Lab 7 Software Engineering

Dev Dodiya

20th October, 2024

Code 1: Armstrong

Program Inspection:

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

- There are two main errors in the code:
 - The calculation of remainder uses integer division, remainder = num / 10;, which gives incorrect results. This line should use modulus (%) to obtain the last digit of the number, i.e., remainder = num % 10;.
 - The calculation of the new value of num in num = num % 10; is incorrect. It should be num = num / 10; to reduce the number by removing the last digit.

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

 Code walkthrough or peer review would be more effective for this type of error identification. These methods allow one or more individuals to thoroughly examine the code logic and arithmetic operations used.

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

 Run-time errors or logical issues not easily visible through inspection might be missed. For example, if there were overflow issues or edge case handling problems, these might not be immediately apparent until the program is executed.

4. Is the program inspection technique worth applying?

 Yes, program inspection can be a useful early-phase debugging technique to catch common syntactic and logical errors before runtime. However, it should be complemented by other testing methods like unit testing and dynamic debugging.

Debugging:

- 1. How many errors are there in the program? Mention the errors you have identified.
 - There are two errors in the program:
 - remainder = num / 10; should be remainder = num % 10;.
 - num = num % 10; should be num = num / 10;.
- 2. How many breakpoints do you need to fix those errors?

Two breakpoints would be enough to identify and fix both issues:

- Set a breakpoint before the line where remainder is calculated to inspect how it's dividing instead of taking the modulus.
- Set another breakpoint before the line where num is being updated to check how it should be reduced.

- Step 1: Change the line remainder = num / 10; to remainder = num %
 10; to get the last digit of the number correctly.
- Step 2: Modify the line num = num % 10; to num = num / 10; to correctly remove the last digit of the number after extracting the remainder.
- 3. Submit your complete executable code:

```
// Corrected Armstrong Number Program

class Armstrong {
   public static void main(String args[]) {
    int num = Integer.parseInt(args[0]); // input number
   int n = num; // store the original number for comparison
```

```
int check = 0, remainder;
   while (num > 0) {
      remainder = num % 10; // get the last digit
      check = check + (int)Math.pow(remainder, 3); // cube the digit and
add to sum
     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");
```

Code 2:- GCD and LCM

Program Inspection:

- 1. How many errors are there in the program? Mention the errors you have identified.
 - There are two main errors in the code:
 - In the gcd method, the while loop condition is incorrect. It should be while(a % b != 0) instead of while(a % b == 0) because we want to continue until the remainder is zero.
 - In the lcm method, the logic for finding LCM is incorrect. The condition if(a % x != 0 && a % y != 0) should be if(a % x == 0 && a % y == 0) because LCM is the smallest number that is divisible by both x and y.

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

- Code walkthrough would be effective for identifying these types of errors. Reviewing the logic and conditions carefully can help pinpoint where the calculations and logic deviate from expectations.
- 3. Which type of error are you not able to identify using the program inspection?
 - Edge cases such as when one of the numbers is zero or when both numbers are the same might not be fully covered using program inspection. These cases would be better caught with test cases and dynamic analysis.

4. Is the program inspection technique worth applying?

 Yes, program inspection is helpful in catching syntactical and logical errors early, especially simple conditional mistakes like the ones in this program. However, it should be supplemented by actual code execution and testing.

Debugging:

- 1. How many errors are there in the program? Mention the errors you have identified.
 - Two errors:
 - The while loop condition in gcd should be while(a % b != 0) instead of while(a % b == 0).
 - The condition inside the if statement in lcm should be if(a % x == 0 && a % y == 0) instead of if(a % x != 0 && a % y != 0).

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

Two breakpoints would be sufficient to fix the errors:

- One before the while loop in the gcd method to examine the condition.
- One before the if statement in the lcm method to check the condition for divisibility of a by both x and y.

Steps to fix the errors:

Step 1: Change while(a % b == 0) to while(a % b != 0) in the gcd method.

