Name: Ansh Pandya
Student Id: 202201303

Program Inspection:

Code file:
https://drive.google.com/file/d/1yEdkZdBoZoF6Aii346IukeYu_UBQBdw3/view?usp=drivesdk

1. Errors Identified in the Program:

Category A: Data Reference Errors

- **Uninitialized Variables**: The program initializes necessary variables for file paths and regexes, but there is no explicit check for cases where file_path might not exist or be initialized correctly, which can cause file handling errors.
- **Array Subscripts**: No arrays are directly used, but indexing through the regex does not involve subscript errors.
- **Pointer/Reference Issues**: The program does not use pointers directly, but it references file paths, and there is no clear check for their existence before operations like renaming or opening files.

Category B: Data-Declaration Errors

- **Explicit Declarations**: Variables like file_path, new_file_path, and last_string are declared and used properly. However, no explicit error handling occurs if file paths are invalid.
- **Default Attributes**: In Python, data types are implicit, and no unexpected defaults seem to occur.
- **Variable Initialization**: Variables such as file_path and new_file_path are initialized based on operations, but there is no check for empty values if the input settings are malformed.

Category C: Computation Errors

- **Inconsistent Data Types**: No computation errors are present. All file path manipulations and regex operations are string-based and handled correctly.
- **Overflow/Underflow**: No risk of such errors, as the code does not perform numeric computations.

Category D: Comparison Errors

- **Mixed Comparisons**: No comparison errors are found, as most operations involve simple string checks or file manipulations.
- **Boolean Logic**: Logical operators like if and elif are used properly, with no apparent mistakes in logic.

Category E: Control-Flow Errors

- **Loop Termination**: The loops appear to terminate correctly. For example, in the file processing loop, each line is handled properly.
- **Off-by-One**: No evidence of off-by-one errors in the loop iterations.
- **Code Block Grouping**: The grouping of code blocks is managed through indentation, which Python enforces, so there are no mismatches.

Category F: Interface Errors

- **Parameter Matching**: The file does not deal with parameters being passed between different modules, so this category does not apply.

Category G: Input/Output Errors

- **File Handling**: There are potential issues with file handling if the paths provided are invalid, such as when opening or renaming files. Error handling for non-existent files (e.g., missing checks before rename()) should be added.
- **I/O Errors**: There is no handling for exceptions related to I/O operations such as file not found, permission denied, etc.

Category H: Other Checks

- **Warnings**: If this code were to be compiled or run, it might produce warnings related to file handling errors, especially regarding non-existent or invalid file paths.
- **Robustness**: The program does not include explicit checks to ensure that input file paths or regex operations are valid before execution.

Code Debugging:

1) Armstrong number:

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

The program has several issues:

- **Logic Issue in Division and Modulo:**
  - **remainder = num / 10;** should be **remainder = num % 10;** to get the last digit of the number.
  - **num = num % 10;** should be **num = num / 10;** to remove the last digit of the number after processing.
- **Incorrect Power Calculation:** The Armstrong number check should raise each digit to the power of the number of digits in the number (in this case, 3 for 3-digit numbers). So, the logic should be applied to the digits of the number, not directly the remainder.
- **Edge Case Handling:** The code assumes that the input will always be a valid integer. It should include error handling for cases where no argument is passed or if the input is not an integer.

b) How many breakpoints you need to fix those errors? What are the steps you have taken to fix the error you identified in the code fragment?

You would need breakpoints in two main locations:

1. Before the while loop to check if num and remainder are being calculated correctly.
2. Inside the loop to verify that the Armstrong number condition is being checked properly.

a. Steps Taken to Fix the Errors:

- I changed remainder = num / 10; to remainder = num % 10; to correctly extract the last digit of the number.
- I changed num = num % 10; to num = num / 10; to reduce the number after processing each digit.
- The power function is used correctly, but it should be based on the remainder (i.e., the digits of the number), not the quotient.
- Finally, I added error handling for invalid input.

