Dhirubhai Ambani
Institute of Information and Communication Technology

Lab07: Program Inspection, Debugging and Static Analysis

IT314 – Software Engineering

Jay Rathod

202201255

Code Link :

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

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

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

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

Formal Code Review and Static Analysis would be the most effective inspection categories for this code. Here's why:

- Formal Code Review:
  - This involves a detailed examination of the code by peers to catch logical errors and ensure the implementation meets the required standards. Given the complexity of date and time manipulations, having multiple sets of eyes on the logic can help uncover nuanced bugs and assumptions that one developer might miss.
- Static Analysis Tools:
  - Using tools that automatically analyze the code for potential errors (like SonarQube, ESLint, or Clang Static Analyzer) would also be beneficial. These tools can identify issues such as unreachable code, potential memory leaks, and adherence to coding standards.

3. Which type of error can you not identify using the program inspection?

Dynamic Errors are a significant category that might go unnoticed during program inspection:

- Runtime Errors:
  - These errors occur when the code is executed rather than when it is compiled. Examples include:
    - Incorrect handling of user input (e.g., invalid dates that lead to out-of-range calculations).
    - Logical errors that only become apparent with specific data inputs, such as unexpected behavior when calculating holidays for non-standard years (like those with adjusted leap year rules).
- Performance Issues:

    - While static analysis can identify some inefficiencies, it cannot truly gauge how the code performs under load. For example, if a user queries a date range spanning many years, the linear iteration over each day may lead to performance bottlenecks that are only identifiable during runtime.

4. Is the program inspection technique worth applying?

Yes, program inspection techniques are absolutely worth applying, and here's why:

- Early Detection of Bugs:
    - Regular inspections can catch bugs before they are deployed, which is much cheaper than fixing them post-deployment. Catching logical errors in date calculations early on can save significant troubleshooting time later.
- Improved Code Quality:
    - Inspections encourage adherence to best practices and coding standards, which enhance the overall quality and maintainability of the code. For example, using consistent naming conventions for functions and variables can make the codebase easier to navigate.
- Knowledge Sharing and Team Learning:
    - Code reviews foster a culture of collaboration and learning among team members. They allow developers to share insights about date-time calculations, potential pitfalls, and alternate approaches.
- Comprehensive Documentation:
    - Inspections often lead to better documentation practices, which is especially important for complex functionalities like date-time handling. This documentation helps future developers understand the reasoning behind certain implementation decisions and improves the maintainability of the codebase.

2. Code Debugging :

1.Armstrong number:

1. There is one error in the program related to the computation of the

remainder, as previously

identified.

2. To fix this error, one should set a breakpoint at the point where the

remainder is computed to ensure it’s calculated correctly. Step through the

code to observe the values of variables and expressions during execution.

3. The corrected executable code is as follows:

```
// Armstrong Number
class Armstrong {
public static void main(String args[]) {
int num = Integer.parseInt(args[0]);
int n = num; // used to check at the last time
int check = 0, remainder;
while (num > 0) {
remainder = num % 10;
check = check + (int) Math.pow(remainder, 3);
num = num / 10;
}
if (check == n)
System.out.println(n + " is an Armstrong Number");
else
System.out.println(n + " is not an Armstrong Number");
```

```
}
}
```

2. GCD and LCM:

```
//program to calculate the GCD and LCM of two given numbers
import java.util.Scanner; public
class GCD_LCM
{
    static int gcd(int x, int y)
    {
        int r=0, a, b;
        a = (x > y) ?
        y : x; // a is greater
        number b = (x < y)
        ? x : y; // b is
        smaller number r = b; while(a % b == 0) //Error replace 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 Identified:

- The GCD calculation uses the condition a % b == 0, which should be a % b != 0.
- The LCM calculation uses the condition if(a % x != 0 && a % y != 0), which should be if(a % x == 0 && a % y == 0).

2. Breakpoints Required:

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

Steps to Fix:

- Change while(a % b == 0) to while(a % b != 0) in the GCD calculation.
- Change if(a % x != 0 && a % y != 0) to if(a % x == 0 && a % y == 0) in the LCM calculation.

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();
    }
}
```