Step 2: Modify if(a % x = 0 && a % y = 0) to if(a % x = 0 && a % y = 0) in the lcm method.

```
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 smaller number
  b = (x < y)? x : y; // b is greater number
  r = b;
  while(a % b != 0) // Fix: change to != 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)
  {
    // Fix: change to == to check divisibility
    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();
```

}
}

Code 3:- Knaopsack

- 1. How many errors are there in the program? Mention the errors you have identified.
 - Error 1: In the line int option1 = opt[n++][w];, the use of n++ is incorrect because it increments n during the iteration, leading to skipping the next item. This should be opt[n-1][w] to refer to the correct value.
 - Error 2: In the condition if (weight[n] > w), this should be if (weight[n] <= w) because we only want to take the item if its weight fits within the current weight limit w.
 - Error 3: In the calculation option2 = profit[n-2] +
 opt[n-1][w-weight[n]];, the profit term uses n-2 instead of n. This will
 lead to incorrect profit values. It should be option2 = profit[n] +
 opt[n-1][w-weight[n]];.
- 2. Which category of program inspection would you find more effective?
 - Code walkthrough would be effective for catching these logical errors.
 Systematically reviewing how indices are incremented and how conditions are structured will reveal these issues.
- 3. Which type of error are you not able to identify using the program inspection?
 - Performance issues and potential edge cases, such as when the weights are zero or when the knapsack is empty, might not be

immediately obvious during inspection. These would be best identified through testing or profiling.

4. Is the program inspection technique worth applying?

 Yes, program inspection can be helpful for identifying logical flaws, especially around index management and conditions in loops.
 However, dynamic testing and runtime checks are still necessary to ensure correctness under different input conditions.

Debugging:

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

- Error 1: int option1 = opt[n++][w]; should be int option1 = opt[n-1][w].
- Error 2: if (weight[n] > w) should be if (weight[n] <= w).
- Error 3: option2 = profit[n-2] + opt[n-1][w-weight[n]]; should be
 option2 = profit[n] + opt[n-1][w-weight[n]];.

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

Three breakpoints would be sufficient:

- One before the line int option1 = opt[n++][w]; to check if n++ is causing unintended skips.
- One before if (weight[n] > w) to verify the weight condition logic.
- One before the line option2 = profit[n-2] + opt[n-1][w-weight[n]]; to inspect the correct profit and weight values.

- Step 1: Change int option1 = opt[n++][w]; to int option1 = opt[n-1][w]; to correctly handle item indexing.
- Step 2: Modify if (weight[n] > w) to if (weight[n] <= w) to ensure the item is taken when it fits within the current weight.

• Step 3: Change option2 = profit[n-2] + opt[n-1][w-weight[n]]; to option2 = profit[n] + opt[n-1][w-weight[n]]; to use the correct profit value for the current item.

```
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 \le 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 \le N; n++) {
      for (int w = 1; w \le W; w++) {
        // don't take item n
        int option1 = opt[n-1][w]; // Fix: change n++ to n-1
        // take item n
        int option2 = Integer.MIN_VALUE;
        if (weight[n] <= w) // Fix: use <= to check if weight fits
```

```
option2 = profit[n] + opt[n-1][w-weight[n]]; // Fix: correct profit
index to profit[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 \le N; n++) {
      System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" +
take[n]);
```

Code 4: Magic Number

Program Inspection:

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

- Error 1: In the inner while loop, the condition while(sum == 0) is incorrect. It should be while(sum > 0) to continue the loop while there are digits to sum.
- Error 2: The statement s = s * (sum / 10); is incorrect. The correct operation should be s = s + (sum % 10); because we are summing the digits, not multiplying them.
- Error 3: The line sum = sum % 10 is missing a semicolon (;), leading to a syntax error.

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

 Code walkthrough would be more effective for identifying these logical and syntactical errors. Reviewing each part of the loop structure and checking how digits are summed would catch these issues.

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

• Edge cases like negative numbers or very large inputs might not be caught during inspection. These would be better identified using testing with a variety of inputs.