c) Submit your complete executable code?

```
class Armstrong {

    public static void main(String args[]) {

        // Error Handling for missing argument

        if (args.length == 0) {

            System.out.println("Please provide a number as an argument.");

            return;

        }

        // Input validation

        try {

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

            int n = num; // used to check at the end
            int check = 0, remainder;

            while (num > 0) {

                remainder = num % 10;  // Extract last digit

                check = check + (int) Math.pow(remainder, 3);  // Add cube of the digit

                num = num / 10;  // Remove the last digit

            }

            // Checking if the sum of cubes is equal to the original number

            if (check == n)

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

            else

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

        } catch (NumberFormatException e) {

            System.out.println("Please enter a valid integer.");

        }

    }

}
```

2) GCD and LCM:

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

Incorrect Comparison in GCD Function: There is a logic error in the gcd method:

- **while(a % b == 0)** is incorrect; it should be **while(a % b != 0)**, as the Euclidean algorithm continues until a % b becomes zero.
- In the gcd function, the variables a and b are reversed when assigning them based on the size of x and y. **a** should be the larger number and **b** the smaller number to follow the Euclidean algorithm.

Incorrect Condition in LCM Function: The condition in the LCM function should use the logical AND (&&):

- **if(a % x != 0 && a % y != 0)** should be **if(a % x == 0 && a % y == 0)**, as the least common multiple is found when both numbers divide a evenly.

Error Handling Missing: There is no error handling for invalid inputs, such as non-integer values

b) How many breakpoints you need to fix those errors? What are the steps you have taken to fix the error you identified in the code fragment?

You would need breakpoints in the following locations:

1. Before the gcd loop to ensure that the Euclidean algorithm logic is correct.
2. Inside the lcm loop to check whether the correct least common multiple is being calculated.

a. Steps Taken to Fix the Errors:

- Corrected the while loop condition in the gcd function to continue until a % b becomes zero.

- In the lcm function, changed the condition to check when both numbers divide a evenly.

c) Submit your complete executable code?

```
import java.util.Scanner;


public class GCD_LCM {

    // Method to calculate GCD using the Euclidean algorithm

    static int gcd(int x, int y) {

        int r = 0, a, b;

        a = (x > y) ? x : y; // a is the greater number

        b = (x < y) ? x : y; // b is the smaller number


        while (b != 0) {  // Continue until the remainder is zero

            r = a % b;

            a = b;

            b = r;

        }

        return a;

    }


    // Method to calculate LCM

    static int lcm(int x, int y) {
```

```
    int a = (x > y) ? x : y; // a starts from the greater number
    while (true) {
        if (a % x == 0 && a % y == 0)  // LCM is found when both divide a
            return a;
        ++a;
    }
}

public static void main(String args[]) {
    Scanner input = new Scanner(System.in);
    System.out.println("Enter the two numbers: ");

    // Error handling for input
    try {
        int x = input.nextInt();
        int y = input.nextInt();

        // Output the GCD and LCM
        System.out.println("The GCD of two numbers is: " + gcd(x, y));
        System.out.println("The LCM of two numbers is: " + lcm(x, y));
    } catch (Exception e) {
            System.out.println("Please enter valid integers.");
        } finally {
            input.close();
        }
    }
}
```

3) Knapsack:

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

There are several issues in the code:

1. **Increment Operator Error in option1**:
    - In the line int option1 = opt[n++][w];, the n++ is incorrect. It increments n when accessing opt[n][w], which leads to an off-by-one error in future iterations. This should be opt[n-1][w] to reference the previous item.
2. **Fix**: Replace opt[n++][w] with opt[n-1][w].
3. **Wrong Index in option2 Calculation**:
    - In the option2 calculation, the profit is accessed with profit[n-2], but it should be profit[n], as it refers to the current item.
4. **Fix**: Replace profit[n-2] with profit[n].
5. **Logic in Condition for Taking an Item**:
    - The condition if (weight[n] > w) should be if (weight[n] <= w) since you can only consider the current item if its weight is less than or equal to the remaining capacity (w).
6. **Fix**: Change if (weight[n] > w) to if (weight[n] <= w).

b) How many breakpoints you need to fix those errors? What are the steps you have taken to fix the error you identified in the code fragment?

Breakpoint 1 (Option1 Calculation):

- Set a breakpoint at the line int option1 = opt[n++][w]; to ensure the correct value is being used for n in each iteration.

Breakpoint 2 (Option2 and Decision Logic):

- Set a breakpoint where option2 is calculated and the decision to take an item is made (line opt[n][w] = Math.max(option1, option2);). This will allow you to check whether the values of option1, option2, and the item selection are working correctly.