3. knapsack:

```
//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 Identified:

- In the knapsack logic, opt[n++][w] should be opt[n-1][w] to properly calculate the profit of not taking the item.
- In the condition if(weight[n] > w), the comparison is reversed. It should be if(weight[n] <= w) to ensure the item is taken if its weight fits.
- In the second option calculation, profit[n-2] should be profit[n] to properly reference the current item.

2. Breakpoints Required:

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

Steps to Fix:

- Change opt[n++][w] to opt[n-1][w].
- Change if(weight[n] > w) to if(weight[n] <= w).
- Change profit[n-2] to profit[n].

3. Corrected 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-1][w]; //

    Corrected


                // take item n
```

```
            int option2 = Integer.MIN_VALUE; if (weight[n] <= w) option2 = profit[n] +
          opt[n-1][w-weight[n]]; // Corrected
            // 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]);
      }
  }
}
```

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%1

               0

            }
```

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 Identified:

- In the inner while loop, the condition sum == 0 should be sum > 0 to perform the digit extraction correctly.
- The statement s=s*(sum/10) should be s = s + (sum % 10) to accumulate the sum of the digits instead of multiplying.
- The line sum = sum % 10 should end with a semicolon.

2. Breakpoints Required:

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

Steps to Fix:

- Change while(sum == 0) to while(sum > 0).
- Change s=s*(sum/10) to s = s + (sum % 10).
- Add a semicolon to sum = sum % 10.

3. Corrected Code:

```
import java.util.*;

public class MagicNumberCheck

{

    public static void main(String args[])

    {

       Scanner ob = new Scanner(System.in);

     System.out.println("Enter the number to be checked."); int n =

     ob.nextInt();

       int sum = 0, num = n; while(num

     > 9)

       {

          sum = num; int s = 0; while(sum

        > 0) //

          Corrected

          {
```

```
            s = s + (sum % 10); // Corrected
            sum = sum / 10; // Corrected
        }
        num = s;
    }
    if(num == 1)
    {
        System.out.println(n + " is a Magic Number.");
    }
    else
    {
        System.out.println(n + " is not a Magic Number.");
    }
  }
}
```

5. Merge Sort:

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

Input: before 14 32 67 76 23 41 58 85

after 14 23 32 41 58 67 76 85

1.Errors Identified:

- In mergeSort, the array slicing logic is incorrect:
    - int[] left = leftHalf(array + 1) and int[] right = rightHalf(array - 1) should be int[] left = leftHalf(array) and int[] right = rightHalf(array).
    - In merge, the function call merge(array, left++, right--) should be merge(array, left, right). The increment/decrement operators are not appropriate here.

2. Breakpoints Required:

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

Steps to Fix:

- Change array + 1 and array - 1 in mergeSort to just array in the respective calls.
- Remove the increment/decrement (++, --) from the merge(array, left++, right--) call.

3. Corrected Code:

```
import java.util.*;



public class MergeSort { public static void

   main(String[] args) { int[] list = {14, 32, 67, 76,

   23, 41, 58, 85};

        System.out.println("before: " + Arrays.toString(list)); mergeSort(list);


            System.out.println("after: " + Arrays.toString(list));

     }

     // Places the elements of the given array into sorted

   order public static void mergeSort(int[] array) { if

   (array.length > 1) { // split array into two halves int[] left
```

```
= leftHalf(array); // Corrected int[] right =
rightHalf(array); // Corrected // recursively sort the two
halves mergeSort(left); mergeSort(right);
          // merge the sorted halves into a sorted whole
        merge(array, left, right); // Corrected
      }
  }

  // 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.
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++;
         }
      }
```

```
    }
}
```

6. Multiply Matrices:

```
//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 Identified:

- In the innermost loop, incorrect array indices are used in the multiplication:

- first[c-1][c-k] should be first[c][k].
- second[k-1][k-d] should be second[k][d].

2. Breakpoints Required:

- One to fix the incorrect matrix indices.

Steps to Fix:

- Change first[c-1][c-k] to first[c][k].
- 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 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 += first[c][k] * second[k][d]; //
          Corrected indices
          }
          multiply[c][d] =
          sum; sum = 0;
      }
    }
    System.out.println("Product of entered matrices:-");
  for (c = 0; c < m; c++) {
```

```
for (d = 0; d < q; d++)
   System.out.print(multiply[c][d] +
   "\t");
```

```
            System.out.print("\n");
          }
        }
      }
}
```

7. Quadratic Probing:

```
/**
    * Java Program to implement Quadratic Probing Hash Table
*/ 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;
 }
 /** Function to check if hash table contains a key
**/ public boolean contains(String key) { return
  get(key) != null;
 }
 /** Function 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(ke
            y)) 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(ke
        y)) 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");
       /** make 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');
    }
}
```

1.Errors Identified:

- Index Calculation Issues: The index calculations in the insert, get, and remove methods are incorrect, which can lead to out-of-bounds errors or incorrect data retrieval.
- Array Clearing in makeEmpty: The makeEmpty method does not effectively clear the arrays, as it only resets the size but does not set the array elements to null.

2. Breakpoints Required:

- Total Breakpoints: 4
    - 0 Breakpoint 1: In insert, to check the index calculation logic.
    - ○ Breakpoint 2: In get, to validate the key retrieval process.
    - ○ Breakpoint 3: In remove, to ensure proper key removal.
    - ○ Breakpoint 4: In makeEmpty, to confirm that all elements are cleared.

Steps to Fix:

- Update Index Calculations:

0 In the insert method, change the index calculation to ensure proper quadratic probing: use (tmp + h * h) % maxSize instead of the incorrect calculations.

○ Similarly, in the get and remove methods, use the updated index calculation.

- Correctly Clear Arrays in makeEmpty: Iterate through the arrays in makeEmpty to set each element to null before resetting the size to 0.

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');
      }
  }
```

8. Array Sorting:

```
  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]);
    }
}
```

1.Errors Identified:

- Redundant Sorting: The sorting algorithm used is inefficient for larger arrays since it uses a nested loop with a time complexity of $O(n^2)$. While not a syntactical error, this can lead to performance issues.
- Missing Input Validation: There's no validation for user input, which can lead to unexpected behavior if the user enters a non-integer value.

2. Breakpoints Required:

- Total Breakpoints: 2
    - Breakpoint 1: After reading the number of elements, to check if n is valid.
    - Breakpoint 2: After sorting to inspect the contents of the array.

Steps to Fix:

- Improve Sorting Algorithm: Replace the nested loop sorting with a more efficient sorting method, such as Arrays.sort(), which has a time complexity of O(n log n).
- Add Input Validation: Implement a check to ensure that the user input for the number of elements is valid and that the elements entered are integers.

Corrected Code:

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


public class Ascending_Order { public
   static void main(String[] args) { int n;
        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("E
      nter 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]);
    }
}
```

9. Stack Implementation:

```
  //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()
        ;
    }
  }
```

1.Errors Identified:

- Incorrect Push Logic: The top index is decremented before assigning the value, which results in an ArrayIndexOutOfBoundsException when the stack is not empty. It should increment for a push.
- Incorrect Pop Logic: The top index is incremented incorrectly, leading to an ArrayIndexOutOfBoundsException when popping the last item.
- Display Logic Error: The display loop uses i > top instead of i <= top, which means it never prints any elements from the stack.

2. Breakpoints Required:

- Total Breakpoints: 3
    - Breakpoint 1: After the push method to verify that elements are being pushed correctly.
    - Breakpoint 2: After the pop method to check the stack's state post-pop.
    - Breakpoint 3: Before the display method to ensure the stack contents are as expected.

Steps to Fix:

- Fix Push Method: Change the logic in the push method to top++ before assigning the value.
- Fix Pop Method: Adjust the pop method to decrement top only if it is not empty.
- Fix Display Method: Change the loop condition in the display method to i <= top for proper printing of stack elements.

Corrected 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(
        );
    }
}
```

10. Tower Of Hanoi:

```
//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 - 1, inter, from, to);
        }
}
}
```

1.Errors Identified:

- Base Case Missing for topN Greater than 1: In the doTowers method, when topN is 1, it correctly prints the move; however, there's no return statement, which might cause further calls to execute incorrectly when the recursion unwinds.
- Printing Disk Number: The print statement assumes that the disks are numbered starting from 1. If more than 9 disks are added, the output may not format correctly without leading zeros.

2. Breakpoints Required:

- Total Breakpoints: 2
    - Breakpoint 1: After the first doTowers call to check if the function is recursively processing correctly.
    - Breakpoint 2: Before printing the move to ensure that the disk number and rods are as expected.

Steps to Fix:

- Add a Return Statement: After printing the move for a single disk to ensure that no further processing occurs.
- Ensure Formatting: Consider modifying the print statement to format disk numbers properly if the number of disks exceeds 9.

Corrected 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);
          return; // Added return statement
       } else { doTowers(topN - 1, from, to,
          inter);
            System.out.println("Disk " + topN + " from " + from + " to " + to); doTowers(topN -
          1, inter, from, to);
         }
      }
  }
```

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.Submit your results in the .xls or .jpg format only.

```
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]
```