4. Is the program inspection technique worth applying?

 Yes, program inspection helps identify syntax and logic errors early, especially in loops and conditions. However, it should be supplemented by testing with different input cases to ensure full correctness.

Debugging:

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

- Error 1: The inner while(sum == 0) condition should be while(sum > 0) to correctly iterate through the digits.
- Error 2: The statement s = s * (sum / 10); should be s = s + (sum % 10); to correctly sum the digits.
- Error 3: The statement sum = sum % 10 is missing a semicolon (;).

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

Three breakpoints are needed:

- One before the inner while loop to check the condition.
- One before the digit summing line to ensure the correct operation is performed.
- One after the semicolon issue to ensure the loop works properly.

Steps to fix the errors:

- Step 1: Change while(sum == 0) to while(sum > 0) to correctly iterate through digits.
- Step 2: Change s = s * (sum / 10); to s = s + (sum % 10); to properly sum the digits.
- Step 3: Add the missing semicolon after sum = sum % 10.

```
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;
}
```

```
// Loop until we reduce the number to a single digit
while (num > 9) {
  sum = num;
  int s = 0;
  // Fix: Change while condition and summing logic
  while (sum > 0) {
    s = s + (sum % 10); // Fix: Sum the digits, not multiply
    sum = sum / 10; // Fix: Divide to get the next digit
  }
  num = s; // Assign the sum of digits back to num for the next loop
}
// Check if the resulting single digit is 1 (Magic number condition)
if (num == 1) {
  System.out.println(n + " is a Magic Number.");
```

```
} else {
    System.out.println(n + " is not a Magic Number.");
}
ob.close();
}
```

Code 5: Merge Sort

- 1. How many errors are there in the program? Mention the errors you have identified.
 - Error 1: The leftHalf and rightHalf methods are passed incorrect parameters in mergeSort. In the line int[] left = leftHalf(array+1); and int[] right = rightHalf(array-1);, the operations array+1 and array-1 are invalid for arrays. They should just pass the array as leftHalf(array) and rightHalf(array).
 - Error 2: In the merge method call within mergeSort, the increment and decrement operations (left++ and right--) are incorrect. Arrays cannot be modified this way. The correct call should be merge(array, left, right);
- 2. Which category of program inspection would you find more effective?

 Code walkthrough would be effective here, as it helps to check each function and how they interact with each other, particularly how arrays are passed and modified in the mergeSort and merge methods.

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

 Performance errors or errors related to handling large input sizes cannot be easily identified using program inspection alone. These would require runtime testing with large datasets to evaluate efficiency and scalability.

4. Is the program inspection technique worth applying?

 Yes, program inspection is useful to catch logical errors and improper parameter handling. However, thorough testing is necessary to ensure correctness across various cases.

Debugging:

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

- Error 1: Incorrect array handling in the calls to leftHalf(array+1) and rightHalf(array-1). Arrays should be passed directly as leftHalf(array) and rightHalf(array).
- Error 2: Incorrect use of increment and decrement (left++, right--) in the merge call. Arrays should be passed directly as merge(array, left, right).

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

Two breakpoints are needed:

 One before the mergeSort method call to check if arrays are passed correctly. One before the merge call to ensure correct merging without modifying array references incorrectly.

- Step 1: Correct the mergeSort calls to pass leftHalf(array) and rightHalf(array) without modifying the array with +1 or -1.
- Step 2: Remove the incorrect increment and decrement operations in the merge method and pass the arrays left and right as they are.
- 3. Submit your complete executable code:

```
import java.util.*;
public class MergeSort {
 public static void main(String[] args) {
   int[] list = {14, 32, 67, 76, 23, 41, 58, 85};
    System.out.println("before: " + Arrays.toString(list));
   mergeSort(list);
    System.out.println("after: " + Arrays.toString(list));
 }
 // 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); // Fix: Pass array without +1
    int[] right = rightHalf(array); // Fix: Pass array without -1
    // recursively sort the two halves
    mergeSort(left);
    mergeSort(right);
    // merge the sorted halves into a sorted whole
    merge(array, left, right); // Fix: Pass arrays without ++/--
```

```
// 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 < size 2; 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++;
}
}
```