Fix the Increment Operator:

- Change int option1 = opt[n++][w]; to int option1 = opt[n-1][w];. This fixes the issue of n being incorrectly incremented.

Correct the Profit Index in Option2:

- Change int option2 = profit[n-2] + opt[n-1][w-weight[n]]; to int option2 = profit[n] + opt[n-1][w-weight[n]]; to use the correct profit value.

Fix the Condition for Including the Item:

- Change if (weight[n] > w) to if (weight[n] <= w). This ensures that the item is only considered if its weight is less than or equal to the current capacity.

c) Submit your complete executable 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]; // fixed increment issue
                // take item n
                int option2 = Integer.MIN_VALUE;
                if (weight[n] <= w) // fixed the condition logic
                    option2 = profit[n] + opt[n-1][w-weight[n]]; // fixed the profit index
                // 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 Number:

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

There are **three main errors** in the code:

1. **Incorrect Condition in the Inner while Loop**:
   - ○ The condition while(sum == 0) is incorrect. It should be while(sum > 0) to iterate over the digits of the number properly.
2. **Incorrect Arithmetic Operation in the Inner Loop**:
   - ○ The line s = s * (sum / 10) is wrong. The goal is to **sum** the digits of the number, not multiply. It should be s = s + (sum % 10).
3. **Missing Semicolon in sum = sum % 10**:
   - ○ The semicolon is missing after sum = sum % 10, which causes a syntax error.

b) How many breakpoints you need to fix those errors? What are the steps you have taken to fix the error you identified in the code fragment?

You would need **two breakpoints** to debug and fix the errors effectively:

1. **Breakpoint 1 (Inner Loop Condition)**:
   - ○ Place a breakpoint before the line while(sum == 0) to check the logic flow and verify that the correct condition (while(sum > 0)) is applied to process the digits.
2. **Breakpoint 2 (Arithmetic Operation)**:
   - ○ Set a breakpoint at s = s * (sum / 10) to ensure that you're properly summing the digits. This will allow you to see that the operation should be s = s + (sum % 10).

Steps Taken to Fix the Errors

a. **Fix the Condition in the Inner Loop**:

- Change the condition from while(sum == 0) to while(sum > 0) so that the loop correctly processes each digit of the number.

b. **Fix the Summing Logic**:

- Replace s = s * (sum / 10) with s = s + (sum % 10) to correctly sum the digits of the number.

c. **Add the Missing Semicolon**:

- Add the semicolon after sum = sum % 10 to resolve the syntax error.

c) Submit your complete executable code?

```
import java.util.Scanner;

public class MagicNumberCheck {

    public static void main(String args[]) {

        Scanner ob = new Scanner(System.in);

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

        int n = ob.nextInt();

        int sum = 0, num = n;

        // Loop to keep reducing the number until it's a single digit

        while (num > 9) {

            sum = num;

            int s = 0;

            // Sum up the digits of the number
            while (sum > 0) {
                s = s + (sum % 10); // Add digits
                sum = sum / 10;     // Remove the last digit
            }
            num = s; // Update num to be the sum of digits
        }
        // A magic number results in 1 after repeated sum of digits
        if (num == 1) {
            System.out.println(n + " is a Magic Number.");
        } else {
            System.out.println(n + " is not a Magic Number.");
        }
    }
}
```

5) Merge Sort:

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

There are **four main errors** in the code:

1. **Wrong Array Slicing in mergeSort**:
    - The current code slices the array incorrectly. Instead of int[] left = leftHalf(array+1) and int[] right = rightHalf(array-1), you should pass the whole array and let the leftHalf and rightHalf methods do the slicing.
    - **Fix**: Pass array directly to the leftHalf and rightHalf functions.

2. **Wrong Array Indexing in the Merge Function**:
    - In the call to merge, the code incorrectly uses left++ and right--. This modifies the arrays incorrectly. It should pass the left and right arrays as they are.
    - **Fix**: Pass left and right directly to merge(array, left, right).
3. **Invalid Slicing Logic in mergeSort**:
    - The merge sort logic tries to split and merge the array incorrectly. The logic in mergeSort should allow the splitting into two equal halves without adding or subtracting from the indices.
    - **Fix**: Correct the call to leftHalf(array) and rightHalf(array) to pass the array directly, without modifications.
4. **Incorrect Logic in mergeSort Method**:
    - The recursion logic is correct but is misused because of improper array slicing. Once the slicing errors are fixed, the recursion should work as expected.

b) How many breakpoints you need to fix those errors? What are the steps you have taken to fix the error you identified in the code fragment?

You would need **two breakpoints** to debug the issues:

1. **Breakpoint 1 (Array Splitting)**:
    - Place a breakpoint before the line int[] left = leftHalf(array+1); and int[] right = rightHalf(array-1);. Here, you can observe the values being passed to the array-splitting methods and confirm that the array is being split correctly.
2. **Breakpoint 2 (Merging Logic)**:
    - Set a breakpoint before calling merge(array, left++, right--);. Verify that the left and right arrays are passed correctly, and ensure that merge properly sorts and combines them into result.

a. **Fix the Array Slicing in mergeSort**:

- Replace int[] left = leftHalf(array+1); with int[] left = leftHalf(array);.

- Replace int[] right = rightHalf(array-1); with int[] right = rightHalf(array);.

b. **Fix the Merge Call**:

- Replace merge(array, left++, right--); with merge(array, left, right);.

c. **Fix the Logic in mergeSort to Properly Split Arrays**:

- Ensure that leftHalf(array) and rightHalf(array) are being passed the full array and correctly split it.

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

6) Multiply Matrices:

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

There are **three main errors** in the code:

Incorrect Indexing in the Multiplication Loop:

i) In the multiplication logic, the expressions first[c-1][c-k] and second[k-1][k-d] are incorrect. These indices will go out of bounds and do not represent the correct elements for matrix multiplication.

ii) **Fix**: The correct expressions should be first[c][k] and second[k][d].

Loop Bound for Multiplication (k):

iii) The loop for k should iterate over the number of columns of the first matrix (which is n, not p).

iv) **Fix**: Change the loop bound from k < p to k < n.

Possible Input Mismatch Issue:

v) The program does not prompt the user for correct input dimensions if the matrices cannot be multiplied, but this is handled by the condition if (n != p).

b) How many breakpoints you need to fix those errors? What are the steps you have taken to fix the error you identified in the code fragment?

You would need **two breakpoints** to debug this program:

Breakpoint 1 (Matrix Input):

Place a breakpoint before the matrix multiplication logic starts, especially in the loop for (k = 0; k < p; k++). This will help ensure that the correct matrix elements are being multiplied.

Breakpoint 2 (Matrix Multiplication Result):

Set a breakpoint after calculating multiply[c][d] to check if the multiplication results are as expected.

a. **Fix the Indexing for Matrix Multiplication**:

- Replace first[c-1][c-k] with first[c][k] and second[k-1][k-d] with second[k][d] to correctly access the matrix elements.

b. **Fix the Loop Bound for k**:

- Change the loop bound for k from k < p to k < n, as n represents the number of columns in the first matrix.

c. **Check Matrix Compatibility for Multiplication**:

- The condition if (n != p) is correct and will prevent invalid matrix multiplications, so no change is needed here.

c) 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();

                }

            }

            // Matrix multiplication logic

            for (c = 0; c < m; c++) {

                for (d = 0; d < q; d++) {

                    sum = 0;  // Initialize sum to zero for each element of the result matrix

                    for (k = 0; k < n; k++) {  // Correct loop bounds

                        sum += first[c][k] * second[k][d];  // Corrected indexing

                    }

                    multiply[c][d] = sum;

                }

            }
            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:

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

Errors Identified in the Code

Syntax Error in Insert Method

In the insert method, the statement: i + = (i + h / h--) % maxSize;

contains an extra space between + and =, which is causing a syntax error.

Incorrect Logic in Insert Method

The logic used for calculating the next index during quadratic probing seems wrong: i + = (i + h / h--) % maxSize;

It should be using quadratic probing logic i = (i + h * h) % maxSize.

Rehash Logic After Removal

In the remove method, after an element is removed, the rehashing logic is flawed: for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) % maxSize)

This statement could skip rehashing some elements and result in an incorrect reordering of elements in the hash table.

b) How many breakpoints you need to fix those errors? What are the steps you have taken to fix the error you identified in the code fragment?

