new boolean[N + 1];
      for (int n = N, w = W; n > 0; n--) {
        if (sol[n][w]) {
          take[n] = true;
          w -= weight[n]; // Correct weight update
        } else {
          take[n] = false;
        }
      }

      System.out.println("item" + "\t" + "profit" + "\t" + "weight" + "\t" + "take");
      for (int n = 1; n <= N; n++) {
        System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" + take[n]);
      }
    }
}
```

4) Magic Number:

Original Code :

```
// Program to check if number is Magic number in JAVA
import java.util.*;
public class MagicNumberCheck
{
   public static void main(String args[])
```

```
  {
    Scanner ob=new Scanner(System.in);
    System.out.println("Enter the number to be checked.");
    int n=ob.nextInt();
    int sum=0,num=n;
    while(num>9)
    {
      sum=num;int s=0;
      while(sum==0)
      {
        s=s*(sum/10);
        sum=sum%10
      }
      num=s;
    }
    if(num==1)
    {
      System.out.println(n+" is a Magic Number.");
    }
    else
    {
      System.out.println(n+" is not a Magic Number.");
    }
  }
}
```

I. Errors in the code:

1. **Line 13: while(sum == 0)**

- This condition is incorrect because the loop should continue as long as there are digits to process. The correct condition should be while(sum > 0) to ensure the loop runs while there are digits left to sum.

2. **Line 14: s = s * (sum / 10)**
    - The logic here is incorrect. Instead of multiplying s by the quotient, we should be adding the remainder of the division to s. The corrected line should be s = s + (sum % 10) to accumulate the sum of the digits.
3. **Line 15: sum = sum % 10**
    - This operation is intended to remove the last digit from sum, but the modulus operator is incorrect here. The line should use integer division to update sum, i.e., sum = sum / 10.

II. Breakpoints:

- **Line 12:** Set a breakpoint to check how the digits are processed inside the loop.
- **Line 14:** Verify if s is updated correctly with the sum of the digits.
- **Line 19:** Check if the correct result is computed as a magic number.

III. Steps to fix the errors:

1. **Fix the loop condition** by changing while(sum == 0) to while(sum > 0) on line 13.
2. **Correct the summing logic** by changing s = s * (sum / 10) to s = s + (sum % 10) on line 14.
3. **Fix the digit removal** by updating sum = sum % 10 to sum = sum / 10 on line 15.

IV. FIXED CODE:

```
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 be checked.");
   int n = ob.nextInt();
   int num = n; // Copy of the original number
   int sum = 0;

   // Reduce the number until it becomes a single digit
   while (num > 9) {
      sum = num;
      int s = 0;

      // Summing the digits of the current number
      while (sum > 0) {  // Corrected loop condition
         s = s + (sum % 10);  // Accumulate sum of digits
         sum = sum / 10;      // Remove the last digit
      }
      num = s;  // Update num with the sum of digits
   }

   // Check if the resulting single digit is 1
   if (num == 1) {
      System.out.println(n + " is a Magic Number.");
   } else {
      System.out.println(n + " is not a Magic Number.");
   }
```

```
        ob.close();
    }
}
```

5) Merge Sort:

Original Code :

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

import java.util.*;
```

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

    // Places the elements of the given array into sorted order
    // using the merge sort algorithm.
    // 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+1);
            int[] right = rightHalf(array-1);

            // 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;
    }

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

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

I. Errors in the code:

1. **Line 15: int[] left = leftHalf(array+1);**

- The expression array + 1 is invalid because it adds an integer to an array, which is not allowed. The correct method call should be leftHalf(array) without modification.

2. **Line 16: int[] right = rightHalf(array-1);**
    - Similarly, subtracting an integer from an array is not allowed. This should also call the method directly as rightHalf(array).
3. **Line 21: merge(array, left++, right--);**
    - The post-increment (left++) and post-decrement (right--) are incorrect because you cannot increment or decrement arrays. Simply pass left and right as they are, without modifying them.

II. Breakpoints:

- **Line 15:** Ensure that the left array is created properly by the leftHalf method.
- **Line 16:** Check if the right array is generated correctly.
- **Line 21:** Confirm that the merge operation works as expected without modifying the arrays.

III. Steps to fix the errors:

1. **Fix the array input** by removing the invalid operations from leftHalf(array+1) and rightHalf(array-1).
2. **Fix the merge operation** by removing the increment and decrement operators in merge(array, left++, right--).

IV. FIXED CODE:

```
import java.util.Arrays;


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));
}

// Implements merge sort algorithm
public static void mergeSort(int[] array) {
    if (array.length > 1) {
        int[] left = leftHalf(array);  // Corrected: removed +1
        int[] right = rightHalf(array); // Corrected: removed -1

        // Recursively sort both halves
        mergeSort(left);
        mergeSort(right);

        // Merge the sorted halves into one
        merge(array, left, right);  // Corrected: no post-increment/decrement
    }
}

// Splits the array into left half
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;
}

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

// Merges the two halves back into one array
public static void merge(int[] result, int[] left, int[] right) {
    int i1 = 0; // Index for left array
    int i2 = 0; // Index for right array

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

```
        }
    }
}
```

6) Multiply Matrices:

Original Code :

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

class MatrixMultiplication
{
    public static void main(String args[])
```

```
{
  int m, n, p, q, sum = 0, c, d, k;

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

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

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

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

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

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

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

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

    for ( c = 0 ; c < m ; c++ )
    {
      for ( d = 0 ; d < q ; d++ )
```

```
          {
            for ( k = 0 ; k < p ; k++ )
            {
               sum = sum + first[c-1][c-k]*second[k-1][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");
        }
      }
    }
}
```

I. Errors in the code:

1. **Line 44: sum = sum + first[c-1][c-k]*second[k-1][k-d];**
    - The matrix access logic is incorrect due to the negative indexing. Using c-1 and k-1 will cause an ArrayIndexOutOfBoundsException. The correct indices should directly reference the elements of the matrices using first[c][k] and second[k][d] without decrementing.

II. Breakpoints:

- **Line 44:** Place a breakpoint to check the matrix multiplication process, specifically focusing on how elements are accessed.

III. Steps to fix the errors:

1. **Correct the indexing:** Update line 44 to use the correct indices: first[c][k] and second[k][d].

IV. FIXED 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 these dimensions can't be multiplied.");
} 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 < n; k++) {
            sum += first[c][k] * second[k][d];  // Corrected index usage
         }
         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.println();
        }
    }
  }
}
```

7. Quadratic Probing:

Original Code :

```
import java.util.Scanner;
/** Class QuadraticProbingHashTable **/
class QuadraticProbingHashTable{
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

    /** Constructor **/
    public QuadraticProbingHashTable(int capacity)
    {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }
    /** Function to clear hash table **/
    public void makeEmpty()
    {
        currentSize = 0;
        keys = new String[maxSize];
```

```
    vals = new String[maxSize];
}
/** Function to get size of hash table **/
public int getSize()
{
    return currentSize;
}
/** Function to check if hash table is full **/
public boolean isFull()
{
    return currentSize == maxSize;
}
```

```
/** Function to check if hash table is empty **/
public boolean isEmpty()
{
    return getSize() == 0;
}
/** Fucntion to check if hash table contains a key **/
public boolean contains(String key)
{
    return get(key) !=  null;
}
/** Functiont to get hash code of a given key **/
private int hash(String key)
{
    return key.hashCode() % maxSize;
}

/** Function to insert key-value pair **/
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 + = (i + h / h--) % maxSize;
    } while (i != tmp);
```

```
}
/** Function to get value for a given key **/
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;
        System.out.println("i "+ i);
    }
    return null;
}
/** Function to remove key and its value **/
public void remove(String key)
{
    if (!contains(key))
        return;
    /** find position key and delete **/
    int i = hash(key), h = 1;
    while (!key.equals(keys[i]))
        i = (i + h * h++) % maxSize;
    keys[i] = vals[i] = null;
    /** rehash all keys **/
    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--;
}
```

```
    /** Function to print HashTable **/

    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();
    }
}

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

        /** maxSizeake object of QuadraticProbingHashTable **/
        QuadraticProbingHashTable qpht = new QuadraticProbingHashTable(scan.nextInt() );
        char ch;
        /**  Perform QuadraticProbingHashTable operations  **/
        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;
}

/** Display hash table **/
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');
    }
}
```

I. Errors in the code:

1. **Line 53: i += (i + h / h--) % maxSize;**
    - The arithmetic operation used here is flawed. The expression involving += and division is incorrect and will not properly increment i. The correct operation should use quadratic probing by modifying i = (i + h * h++) % maxSize.
2. **Line 110: Incomplete comment block.**
    - The comment block before the object creation is unfinished, causing confusion in the code's readability.

II. Breakpoints:

- **Line 53:** To check the probing logic and how i is incremented during collisions.

III. Steps to fix the errors:

1. **Fix the quadratic probing logic** in line 53 by using i = (i + h * h++) % maxSize.
2. **Complete the comment** for clarity.

IV. FIXED 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 = new 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 = (i + h * h++) % maxSize;  // Corrected quadratic probing logic
    } 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("Enter size of the hash table:");
        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:

Original Code :

```
// sorting the array in ascending order
import java.util.Scanner;
public class Ascending _Order
{
    public static void main(String[] args)
    {
        int n, temp;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no. of elements you want in 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]);
    }
}
```

I. Errors in the code:

1. **Line 9: public class Ascending _Order**
    - The class name contains a space, which is not allowed in Java. It should be renamed AscendingOrder without a space or underscore.
2. **Line 18: Incorrect for-loop condition (i >= n)**
    - The condition should be i < n for the loop to run correctly. Additionally, the semicolon at the end of the for-loop prevents proper execution.
3. **Line 21: Wrong sorting condition (if (a[i] <= a[j]))**
    - For sorting in ascending order, the condition should be if (a[i] > a[j]) to swap elements when the left value is greater than the right one.

II. Breakpoints:

- **Line 18:** Set a breakpoint to monitor the loop's functionality.
- **Line 21:** Check if the sorting condition behaves correctly for ascending order.

III. Steps to fix the errors:

1. **Fix the class name** by removing the space and underscore.
2. **Correct the loop condition** by using i < n and remove the semicolon.
3. **Fix the sorting logic** by changing the condition to if (a[i] > a[j]).

IV. FIXED 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 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();
        }

        // Corrected sorting loop
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) {  // Corrected condition for ascending order
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }
```

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

9) Stack Implementation:

Original Code :

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

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();
    }
}
public class StackReviseDemo {

    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);

        newStack.display();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.display();
    }
}
```

I. Errors in the code:

1. **Line 18 (push method): top--**
    - The top index should be incremented when pushing a value onto the stack, not decremented.
2. **Line 26 (pop method): top++**
    - The top index should be decremented when popping a value from the stack.
3. **Line 35 (display method): i > top**
    - The loop condition is incorrect. It should be i <= top to correctly iterate and display all elements in the stack.

II. Breakpoints:

- **Line 18:** Monitor the push method to see how top changes.
- **Line 26:** Check the pop method for correct index behavior.
- **Line 35:** Verify the display method for correct iteration.

III. Steps to fix the errors:

1. **Fix the push logic** by changing top-- to top++.
2. **Fix the pop logic** by changing top++ to top--.
3. **Fix the display condition** by changing i > top to i <= top.

IV. FIXED CODE:

```
import java.util.Arrays;


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++;  // Corrected: increment top
      stack[top] = value;
   }
}

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

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

    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return;
        }
        for (int i = 0; i <= top; i++) {  // Corrected loop condition
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }
}

public class StackReviseDemo {
    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);

        newStack.display();  // Display stack before popping
```

```
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();

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

10) Tower of Hanoi:

Original Code :

```
//Tower of Hanoi
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 ++, inter--, from+1, to+1)
      }
    }
  }
```

I. Errors in the code:

1. **Line 16: doTowers(topN ++, inter--, from+1, to+1)**
    - The post-increment (topN++) and post-decrement (inter--) are unnecessary. Additionally, adding 1 to the character variables from and to will convert them into integers, which is incorrect.

II. Breakpoints:

- **Line 16:** Set a breakpoint to monitor how recursion is handled and if the variables are passed correctly.

III. Steps to fix the errors:

1. **Remove the post-increment/decrement** and pass the variables as they are, without modifying them.

IV. FIXED CODE:

java

Copy 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 {
            // Recursive call to move (n-1) disks from 'from' to 'inter'
            doTowers(topN - 1, from, to, inter);

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

            // Recursive call to move (n-1) disks from 'inter' to 'to'
            doTowers(topN - 1, inter, from, to);  // Corrected recursion
        }
    }
}
```