Code 6: Multiply Matrics

- 1. How many errors are there in the program? Mention the errors you have identified.
 - Error 1: In the nested loop responsible for matrix multiplication, the indices used in the line sum = sum + first[c-1][c-k]*second[k-1][k-d]; are incorrect. The indices c-1, c-k, k-1, and k-d do not correctly reference the matrix elements. The correct indices should be first[c][k] * second[k][d].
 - Error 2: The input prompt messages for the second matrix are mistakenly asking for the "rows and columns of the first matrix" again instead of the second matrix.
- 2. Which category of program inspection would you find more effective?

• Code review is effective here, where errors in array indexing and logic can be caught by carefully reviewing the matrix multiplication logic.

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

 Boundary errors or errors occurring due to matrix sizes and edge cases (like 1x1 or very large matrices) may not be identified until runtime testing.

4. Is the program inspection technique worth applying?

• Yes, it is valuable for identifying logical errors in the nested loops and incorrect indexing, which are crucial for matrix operations.

Debugging:

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

- Error 1: Incorrect indices used in the matrix multiplication logic: sum
 = sum + first[c-1][c-k]*second[k-1][k-d];. This causes an
 ArrayIndexOutOfBoundsException.
- Error 2: The input prompt message is wrong for the second matrix: "Enter the number of rows and columns of the second matrix" should be used, not the first matrix.

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

Two breakpoints:

- One before the nested loops start to inspect the indices and ensure the correct matrix elements are being accessed.
- One to check the input message for the second matrix dimensions.

Steps to fix the errors:

Step 1: Fix the incorrect indices in the matrix multiplication logic.
 Change sum = sum + first[c-1][c-k]*second[k-1][k-d]; to sum = sum + first[c][k] * second[k][d];.

- Step 2: Correct the input prompt for the second matrix by updating it to "Enter the number of rows and columns of the second matrix."
- 3. Submit your complete executable code:

```
import java.util.Scanner;
class MatrixMultiplication {
 public static void main(String args[]) {
    int m, n, p, q, sum = 0, c, d, k;
    Scanner in = new Scanner(System.in);
    System.out.println("Enter the number of rows and columns of the first
matrix");
    m = in.nextInt();
    n = in.nextInt();
    int first[][] = new int[m][n];
```

```
System.out.println("Enter the elements of the first matrix");
    for (c = 0; c < m; c++) {
      for (d = 0; d < n; d++) {
        first[c][d] = in.nextInt();
      }
    System.out.println("Enter the number of rows and columns of the second
matrix");
    p = in.nextInt();
    q = in.nextInt();
    if (n != p) {
      System.out.println("Matrices with entered orders can't be multiplied with
each other.");
    } else {
```

```
int second[][] = new int[p][q];
int multiply[][] = new int[m][q];
System.out.println("Enter the elements of the second matrix");
for (c = 0; c < p; c++) {
  for (d = 0; d < q; d++) {
    second[c][d] = in.nextInt();
for (c = 0; c < m; c++) {
  for (d = 0; d < q; d++) {
    for (k = 0; k < n; k++) {
       sum = sum + first[c][k] * second[k][d]; // Fix: Corrected the 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");
```

Code 7: Quadratic Probing:

- 1. How many errors are there in the program? Mention the errors you have identified.
 - Error 1: In the insert method, the line i + = (i + h / h--) % maxSize; has incorrect syntax. It should be i = (i + h * h++) % maxSize; to properly handle the quadratic probing calculation.
 - Error 2: In the remove method, the rehashing logic decreases the currentSize twice. First during the deletion, and second while re-inserting the keys. This results in incorrect size management.
 - Error 3: There's no initial handling for cases where the hash code might be negative, which can lead to array index out-of-bounds exceptions when performing modulus operations.
 - Error 4: The test inputs suggest entering multiple key-value pairs one after the other, but the current implementation allows only one pair to be inserted per iteration of the loop.
- 2. Which category of program inspection would you find more effective?
 - Static code analysis is effective in identifying syntax errors, logical errors, and potential edge cases in code (such as handling hash codes and rehashing).
- 3. Which type of error are you not able to identify using the program inspection?
 - Runtime errors such as hash collisions or handling large inputs may not be immediately identified through static inspection. These errors often surface only during testing or execution.
- 4. Is the program inspection technique worth applying?

 Yes, it helps catch both syntax errors and logical flaws that are hard to detect just by executing the program without thorough testing.

Debugging:

- 1. How many errors are there in the program? Mention the errors you have identified.
 - Error 1: Syntax error in the insert method on the line i + = (i + h / h--)
 % maxSize;. This needs to be corrected to i = (i + h * h++) % maxSize;.
 - Error 2: Incorrect rehashing and double decrementing of currentSize in the remove method, which needs adjustment.
 - Error 3: No handling of negative hash codes, leading to array out-of-bounds errors.
 - Error 4: Incomplete test input as the program expects one key-value pair per input, but the prompt suggests multiple key-value pairs.
- 2. How many breakpoints do you need to fix those errors? Three breakpoints:
 - One at the insert method to fix the probing calculation.
 - One at the remove method to handle correct size decrement.
 - One at the hash code calculation to handle negative values.

- Step 1: Fix the probing calculation in the insert method by correcting i
 + = (i + h / h--) % maxSize; to i = (i + h * h++) % maxSize;.
- Step 2: Adjust the remove method to avoid double decrementing the currentSize by only decrementing it once after rehashing.
- Step 3: Handle negative hash codes by modifying the hash method to return Math.abs(key.hashCode() % maxSize);.
- Step 4: Modify test inputs in the loop to prompt for one key-value pair at a time.

```
import java.util.Scanner;
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 Math.abs(key.hashCode() % maxSize); // Fix for handling
negative hash codes
 }
 /** Function to insert key-value pair **/
 public void insert(String key, String val) {
```

```
if (isFull()) {
  System.out.println("Hash Table is full!");
  return;
int tmp = hash(key);
int i = tmp, h = 1;
while (keys[i] != null) {
  if (keys[i].equals(key)) {
     vals[i] = val; // Replace existing value
    return;
  i = (i + h * h++) % maxSize; // Fix: Correct probing
keys[i] = key;
```

```
vals[i] = val;
  currentSize++;
/** 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;
  }
  return null;
/** Function to remove key and its value **/
```

```
public void remove(String key) {
  if (!contains(key)) {
    System.out.println("Key not found!");
    return;
  int i = hash(key), h = 1;
  while (!key.equals(keys[i])) {
    i = (i + h * h++) \% maxSize;
  keys[i] = vals[i] = null;
  currentSize--;
  // Rehash all keys
  for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) % maxSize)
```

```
String tmpKey = keys[i], tmpVal = vals[i];
    keys[i] = vals[i] = null;
    currentSize--;
    insert(tmpKey, tmpVal);
/** 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:");
   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;
  }
  qpht.printHashTable();
  System.out.println("\nDo you want to continue (Type y or n) \n");
  ch = scan.next().charAt(0);
} while (ch == 'Y' | | ch == 'y');
```

}

Code 8: Sorting Array

Program Inspection:

- 1. How many errors are there in the program? Mention the errors you have identified.
 - Error 1: The class name is Ascending _Order with an underscore, which is not a valid identifier in Java. Class names should not have spaces or special characters like underscores.
 - Error 2: The for-loop for sorting has an incorrect condition: for (int i = 0; i >= n; i++);. The loop condition i >= n will prevent the loop from executing since it's checking if i is greater than or equal to n. It should be i < n.
 - Error 3: The loop header for (int i = 0; i >= n; i++); has a semicolon at the end, which will terminate the loop immediately, making it ineffective.
 - Error 4: The condition inside the nested loop is incorrect for sorting in ascending order. The condition if (a[i] <= a[j]) should be if (a[i] > a[j]) to correctly swap the elements when they are out of order.
 - Error 5: The array is printed incorrectly at the end; in the final loop, it should iterate until i < n to print all elements correctly.

