Name: Mavdiya Sujal Pravinbhai

ID: 202201040

1.Program Inspection:

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

Logical Flaws

- **Month and Day Adjustment Logic**: In the `AddMonths` function, adding months can sometimes lead to a situation where the day number surpasses the number of days in the resulting month. For instance, when adding one month to January 31st, the resulting date might fall on February 28th or 29th, depending on whether it's a leap year. This could lead to inconsistent behavior if users expect that the day always remains at the month's end.

Assumptions Regarding Date Validity

- **Date Range Constraints**: The internal logic, particularly when relying on Julian Day Numbers (JDN), makes assumptions about the input dates being within a reasonable range. Dates provided that are too far in the past or future may cause the algorithm to return erroneous results. For example, dates before 4714 BC or those surpassing the system's maximum supported date may not be handled properly, potentially leading to overflows or unexpected outputs.

Undefined Behavior in `DoIsHoliday`

- **Incomplete Functionality**: The function `DoIsHoliday(dt)` is repeatedly invoked within loops, but no definition for this function is provided in the given code snippets. If this function is missing or improperly implemented, the process for determining holidays will fail entirely, as the logic depends on this function.

Error Handling Concerns

- **Insufficient Error Validation**: The current implementation lacks comprehensive error-checking mechanisms. For example, there are no safeguards in place to detect when the `dtStart` date is later than `dtEnd`. In such cases, the program simply returns no results without providing any clear indication of what went wrong. Incorporating error detection and providing helpful error messages would significantly improve the overall user experience.

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

To ensure the code is reliable and error-free, two key inspection techniques stand out: **Formal Code Review** and **Static Code Analysis**.

- **Formal Code Review**: Engaging in a structured peer review process can help uncover logical flaws and ensure the code adheres to established coding standards. Since date and time manipulations can be tricky, involving multiple reviewers will increase the chances of catching subtle bugs or faulty assumptions. Reviewers should give special attention to edge cases, such as leap year considerations, holiday logic, and whether the assumptions embedded in the code are universally valid.
- **Static Code Analysis**: Utilizing static analysis tools (e.g., SonarQube, Clang Static Analyzer) would offer automated insights into potential code issues. These tools are excellent for detecting unreachable code, memory management problems, and ensuring conformity to coding standards. In the context of date manipulation, they could flag missing validations, such as checking whether a date is legitimate before determining if it is a holiday.

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

Types of Errors, Particularly Dynamic Ones, May Not Be Detected During Code Inspection

Some errors, especially those related to runtime behavior, might escape detection during static code reviews or inspections:

- **Runtime Errors**: These problems only appear when the program is executed, rather than during the compilation phase. Examples of such issues include:
    - Incorrect handling of user inputs, such as invalid date entries that result in out-of-bounds calculations.
    - Logical issues that arise only with specific data, like unexpected behavior when calculating holidays in years with unique leap year rules.
- **Performance Issues**: While some inefficiencies can be detected by static analysis tools, these tools can't assess the performance of the program under real-world usage. For instance, if a user requests date ranges spanning several years, performance slowdowns may only become evident at runtime due to inefficient day-by-day iteration over long time periods.

4. Is the program inspection technique is worth applicable?

The Value of Applying Program Inspection Techniques

Using program inspection methods offers multiple advantages, making it a highly beneficial practice:

- **Early Detection of Bugs**: Regular inspections allow developers to catch issues early in the development process, which is much more cost-effective than addressing them after deployment. By identifying errors in date-related calculations ahead of time, you can avoid the need for time-consuming debugging later.
- **Enhanced Code Quality**: Inspections ensure adherence to coding standards and best practices, leading to a cleaner, more maintainable codebase. For example, using consistent naming conventions improves the ease with which other developers can navigate and understand the code.
- **Fostering Collaboration and Knowledge Transfer**: Code reviews create opportunities for team collaboration, allowing developers to share knowledge and insights. These sessions can be particularly useful for discussing the complexities and potential pitfalls of date and time calculations.
- **Better Documentation**: Inspections typically lead to improved documentation, which is essential for complex features like date-time handling. Comprehensive documentation helps future developers understand the thought process behind key implementation decisions, ensuring the code remains maintainable over time.

2. CODE DEBUGGING:

1. Armstrong number:

```java
//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");
    }
```

```
Input: 153
Output: 153 is an armstrong Number.
```

1. Errors Detected:

- **Digit Extraction Issue**: The last digit is extracted incorrectly using division (`/`). It should be extracted using modulus (`%`).
- **Number Reduction Error**: The number is incorrectly reduced using modulus (`%`). It should be reduced using division (`/`).

2. Breakpoints Needed:

- One for correcting the digit extraction method.
- One for ensuring proper number reduction.

. Fix Steps:

1. Change the extraction method to `number % 10` to get the last digit.
2. Update the reduction method to `number / 10` to correctly reduce the number.

3. Corrected Code:

```
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 += (int) Math.pow(remainder, 3);
 num /= 10;
 }
 if (check == n)
 System.out.println(n + " is an Armstrong Number");
 else
 System.out.println(n + " is not an Armstrong Number");
 }
}
```

```
//program to calculate the GCD and LCM of two given numbers
import java.util.Scanner;
```

```java
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();
    }
}

Input:4 5
Output: The GCD of two numbers is 1
        The GCD of two numbers is 20
```

1. Errors Detected:

- **GCD Calculation Flaw**: The condition in the GCD calculation should be `a % b != 0` instead of `a % b == 0`.
- **LCM Calculation Error**: The LCM calculation incorrectly uses `if(a % x != 0 && a % y != 0)`; it should be `if(a % x == 0 && a % y == 0`.

2. Breakpoints Needed:

- One to correct the GCD condition.
- One to fix the LCM condition.

Fix Steps:

1. Modify the GCD condition from `while(a % b == 0)` to `while(a % b != 0)`.
2. Adjust the LCM condition from `if(a % x != 0 && a % y != 0)` to `if(a % x == 0 && a % y == 0)`.

3. Corrected Code:

```
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;
 b = (x < y) ? x : y;
 r = b;
 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;
 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();
 }
}
```

```
//Knapsack
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]);
        }
    }
}
Input: 6, 2000
Output:

Item    Profit  Weight  Take
1   336 784 false
2   674 1583    false
```

```
3   763 392 true
4   544 1136    true
5   14  1258    false
6   738 306 true
```

1. Errors Detected:

- **Knapsack Logic Mistake**: The expression `opt[n++][w]` should be `opt[n-1][w]` to accurately compute the profit of not including the item.
- **Comparison Error**: The check `if(weight[n] > w)` should be corrected to `if(weight[n] <= w)` to ensure the item is considered if its weight fits.
- **Profit Reference Issue**: The expression `profit[n-2]` should be corrected to `profit[n]` to reference the current item correctly.

2. Breakpoints Needed:

- One to correct the logic for not taking the item.
- One to fix the condition for taking the item.
- One to adjust the profit reference when including the item.

Fix Steps:

1. Change `opt[n++][w]` to `opt[n-1][w]`.
2. Modify the condition from `if(weight[n] > w)` to `if(weight[n] <= w)`.
3. Update the profit reference from `profit[n-2]` to `profit[n]`.

3. Corrected Code:

```
public class Knapsack {
    public static void main(String[] args) {
        int numItems = Integer.parseInt(args[0]);
        int capacity = Integer.parseInt(args[1]);
        int[] values = new int[numItems+1];
        int[] weights = new int[numItems+1];

        for (int i = 1; i <= numItems; i++) {
            values[i] = (int) (Math.random() * 1000);
            weights[i] = (int) (Math.random() * capacity);
        }

        int[][] maxProfit = new int[numItems+1][capacity+1];
        boolean[][] decisions = new boolean[numItems+1][capacity+1];

        for (int i = 1; i <= numItems; i++) {
            for (int w = 1; w <= capacity; w++) {
```

```
                int excludeItem = maxProfit[i-1][w];
                int includeItem = Integer.MIN_VALUE;

                if (weights[i] <= w)
                    includeItem = values[i] + maxProfit[i-1][w - weights[i]];

                maxProfit[i][w] = Math.max(excludeItem, includeItem);
                decisions[i][w] = (includeItem > excludeItem);
            }
        }

        boolean[] selected = new boolean[numItems+1];
        for (int i = numItems, w = capacity; i > 0; i--) {
            if (decisions[i][w]) {
                selected[i] = true;
                w = w - weights[i];
            } else {
                selected[i] = false;
            }
        }

        // print results
        System.out.println("item" + "\t" + "value" + "\t" + "weight" + "\t" +
"selected");
        for (int i = 1; i <= numItems; i++) {
            System.out.println(i + "\t" + values[i] + "\t" + weights[i] + "\t"
+ selected[i]);
        }
    }
}
```

4. Magic Numbers:

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

Input: Enter the number to be checked 119
Output 119 is a Magic Number.
Input: Enter the number to be checked 199
Output 199 is not a Magic Number.
```

1. Errors Detected:

- **Loop Condition Flaw**: The condition `sum == 0` in the inner loop should be `sum > 0` for proper digit extraction.
- **Incorrect Summation Logic**: The statement `s = s * (sum / 10)` should be `s = s + (sum % 10)` to accumulate the digit sum correctly.
- **Syntax Error**: The line `sum = sum % 10` is missing a semicolon.

2. Breakpoints Needed:

- One to correct the inner loop condition.
- One to fix the digit extraction and summation logic.
- One to address the missing semicolon.

Fix Steps:

1. Change the loop condition from `while(sum == 0)` to `while(sum > 0)`.
2. Modify the summation line from `s = s * (sum / 10)` to `s = s + (sum % 10)`.
3. Add a semicolon at the end of `sum = sum % 10`.

3. Corrected Code:

```java
import java.util.Scanner;

public class MagicNumberChecker {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Please enter a number to verify if it's a magic
number:");

        int originalNumber = scanner.nextInt();
        int currentNumber = originalNumber;

        while (currentNumber > 9) {
            int digitSum = 0;
            while (currentNumber > 0) {
                digitSum += currentNumber % 10;
                currentNumber /= 10;
            }
            currentNumber = digitSum;
        }

        if (currentNumber == 1) {
            System.out.println(originalNumber + " is a magic number.");
        } else {
            System.out.println(originalNumber + " is not a magic number.");
        }

        scanner.close();
    }
}
```

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

import java.util.*;
```

```java
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++;
            }
        }
    }
 }

Input: before 14 32 67 76 23 41 58 85
       after 14 23 32 41 58 67 76 85
```

1. Errors Detected:

- **Incorrect Array Slicing**: In `mergeSort`, `int[] left = leftHalf(array + 1)` and `int[] right = rightHalf(array - 1)` should be `int[] left = leftHalf(array)` and `int[] right = rightHalf(array)`.
- **Improper Merge Call**: The function call `merge(array, left++, right--)` should be `merge(array, left, right)`, as the increment/decrement operators are inappropriate here.

2. Breakpoints Needed:

- One to fix the array slicing logic.
- One to correct the incorrect increment/decrement in the merge operation.

Fix Steps:

1. Change `array + 1` and `array - 1` to just `array` in the calls to `leftHalf` and `rightHalf`.
2. Remove the increment/decrement operators from `merge(array, left++, right--)`.

3. Corrected Code:

```
import java.util.Arrays;
```

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

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

Input: Enter the number of rows and columns of first matrix
       2 2
       Enter the elements of first matrix
       1 2 3 4
       Enter the number of rows and columns of first matrix
       2 2
       Enter the elements of first matrix
       1 0 1 0
Output: Product of entered matrices:
        3 0
```

```
        7 0
```

1. Errors Detected:

- **Indexing Errors**: Incorrect array indices are utilized in multiplication: `first[c-1][c-k]` should be `first[c][k]`, and `second[k-1][k-d]` should be `second[k][d]`.

2. Breakpoints Needed:

- One to correct the matrix indices used in the multiplication.

a. Fix Steps:

1. Change `first[c-1][c-k]` to `first[c][k]`.
2. Change `second[k-1][k-d]` to `second[k][d]`.

3. Corrected 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 += 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");
            }
        }
    }
}
```

```
  /**

    *   Java Program to implement Quadratic Probing Hash Table

    **/
```

```java
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');

        }

    }
```

Input:

```
Hash table test

Enter size: 5
Hash Table Operations
1. Insert
2. Remove
3. Get
4. Clear
5. Size

1

Enter key and value
```

```
c computer
d desktop
h harddrive

Output:
Hash Table:
c computer
d desktop
h harddrive
```

1. Errors Detected:

- **Index Calculation Errors**: The insert, get, and remove methods contain incorrect index calculations that could lead to out-of-bounds errors or wrong data retrieval.
- **Ineffective Clearing in makeEmpty**: The `makeEmpty` method does not properly clear the arrays, resetting size without nullifying elements.

2. Breakpoints Needed:

- Total Breakpoints: 4
  - Breakpoint 1: In `insert` to verify index calculations.
  - Breakpoint 2: In `get` to validate key retrieval.
  - Breakpoint 3: In `remove` for proper key removal.
  - Breakpoint 4: In `makeEmpty` to confirm array elements are cleared.

a. Fix Steps:

1. Update index calculations in `insert` to use `(tmp + h * h) % maxSize`.
2. Apply the same index calculation in `get` and `remove`.
3. In `makeEmpty`, iterate through the arrays to nullify elements before resetting size.

3. Corrected 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() {
        for (int i = 0; i < maxSize; i++) {
            keys[i] = null;
            vals[i] = null;
        }
        currentSize = 0;
    }

    public int getSize() {
        return currentSize;
    }

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

    private int hash(String key) {
        return Math.abs(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 = (tmp + h * h) % maxSize;
            h++;
        } while (keys[i] != null);
    }

    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;
            h++;
        }
        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;
        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]);
    }
}

public class QuadraticProbingHashTableTest {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        QuadraticProbingHashTable qpht = new
QuadraticProbingHashTable(scan.nextInt());
        char ch;
        do {
            System.out.println("\n1. insert \n2. remove\n3. get\n4. clear\n5.
size");
            int choice = scan.nextInt();
            switch (choice) {
                case 1:
                    qpht.insert(scan.next(), scan.next());
                    break;
                case 2:
                    qpht.remove(scan.next());
                    break;
                case 3:
                    System.out.println("Value = " + qpht.get(scan.next()));
                    break;
                case 4:
                    qpht.makeEmpty();
                    break;
                case 5:
                    System.out.println("Size = " + qpht.getSize());
                    break;
                default:
                    System.out.println("Wrong Entry \n");
                    break;
```

```
            }
            qpht.printHashTable();
            ch = scan.next().charAt(0);
        } while (ch == 'Y' || ch == 'y');
    }
}
```

```
// 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]);
    }
}

Input: Enter no. of elements you want in array: 5
       Enter all elements:
       1 12 2 9 7
       1 2 7 9 12
```

1. Errors Detected:

- **Inefficient Sorting**: The sorting algorithm implemented uses a nested loop, resulting in a time complexity of $O(n^2)$, which can lead to performance issues for larger arrays.
- **Lack of Input Validation**: There is no validation for user input, which may result in unexpected behavior if non-integer values are entered.

2. Breakpoints Needed:

- Total Breakpoints: 2
  - Breakpoint 1: After reading the number of elements to validate `n`.
  - Breakpoint 2: After sorting to inspect the array contents.

a. Fix Steps:

1. Replace the nested loop sorting with a more efficient method, like `Arrays.sort()`, which has a time complexity of O(n log n).
2. Implement input validation to ensure that user inputs for the number of elements and their values are valid integers.

3. Corrected Code:

```
import java.util.Arrays;
import java.util.Scanner;

public class AscendingOrder {
    public static void main(String[] args) {
        int n;
        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();
        }
```

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