You would need **two breakpoints** to debug the code:

1. **Breakpoint 1 (Insert Method)**: Set a breakpoint inside the `insert` method to verify that the correct index is being computed for inserting a new key-value pair.
2. **Breakpoint 2 (Remove Method)**: Set a breakpoint in the `remove` method to verify if the rehashing works correctly after an element is removed.

Steps to Fix the Errors

Fix the Syntax Error: Correct the assignment operation in the insert method by removing the space:
i = (i + h * h) % maxSize;

Fix the Rehashing Logic: Modify the rehashing logic in the remove method to correctly reinsert all elements after deletion:
for (i = (i + h * h) % maxSize; keys[i] != null; i = (i + h * h) % maxSize)

c) Submit your complete executable code?

```
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 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;  // Fixed logic for quadratic probing
            h++;
        } 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;
            h++;
        }
    return null;
}
/** Function to remove key and its value **/
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;
        h++;
    }
    keys[i] = vals[i] = null;
    i = (i + h * h) % maxSize;
    h++;
    while (keys[i] != null) {
        String tmp1 = keys[i], tmp2 = vals[i];
        keys[i] = vals[i] = null;
        currentSize--;
        insert(tmp1, tmp2);
        i = (i + h * h) % maxSize;
```

```
            h++;
        }
        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();
    }
}
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;
do {
    System.out.println("\nHash Table Operations\n");
    System.out.println("1. insert ");
    System.out.println("2. remove");
    System.out.println("3. get");
    System.out.println("4. clear");
    System.out.println("5. size");
    int choice = scan.nextInt();
    switch (choice) {
        case 1:
            System.out.println("Enter key and value");
            qpht.insert(scan.next(), scan.next());
            break;
        case 2:
            System.out.println("Enter key");
            qpht.remove(scan.next());
            break;
        case 3:
            System.out.println("Enter key");
            System.out.println("Value = " + qpht.get(scan.next()));
```

```
                    break;
                case 4:
                    qpht.makeEmpty();
                    System.out.println("Hash Table Cleared\n");
                    break;
                case 5:
                    System.out.println("Size = " + qpht.getSize());
                    break;
                default:
                    System.out.println("Wrong Entry \n ");
                    break;
            }
            qpht.printHashTable();
            System.out.println("\nDo you want to continue (Type y or n) \n");
            ch = scan.next().charAt(0);
        } while (ch == 'Y' || ch == 'y');
    }
}
```

8) Sorting Array:

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

There are **four errors** in the provided code:

1. **Class Name Issue**: The class name Ascending _Order contains an extra space, which is not allowed in Java class names.
2. **Incorrect Loop Condition**: The outer for loop has an incorrect condition: i >= n. This condition will not allow the loop to run at all. It should be i < n.
3. **Unnecessary Semicolon** after the outer for loop: The for loop has a semicolon at the end, which stops the block of code from executing correctly.
4. **Incorrect Sorting Logic**: The condition inside the inner loop is if (a[i] <= a[j]), which swaps only when a[i] is smaller than or equal to a[j]. This is incorrect for ascending order. The condition should be if (a[i] > a[j]).

b) How many breakpoints you need to fix those errors? What are the steps you have taken to fix the error you identified in the code fragment?

You need **4 breakpoints** to identify and fix the issues:

1. **Breakpoint 1**: At the line with the class name, to check for the incorrect name and remove the space.
2. **Breakpoint 2**: At the outer loop condition (`i >= n`), to identify that the condition is not correct and should be changed.
3. **Breakpoint 3**: At the semicolon after the outer `for` loop, to remove the unwanted semicolon that is causing the block not to execute properly.
4. **Breakpoint 4**: At the inner loop's comparison statement, to correct the comparison for sorting in ascending order.

Step 1: Fix the class name.

- **Action**: Remove the space in the class name and change it from Ascending _Order to AscendingOrder.

Step 2: Correct the outer loop condition.

- **Action**: Change the outer for loop condition from i >= n to i < n.

Step 3: Remove the semicolon after the outer loop.

- **Action**: Delete the semicolon (;) after the outer loop definition so the block can execute correctly.

Step 4: Fix the inner loop comparison.

- **Action**: Change the comparison from if (a[i] <= a[j]) to if (a[i] > a[j]) to correctly sort the array in ascending order.

c) Submit your complete executable code?

```
import java.util.Scanner;

public class AscendingOrder // Removed space in class name

{

    public static void main(String[] args)

    {

        int n, temp;

        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();
```

```
}
// Sorting logic
for (int i = 0; i < n; i++)  // Fixed loop condition
{
    for (int j = i + 1; j < n; j++)
    {
        if (a[i] > a[j])  // Corrected the comparison for ascending order
        {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
}
// Output the sorted array
System.out.print("Ascending Order: ");
for (int i = 0; i < n; i++)
{
    System.out.print(a[i]);
    if (i < n - 1) {
        System.out.print(", ");
```

```
         }
      }
   }
}
```

9) Stack Implementation:

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

There are **four errors** in the program:

1. **Decrementing top in push method**: The code decrements top (top--) in the push method when it should increment it to add an element to the top of the stack.
2. **Incorrect loop condition in display method**: The loop inside display method has the condition i > top, which is incorrect. It should be i <= top to iterate through the stack elements correctly.
3. **Printing uninitialized elements in display method**: The current display method starts iterating from i = 0 but goes beyond top, which could print uninitialized elements. The loop should be restricted to the elements currently in the stack.
4. **Unused import**: The import java.util.Arrays; is unnecessary because Arrays class is not used in this code.

b) How many breakpoints you need to fix those errors? What are the steps you have taken to fix the error you identified in the code fragment?

You need **3 breakpoints** to fix these errors:

1. **Breakpoint 1**: In the push method where top-- is used to fix the logic and replace it with top++.
2. **Breakpoint 2**: In the display method, to fix the condition of the for loop (i > top to i <= top).

3. **Breakpoint 3**: Also in the display method, to remove the unnecessary loop beyond top and restrict it only to the elements in the stack.

---

3. Steps taken to fix the errors:

Step 1: Fix the push method.

- **Action**: Replace top-- with top++ to correctly add the element to the stack.

Step 2: Correct the display method's loop condition.

- **Action**: Change i > top to i <= top in the display method to correctly iterate through the stack's elements.

Step 3: Remove the unused import statement.

- **Action**: Remove import java.util.Arrays; since it is unnecessary.

c) Submit your complete executable code?

```
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++;  // Fixed: Increment top before adding the value
        stack[top] = value;
    }
}
public void pop(){
    if(!isEmpty()) {
        top--;  // Fixed: Decrement top to remove the 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");
        } else {
            for(int i = 0; i <= top; i++){  // Fixed: Corrected loop condition
                System.out.print(stack[i] + " ");
            }
            System.out.println();
        }
    }
}
public class StackReviseDemo {
    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);
        newStack.display(); // Displays the stack
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.display(); // Displays the stack after multiple pops
    }
}
```

10) Tower Of Hanoi:

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

Post-increment and post-decrement issues:

- topN ++, inter--, from+1, and to+1 are incorrect. These operations are modifying the values unnecessarily. In recursive function calls, these values should be passed as they are, without any increments or decrements.

Missing semicolon:

- The recursive call doTowers(topN ++, inter--, from+1, to+1) is missing a semicolon after it.

b) How many breakpoints you need to fix those errors? What are the steps you have taken to fix the error you identified in the code fragment?

Steps to fix:

- Remove the ++ and -- operators on topN, from, inter, and to in the recursive calls.
- Add the missing semicolon after the second recursive call.

c) Submit your complete executable code?

```
public class MainClass {

  public static void main(String[] args) {

    int nDisks = 3;  // Number of disks

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

  }

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

    if (topN == 1) {

      // Base case: Move one disk from 'from' to 'to'

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

    } else {

      // Move topN-1 disks from 'from' to 'inter' using 'to' as auxiliary

      doTowers(topN - 1, from, to, inter);

      // Move the nth disk from 'from' to 'to'

      System.out.println("Disk " + topN + " from " + from + " to " + to);

      // Move the topN-1 disks from 'inter' to 'to' using 'from' as auxiliary

      doTowers(topN - 1, inter, from, to);

    }

  }

}
```