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

• Static code inspection is most effective here. This would catch syntactic errors like the improper class name, misplaced semicolon in the loop, and incorrect logic for sorting.

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

 Runtime errors or edge cases such as entering negative values or large input sizes might not be caught during static inspection. These would surface during actual execution and testing.

4. Is the program inspection technique worth applying?

 Yes, program inspection is worth applying. It helps identify logical and syntactical mistakes before running the program and can save significant debugging time.

Debugging:

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

- Error 1: The class name Ascending Order has an invalid identifier.
- Error 2: The incorrect for-loop condition i >= n.
- Error 3: The misplaced semicolon in the sorting loop for (int i = 0; i >= n; i++);.
- Error 4: The incorrect comparison for swapping elements if (a[i] <= a[j]) should be if (a[i] > a[j]).
- Error 5: The loop to print the array stops too early (i < n 1).

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

Five breakpoints:

- One at the class declaration to correct the class name.
- One at the outer for-loop to correct the condition and remove the misplaced semicolon.
- One at the inner for-loop to adjust the comparison for sorting.
- One for printing the array, correcting the loop condition to iterate over all elements.

Steps to fix the errors:

- Step 1: Rename the class from Ascending Order to AscendingOrder.
- Step 2: Fix the outer loop condition from i >= n to i < n and remove the semicolon after the loop.
- Step 3: Fix the comparison inside the inner loop from if (a[i] <= a[j]) to
 if (a[i] > a[j]) for proper sorting.
- Step 4: Correct the print loop to iterate over all elements (i < n).

3. Submit your complete executable code:

```
import java.util.Scanner;
public class AscendingOrder {
 public static void main(String[] args) {
    int n, temp;
    Scanner s = new Scanner(System.in);
    System.out.print("Enter 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();
   // Sorting the array in ascending order
   for (int i = 0; i < n; i++) { // Corrected loop condition and removed
semicolon
      for (int j = i + 1; j < n; j++) {
        if (a[i] > a[j]) { // Fixed comparison for ascending order
           temp = a[i];
           a[i] = a[j];
           a[j] = temp;
   }
   // Printing the sorted array
   System.out.print("Ascending Order: ");
```

```
for (int i = 0; i < n - 1; i++) {
        System.out.print(a[i] + ", ");
}
System.out.print(a[n - 1]); // Print last element without trailing comma
}
</pre>
```

Code 9: Stack Implementation

Program Inspection:

- 1. How many errors are there in the program? Mention the errors you have identified.
 - Error 1: In the push() method, the operation top-- should be top++.
 The current implementation decreases the top index instead of increasing it when a new value is pushed onto the stack.
 - Error 2: In the display() method, the for-loop condition i > top should be i <= top in order to display all the elements in the stack correctly.
 The current loop doesn't print the elements properly.
 - Error 3: The array indexing for the display() loop also starts from i = 0, which is correct, but the condition and logic of comparing top is wrong.

• Error 4: In the pop() method, the operation top++ is wrong for popping an element. It should be top-- to reduce the index after removing the element.

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

• Static code inspection would be the most effective for identifying syntactical and logical errors in this program, especially regarding the management of the top index and loop conditions.

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

 Runtime errors such as stack overflow, underflow, or mismanagement of array bounds might not surface immediately during inspection without test cases. These errors could show up during execution and would be harder to identify using only inspection.

4. Is the program inspection technique worth applying?

 Yes, it is worth applying. It can catch structural and logical issues related to stack operations before execution, improving the reliability of the code.