```
//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();
    }
}


output: 10
    1
    50
    20
    90

    10
```

1. Errors Detected:

- **Push Logic Error**: The top index is decremented before value assignment, causing an `ArrayIndexOutOfBoundsException`. It should increment for a push.
- **Pop Logic Error**: The top index is incorrectly incremented, leading to an `ArrayIndexOutOfBoundsException` during pop operations.
- **Display Logic Mistake**: The display loop condition uses `i > top` instead of `i <= top`, preventing any stack elements from being printed.

2. Breakpoints Needed:

- Total Breakpoints: 3
    - Breakpoint 1: After the `push` method to verify element addition.
    - Breakpoint 2: After the `pop` method to check the stack's state.
    - Breakpoint 3: Before the `display` method to ensure the stack contents are correct.

a. Fix Steps:

1. In the `push` method, change the logic to increment `top++` before assigning the value.
2. Adjust the `pop` method to decrement `top` only when the stack is not empty.
3. Modify the display method loop condition to `i <= top` for proper output of stack elements.

3. Corrected 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++;
            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();
    }
}
```

```
//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)
      }
   }
}

Output: Disk 1 from A to C
    Disk 2 from A to B
    Disk 1 from C to B
    Disk 3 from A to C
    Disk 1 from B to A
    Disk 2 from B to C
    Disk 1 from A to C
```

1. Errors Detected:

- **Missing Base Case for doTowers**: In the `doTowers` method, there is no return statement after moving a single disk, potentially causing incorrect execution upon recursion unwinding.
- **Disk Number Printing Issue**: The print statement assumes disk numbering starts at 1, which may lead to formatting issues for more than 9 disks without leading zeros.

2. Breakpoints Needed:

- Total Breakpoints: 2
    - Breakpoint 1: After the first `doTowers` call to check recursive processing.
    - Breakpoint 2: Before printing the move to verify disk numbers and rod labels.

Fix Steps:

1. Add a return statement after moving a single disk to prevent further processing.
2. Modify the print statement to ensure proper formatting of disk numbers, especially for more than 9 disks.

3. Corrected Code:

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

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

Code Link: 2500LOC

https://github.com/wxWidgets/wxWidgets/blob/master/src/c ommon/datetime.cpp

Static Analysis Tools

3.Choose a static analysis tool (in Java, Python, C, C++) in any programming language of your interest and identify the defects. You can also choose your own code fragment from GitHub (more than 2000 LOC) in any programming language to perform static analysis.

```
Checking static_code.cpp: __WINDOWS__;wxUSE_DATETIME...
Checking static_code.cpp: wxDEBUG_LEVEL;wxUSE_DATETIME...
Checking static_code.cpp: wxHAS_STRFTIME;wxUSE_DATETIME...
Checking static_code.cpp: wxUSE_DATETIME...
Checking static_code.cpp: wxUSE_DATETIME;wxUSE_EXTENDED_RTTI...
Checking static_code.cpp: wxUSE_DATETIME;wxUSE_INTL...
static_code.cpp:94:0: style: The function 'wxStringReadValue' is never used. [unusedFunction]
template<> void wxStringReadValue(const wxString &s , wxDateTime &data )
^
static_code.cpp:99:0: style: The function 'wxStringWriteValue' is never used. [unusedFunction]
template<> void wxStringWriteValue(wxString &s , const wxDateTime &data )
^
static_code.cpp:123:0: style: The function 'OnInit' is never used. [unusedFunction]
    virtual bool OnInit() override
^
static_code.cpp:130:0: style: The function 'OnExit' is never used. [unusedFunction]
    virtual void OnExit() override
^
static_code.cpp:2426:0: style: The function 'wxPrevMonth' is never used. [unusedFunction]
WXDLLIMPEXP_BASE void wxPrevMonth(wxDateTime::Month& m)
^
static_code.cpp:2434:0: style: The function 'wxNextWDay' is never used. [unusedFunction]
WXDLLIMPEXP_BASE void wxNextWDay(wxDateTime::WeekDay& wd)
^
static_code.cpp:2442:0: style: The function 'wxPrevWDay' is never used. [unusedFunction]
WXDLLIMPEXP_BASE void wxPrevWDay(wxDateTime::WeekDay& wd)
^
nofile:0:0: information: Active checkers: 161/592 (use --checkers-report=<filename> to see details) [checkersReport]
```