Debugging:

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

- Error 1: top-- in the push() method should be top++ to correctly increase the index when an element is added.
- Error 2: In the pop() method, the operation top++ should be top-- to reduce the index when removing an element.
- Error 3: The display() method's loop should iterate with i <= top, and it should print all elements in the stack correctly.
- Error 4: In the display() loop, it should print elements as stack[i] with proper boundaries and conditions.

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

Four breakpoints:

- One in the push() method to correct top-- to top++.
- One in the pop() method to correct top++ to top--.
- One in the display() method to fix the loop condition (i <= top).
- One in the display() loop to print elements in the correct order.

Steps to fix the errors:

- Step 1: Correct the logic in push() to increment top instead of decrementing.
- Step 2: Fix the pop() method to decrement top correctly.
- Step 3: Adjust the display() loop condition to iterate over the valid range of the stack.
- Step 4: Ensure the array elements are printed correctly in the display() method.

3. Submit your complete executable code:

```
import java.util.Scanner;

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: Incrementing top to add new value
    stack[top] = value;
public void pop() {
  if (!isEmpty()) {
```

```
top--; // Corrected: Decrementing top to remove value
  } 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 to display elements
properly
     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.display();

}
```

Code 10: Tower of Hanoi

Program Inspection:

- 1. How many errors are there in the program? Mention the errors you have identified.
 - Error 1: In the recursive call doTowers(topN ++, inter--, from+1, to+1), the increment (++) and decrement (--) operators on the method parameters are incorrect. The parameters should remain unchanged throughout the recursive calls. The correct call should be doTowers(topN - 1, inter, from, to).
 - Error 2: The characters (char) from+1, to+1, and inter-- are incorrect, as arithmetic operations on characters are not meaningful in this

- context. The characters represent the rods, so no such operations should be applied.
- Error 3: Missing semicolon after the doTowers recursive call in the else block: doTowers(topN ++, inter--, from+1, to+1).

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

 Static code inspection is useful for catching syntax errors (like the missing semicolon) and logical errors, such as the incorrect use of increment and arithmetic operations on characters.

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

 Logical runtime errors related to stack overflow or incorrect recursion depth would not be easily identified through static inspection without actual testing or tracing through execution.

4. Is the program inspection technique worth applying?

• Yes, it can catch errors early in the code review phase, especially for simple syntactical mistakes and logical errors in recursive functions like this one.

Debugging:

- 1. How many errors are there in the program? Mention the errors you have identified.
 - Error 1: Incorrect increment (++) and decrement (--) of parameters in recursive call.
 - Error 2: Invalid character arithmetic (from+1, to+1, inter--).
 - Error 3: Missing semicolon in the recursive call.
- 2. How many breakpoints do you need to fix those errors? Three breakpoints:

- One to fix the recursive call by removing the incorrect arithmetic on parameters.
- One to remove the increment and decrement operations on the parameters.
- One to add the missing semicolon.

Steps to fix the errors:

- Step 1: Remove ++ and -- from doTowers(topN ++, inter--, from+1, to+1) and replace it with the correct parameters for recursion.
- Step 2: Remove invalid arithmetic on from+1, to+1, and inter--.
- Step 3: Add the missing semicolon after the recursive method call.
- 3. Submit your complete executable code:

```
public class MainClass {

public static void main(String[] args) {

int nDisks = 3;

doTowers(nDisks, 'A', 'B', 'C');
}

public static void doTowers(int topN, char from, char inter, char to) {

if (topN == 1) {

System.out.println("Disk 1 from " + from + " to " + to);
}
```

```
} else {
 // Move top n-1 disks from 'from' to 'inter' using 'to' as auxiliary
 doTowers(topN - 1, from, to, inter);
 // Move nth disk from 'from' to 'to'
 System.out.println("Disk" + topN + "from" + from + "to" + to);
 // Move top n-1 disks from 'inter' to 'to' using 'from' as auxiliary
 doTowers(topN - 1, inter, from, to); // Corrected recursion
```

Static Analysis Tools:

