IT 314 SOFTWARE ENGINEERING

Lab 7- Program Inspection, Debugging and Static Analysis

202201251

[ii] CODE DEBUGGING:

Debugging is the process of localizing, analyzing, and removing suspected errors in the code (Java code given in the .zip file)

[1] Armstrong Number

- There is one error in the program related to the computation of the remainder, as previously identified.
- 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.
- The corrected executable code is as follows:

```
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

- There are two errors in the program as mentioned above.
- To fix these errors:
    - For Error 1 in the gcd function, you need one breakpoint at the beginning of the while loop to verify the correct execution of the loop.

- For Error 2 in the lcm function, you would need to review the logic for calculating LCM, as it's a logical error.

The corrected executable code is as follows:

```java
import java.util.Scanner;

public class GCD_LCM {
    static int gcd(int x, int y) {
        int a, b;
        a = (x > y) ? x : y; // a is greater number
        b = (x < y) ? x : y; // b is smaller number
        while (b != 0) { // Fixed the while loop condition
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }

    static int lcm(int x, int y) {
        return (x * y) / gcd(x, y); // Calculate LCM using GCD
    }

    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

- There is one error in the program, as identified above.
- To fix this error, you would need one breakpoint at the line: int option1 = opt[n][w]; to ensure n and w are correctly used without unintended increments.
- The corrected executable code is as follows:

```
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];
          boolean[] take = new boolean[N + 1]; // Array to track items
taken

        // 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);
        }

        int[][] opt = new int[N + 1][W + 1];
        boolean[][] sol = new boolean[N + 1][W + 1];

        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {
                int option1 = opt[n - 1][w]; // Fixed the increment here
                int option2 = Integer.MIN_VALUE;

                if (weight[n] <= w) {
                    option2 = profit[n] + opt[n - 1][w - weight[n]];
                }

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

```
        // Print the items with their profits, weights, and whether they
are taken
         System.out.println("Item" + "\t" + "Profit" + "\t" + "Weight" +
"\t" + "Take");
        for (int n = 1; n <= N; n++) {
            take[n] = sol[n][W]; // Determine if the item is taken
             System.out.println(n + "\t" + profit[n] + "\t" + weight[n] +
"\t" + take[n]);
        }
    }
}
```

[4] Magic Number

- There are two errors in the program, as identified above.
- To fix these errors, you would need one breakpoint at the beginning of the inner while loop to verify the execution of the loop. You can also use breakpoints to check the values of num and s during execution.
- The corrected executable code is as follows:

```
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) { // Fixed the condition here
                s += sum % 10; // Corrected to sum the digits
```

```
                sum = sum / 10; // Corrected to divide by 10 to get the
next digit
            }
            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

- There are multiple errors in the program, as identified above.
- To fix these errors, you would need to set breakpoints to examine the values of left, right, and array during execution. You can also use breakpoints to check the values of i1 and i2 inside the merge method.
- The corrected executable code is as follows:

```
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));
    }

    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            int[] left = leftHalf(array);
```

```
            int[] right = rightHalf(array);
            mergeSort(left);
            mergeSort(right);
            merge(array, left, right);
        }
    }

    public static int[] leftHalf(int[] array) {
        int size1 = array.length / 2;
        int[] left = new int[size1];
        for (int i = 0; i < size1; i++) {
            left[i] = array[i];
        }
        return left;
    }

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

      public static void merge(int[] result, int[] left, int[]
right) {
        int i1 = 0;
        int i2 = 0;
        for (int i = 0; i < result.length; i++) {
                if (i2 >= right.length || (i1 < left.length &&
left[i1] <= right[i2])) {
                result[i] = left[i1];
                i1++;
            } else {
                result[i] = right[i2];
                i2++;
            }
```

```
        }
    }
}
```

[6] Multiply Matrices

- There are multiple errors in the program, as identified above.
- To fix these errors, you would need to set breakpoints to examine the values of c, d, k, and sum during execution. You should pay particular attention to the nested loops where the matrix multiplication occurs.
- The corrected executable code is as follows:

```
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];
                    }
                    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");
            }
        }
        in.close(); // Close the scanner
    }
}
```

[7] Quadratic Probing

- There are three errors in the program, as identified above.
- To fix these errors, you would need to set breakpoints and step through the code while examining variables like i, h, tmp1, and tmp2. You should pay attention to the logic of the insert, remove, and get methods.
- The corrected executable code is as follows:

```
import java.util.Scanner;

class QuadraticProbingHashTable {
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

    public QuadraticProbingHashTable(int capacity) {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    public void makeEmpty() {
        currentSize = 0;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    public int getSize() {
        return currentSize;
    }

    public boolean isFull() {
        return currentSize == maxSize;
    }

    public boolean isEmpty() {
        return getSize() == 0;
    }
```

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

private int hash(String key) {
    return key.hashCode() % maxSize;
}

public void insert(String key, String val) {
    int tmp = hash(key);
    int i = tmp, h = 1;

    do {
        if (keys[i] == null) {
            keys[i] = key;
            vals[i] = val;
            currentSize++;
            return;
        }
        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        }
        i += (h * h++) % maxSize;
    } while (i != tmp);
}

public String get(String key) {
    int i = hash(key), h = 1;
    while (keys[i] != null) {
        if (keys[i].equals(key))
            return vals[i];
        i = (i + h * h++) % maxSize;
    }
    return null;
}

public void remove(String key) {
```

```
        if (!contains(key))
            return;

        int i = hash(key), h = 1;
        while (!key.equals(keys[i]))
            i = (i + h * h++) % maxSize;

        keys[i] = vals[i] = null;
         for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h *
h++) % maxSize) {
            String tmp1 = keys[i], tmp2 = vals[i];
            keys[i] = vals[i] = null;
            currentSize--;
            insert(tmp1, tmp2);
        }
        currentSize--;
    }

    public void printHashTable() {
        System.out.println("\nHash Table: ");
        for (int i = 0; i < maxSize; i++) {
            if (keys[i] != null) {
                System.out.println(keys[i] + " " + vals[i]);
            }
        }
        System.out.println();
    }
}

public class QuadraticProbingHashTableTest {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("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');
```

```
        scan.close(); // Close the scanner
    }
}
```

[8] Sorting Array

- There are two errors in the program as identified above.
- To fix these errors, you need to set breakpoints and step through the code. You should focus on the class name, the loop conditions, and the unnecessary semicolon.
- The corrected executable code is as follows:

```
import java.util.Scanner;

public class AscendingOrder {
    public static void main(String[] args) {
        int n, temp;
        Scanner s = new Scanner(System.in);
          System.out.print("Enter the number of elements 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();
        }

        // Bubble sort to arrange the elements in ascending order
        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]); // Print the last element without a
trailing comma
        s.close(); // Close the scanner
    }
}
```

[9] Stack Implementation

- There are three errors in the program, as identified above.
- To fix these errors, you would need to set breakpoints and step through the code, focusing on the push, pop, and display methods. Correct the push and display methods and add the missing pop method to provide a complete stack implementation.
- The corrected executable code is as follows:

```
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();
    }
}
```

[10] Tower of Hanoi

- There is one error in the program, as identified above.
- To fix this error, you need to replace the line:
  doTowers(topN ++, inter--, from+1, to+1);
- with the correct version:
  doTowers(topN - 1, inter, from, to);
- The corrected executable code is as follows:

```
public class MainClass {
```

```
    public static void main(String[] args) {
        int nDisks = 3; // Number of disks
         doTowers(nDisks, 'A', 'B', 'C'); // A, B, and C are the names of
the rods
    }

     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); // Move topN-1 disks
from 'from' to 'inter'
             System.out.println("Disk " + topN + " from " + from + " to "
+ to); // Move the bottom disk
              doTowers(topN - 1, inter, from, to); // Move the disks from
'inter' to 'to'
        }
    }
}
```

Code:

200 LOC

```
/**** QUESTION 1: Printing Hashes ****/
console.log('*** QUESTION 1: Printing Hashes ***');
const printHashes = () => {
  let hash = '';
  for (let i = 0; i < 7; i++) {
    hash += '#';
    console.log(hash);
  }
}
printHashes();
/**** QUESTION 2: FizzBuz ****/
console.log('*** QUESTION 2: FizzBuz ***');
const fizzBuz  = () =>  {
  for (let i = 1; i <= 100; i++) {
    if (i % 3 === 0 && i % 5 === 0) {
      console.log('FizBuzz', i);
    } else if (i % 3 !== 0 && i % 5 == 0) {
      console.log('Buz', i);
    } else if (i % 3 == 0) {
      console.log('Fizz', i);
    } else {
      console.log(i);
    }
  }
}
fizzBuz();
/**** QUESTION 3: Maximum ***
 *
 *
 */
console.log('*** QUESTION 3: Maximum ***');
const findMax = (a, b, c) => {
  let max;
  if (a > b && a > c) {
    max = a;
  } else if (b > a && b > c) {
    max = b;
  } else {
    max = c;
  }
  return max;
}
console.log(findMax(10, -9, 5));
console.log(findMax(-10, -9, -20));
console.log(findMax(-10, -9, 20));
```

```
/*
 *** QUESTION 4: Reverse Array ***
 */
console.log('*** QUESTION 4: Reverse Array ***');
const reverseArray = (arr) => {
  let newArr = [];
  let len = arr.length - 1;
  for (let i = 0; i <= len; i++) {
    newArr[i] = arr[len - i];
  }
  return newArr;
}
console.log(reverseArray([1, 2, 3, 4, 5]));
console.log(reverseArray(['A', 'B', 'C']));

/*
 *** QUESTION 5: Modify Array***
 */
console.log('*** QUESTION 5: Modify Array***');
const modifyArray = (arr) => {
  let modifiedArr = [];
  if (arr.length < 5) {
    return 'Not Found';
  }
  for (let i = 0; i < arr.length; i++)
    i === 4
      ? (modifiedArr[i] = arr[i].toUpperCase())
      : (modifiedArr[i] = arr[i]);

  return modifiedArr;
}
console.log(
  modifyArray(['Avocado', 'Tomato', 'Potato', 'Mango', 'Lemon', 'Carrot'])
);

/*
 *** QUESTION 6 : Seven unique random numbers in an array***
 */
console.log('*** QUESTION 6 : Seven unique random numbers in an array***');
// solution 1
function sevenRandomNumbers() {
  const randNumbers = [];
  while (randNumbers.length < 7) {
    const randNum = Math.floor(Math.random() * 9) + 1;
    if (randNumbers.indexOf(randNum) === -1) {
      randNumbers.push(randNum);
    }
  }
  return randNumbers;
}
```

```
console.log(sevenRandomNumbers());
// solution 2
function sevenRandomNumbers() {
  const randNumbers = [];
  let i = 0;
  let randNum;
  let len = randNumbers.length;
  while (i < 7) {
    randNum = Math.floor(Math.random() * 10 + 1);
    if (i == 0) {
      randNumbers[i] = randNum;
    } else {
      if (randNumbers.indexOf(randNum) == -1) {
        randNumbers[i] = randNum;
      } else {
        i--;
      }
    }
    i++;
  }

  return randNumbers;
}
console.log(sevenRandomNumbers());
function sevenRandomNumber() {
  const randNumbers = [];
  while (array.length < 7) {
    const random = Math.floor(Math.random() * 9);
    if (randNumbers.indexOf(random) === -1) {
      randNumbers.push(random);
    }
  }
  console.log(randNumbers);
  return randNumbers;
}

sevenRandomNumber();

/*
 *** QUESTION 7: Sum ***
 */
console.log('*** QUESTION 7: Sum of any number of arguments***');
const sumOfArgs = (...args) => {
  let total = 0;
  args.forEach(arg => (total += arg));
  return total;
};
console.log(sumOfArgs(1, 2, 3));
console.log(sumOfArgs(1, 2, 3, 4));
```

```
function sum() {
  let total = 0;
  Array.from(arguments).forEach(arg => (total += arg));
  return total;
}

console.log(sum(1, 2, 3));
console.log(sum(1, 2, 3, 4));
/*
 *** QUESTION 8: Replace the middle item with three items***
 */
console.log('*** QUESTION 8: Replace the middle item with three items***');
const removeMiddleItem = (arr, itemOne, itemTwo, itemThree) => {
  let arrayLen = arr.length;
  let middleIndex;
  if (arrayLen % 2 === 0) {
    middleIndex = arrayLen / 2 - 1;
    arr.splice(middleIndex, 2, itemOne, itemTwo, itemThree);
  } else {
    middleIndex = (arrayLen + 1) / 2 - 1;
    arr.splice(middleIndex, 1, itemOne, itemTwo, itemThree);
  }
  return arr;
}
console.log(removeMiddleItem([1, 2, 3], 'item 1', 'item2', 'item3'));
console.log(removeMiddleItem([1, 2, 3, 4], 'item 1', 'item2', 'item3'));
console.log(removeMiddleItem([1, 2, 3], 4, 5, 6));
/*
 *** QUESTION 9: Extract numbers from text ***
 */
console.log('*** QUESTION 9: Extract numbers from a text ***');
const calculateAnnualIncome = () => {
  const pattern = /[0-9]+/g;
  const incomes = 'He earns 5000 euro from salary per month, 10000 euro annual bonus,
15000 euro online courses per month.'.match(
    pattern
  );
  let sum = 0;
  incomes.forEach((income, i) => {
    if (i == 0 || i == 2) {
      sum += parseFloat(income) * 12;
    } else {
      sum += parseFloat(income);
    }
  });
  return sum;
}
console.log(calculateAnnualIncome());
/*
```

```
 *** QUESTION 10: Check if a sub string is an end of a text ***
 */
console.log('*** QUESTION 10: Check if a sub string is an end of a text ***');
const checkEndOfString = (mainString, subString) => {
  return mainString.endsWith(subString);
}
console.log(checkEndOfString('integrity', 'ity'));
console.log(checkEndOfString('integration', 'tio'));
```

Category A: Data Reference Errors

1. Does a referenced variable have a value that is unset or uninitialized? This probably is the most frequent programming error; it occurs in a wide variety of circumstances. For each reference to a data item (variable, array element, field in a structure), attempt to "prove" informally that the item has a value at that point.

Ans: while (array.length < 7) {

Error:array is referenced but not initialized. The correct variable here should be randNumbers, which is initialized.

2. For all array references, is each subscript value within the defined bounds of the corresponding dimension?

Ans: newArr[i] = arr[len - i];

Error: There's no explicit check to ensure `len - i` stays within bounds for array `arr`. While this seems logically fine for typical input, consider bounds checking when performing similar operations.

3. For all array references, does each subscript have an integer value? This is not necessarily an error in all languages, but it is a dangerous practice.

Ans: arr.splice(middleIndex, 2, itemOne, itemTwo, itemThree);

Issue: The middleIndex is calculated as a float when the array length is odd (due to division), but this is later handled by the splice function. JavaScript automatically truncates it to an integer, but in other languages or cases, this might be an issue.

4. For all references through pointer or reference variables, is the referenced memory currently allocated? This is known as the "dangling reference" problem. It occurs in situations where the lifetime of a pointer is greater than the lifetime of the referenced memory. One situation occurs where a pointer references a local variable within a procedure, the pointer value is assigned to an output parameter or a global variable, the procedure returns (freeing the referenced location), and later the program attempts to use the pointer value. In a manner similar to checking for the prior errors, try to prove informally that, in each reference using a pointer variable, the reference memory exists.

Ans: JavaScript does not have explicit pointer variables, so dangling pointers or unallocated memory issues are less common in this context. However, you should ensure that arrays or objects are initialized before using them, which your code generally handles.

5. When a memory area has alias names with differing attributes, does the data value in this area have the correct attributes when referenced via one of these names? Situations to look for are the use of the EQUIVALENCE statement in FORTRAN, and the REDEFINES clause in COBOL. As an example, a FORTRAN program contains a real variable A and an integer variable B; both are made aliases for the same memory area by using an EQUIVALENCE statement. If the program stores a value into A and then references variable B, an error is likely present since the machine would use the floating-point bit representation in the memory area as an integer.
Ans: JavaScript does not have aliasing issues like EQUIVALENCE in FORTRAN or REDEFINES in COBOL.

6. Does a variable's value have a type or attribute other than what the compiler expects? This situation might occur where a C, C++, or COBOL program reads a record into memory and references it by using a structure, but the physical representation of the record differs from the structure definition.
Ans: incomes.forEach((income, i) => { sum += parseFloat(income) * 12; });

- Potential Issue:income is extracted from a string and converted to a float. Ensure that this conversion is always valid, and parseFloat() doesn't return `NaN`.

7. Are there any explicit or implicit addressing problems if, on the machine being used, the units of memory allocation are smaller than the units of memory addressability? For instance, in some environments, fixed-length bit strings do not necessarily begin on byte boundaries, but addresses only point to byte boundaries. If a program computes the address of a bit string and later refers to the string through this address, the wrong memory location may be referenced.This situation also could occur when passing a bit-string argument to a subroutine.
Ans: JavaScript abstracts memory management, so this issue doesn't apply.

8. If pointer or reference variables are used, does the referenced memory location have the attributes the compiler expects? An example of such an error is where a C++pointer upon which a data structure is based is assigned the address of a different data structure.
Ans: Since JavaScript does not have pointers, this issue is less relevant.

9. If a data structure is referenced in multiple procedures or subroutines, is the structure defined identically in each procedure?
Ans: When referencing arrays like in reverseArray() and removeMiddleItem(), the structure of arrays remains consistent, so no issues here.

10. When indexing into a string, are the limits of the string off by-one errors in indexing operations or in subscript references to arrays?
Ans: There are no off-by-one errors in this case.

11. For object-oriented languages, are all inheritance requirements met in the implementing
Class?
Ans: No classes or inheritance pattern in this code.

Category B: Data-Declaration Errors

1. Have all variables been explicitly declared? A failure to do so is not necessarily an error, but it is a common source of trouble. For instance, if a program subroutine receives an array parameter,and fails to define the parameter as an array (as in a DIMENSION statement, for example), a reference to the array (such as C=A (I)) is interpreted as a function call, leading to the machine's attempting to execute the array as a program. Also, if a variable is not explicitly declared in an inner procedure or block, is it understood that the variable is shared with the enclosing block?
Ans: const randNumbers = []; while (array.length < 7)

- Error: The array variable in this code is not declared, which is an oversight.

2. If all attributes of a variable are not explicitly stated in the declaration, are the defaults well understood? For instance, the default attributes received in Java are often a source of surprise.
Ans: JavaScript variables don't have fixed types (dynamically typed), so you should be cautious about how variables are used and what values they might hold at runtime. For instance, using a variable as both a number and a string in different parts of the code might lead to unexpected behavior.

3. Where a variable is initialized in a declarative statement, is it properly initialized? In many languages, initialization of arrays and strings is somewhat complicated and, hence, error prone.
Ans: let sum = 0;
let max;
let randNumbers = [];
These variables are properly initialized at the time of declaration, so no errors.

4. Is each variable assigned the correct length and data type?
Ans: Variables in JavaScript are dynamically typed, but length management (e.g., in arrays) is critical.

5. Is the initialization of a variable consistent with its memory type?
Ans: Memory type errors don't apply in the same way here since JavaScript is loosely typed.

6. Are there any variables with similar names (VOLT and VOLTS, for example)? This is not necessarily an error, but it should be seen as a warning that the names may have been confused somewhere within the program.
Ans: All variables in the code have distinct names, so there are no obvious cases where similarly named variables might cause confusion.

Category C: Computation Errors

1. Are there any computations using variables having inconsistent (such as non-arithmetic) data types?
Ans: Since JavaScript is a dynamically typed language, variables can change types at runtime.

2. Are there any mixed-mode computations? An example is the addition of a floating-point variable to an integer variable. Such occurrences are not necessarily errors, but they should be explored carefully to ensure that the language's conversion rules are understood. Consider the following Java snippet showing the rounding error that can occur when working with integers:
int x = 1;
int y = 2;
int z = 0;
z = x/y;
System.out.println ("z = " + z);

OUTPUT:
z = 0
Ans: This shows an integer division problem, where dividing two integers results in a loss of precision (truncation). In JavaScript, floating-point division is automatic, but if you're dividing integers, you may need to handle precision manually.

3. Are there any computations using variables having the same data type but different lengths?
Ans: This issue is not typically a problem in JavaScript because it dynamically handles variable sizes.

4. Is the data type of the target variable of an assignment smaller than the data type or result of the right-hand expression?
Ans: This is not an issue in JavaScript because it dynamically adjusts variable types.

5. Is an overflow or underflow expression possible during the computation of an expression? That is, the end result may appear to have valid value, but an intermediate result might be too big or too small for the programming language's data types.
Ans: JavaScript uses floating-point arithmetic for numbers, and while it does handle large numbers, there is a possibility of overflow or underflow with extremely large or small values.
let total = 0; total += arg;
If a large number is passed into the sumOfArgs function, JavaScript could potentially overflow,

6. Is it possible for the divisor in a division operation to be zero?
Ans: There are no explicit division operations in this code.

7. If the underlying machine represents variables in base-2 form, are there any sequences of the resulting inaccuracy? That is, 10 x 0.1 is rarely equal to 1.0 on a binary machine.
Ans: No such operations involving decimals are in this code.

8. Where applicable, can the value of a variable go outside the meaningful range? For example,statements assigning a value to the variable PROBABILITY might be checked to ensure that the assigned value will always be positive and not greater than
Ans:

- Example: let incomes = 'He earns 5000 euro from salary...'.match(pattern);

In the income calculation, the numbers extracted from the string are within a reasonable range.

9. For expressions containing more than one operator, are the assumptions about the order of evaluation and precedence of operators correct?
Ans: The logical operators here follow standard precedence rules, and no issues are present.
Example: if (i % 3 === 0 && i % 5 === 0) { console.log('FizBuzz', i); }

10. Are there any invalid uses of integer arithmetic, particularly divisions? For instance, if i is an integer variable, whether the expression 2*i/2 == i depends on whether i has an odd or an even value and whether the multiplication or division is performed first.
Ans: There are no direct integer division operations that would cause problems.

Category D: Comparison Errors

1. Are there any comparisons between variables having different data types, such as comparing a character string to an address, date, or number?
Ans:

- Example: if (age === "30") { ... }
- Issue: JavaScript's === operator does not perform type coercion, so comparing "30" (a string) to 30 (a number) will return false. Using == would result in type coercion, which might yield unexpected results.

2. Are there any mixed-mode comparisons or comparisons between variables of different lengths? If so, ensure that the conversion rules are well understood.
Ans: if (name === 'Alex' && name.length < 5) { ... }
If both variables are strings, the comparison is valid.

3. Are the comparison operators correct? Programmers frequently confuse such relations as at most, at least, greater than, not less than, less than or equal.
Ans: if (age <= 30) { ... }
Programmers often confuse operators like >= and <=. Ensure you're using the correct operator for the intended logic. For example,<= means "at most" and >= means "at least".

4. Does each Boolean expression state what it is supposed to state? Programmers often make mistakes when writing logical expressions involving and, or, and not.
Ans: if (age > 18 && age < 60) { ... }
This is a correct Boolean expression, ensuring `age` is within a valid range.

5. Are the operands of a Boolean operator Boolean? Have comparison and Boolean operators been erroneously mixed together? This represents another frequent class of mistakes.Examples of a few typical mistakes are illustrated here. If you want to determine whether i is between 2 and 10, the expression 2<i<10 is incorrect; instead, it should be (2<i) && (i<10). If you want to determine whether i is greater than x or y, i>x||y is incorrect; instead, it should be (i>x)||(i>y). If you want to compare three numbers for equality, if(a==b==c) does something quite different. If you want to test the mathematical relation x>y>z, the correct expression is (x>y)&&(y>z).
Ans: if (age > 30 || 40) { ... }

- Fix: if (age > 30 || age === 40) { ... }

6. Are there any comparisons between fractional or floating- point numbers that are represented in base-2 by the underlying machine? This is an occasional source of errors because of truncation and base-2 approximations of base-10 numbers.
Ans: Comparisons between floating-point numbers can lead to errors due to truncation and base-2 approximations of base-10 numbers, as many decimal fractions cannot be represented exactly in binary.

7. For expressions containing more than one Boolean operator, are the assumptions about the order of evaluation and the precedence of operators correct? That is, if you see an expression such as (if((a==2) && (b==2) || (c==3)), is it well understood whether the and or the or is performed first?
Ans:

- Example: && has higher precedence than `||`, so the expression is evaluated as (a==2 && b==2)|| c==3. This may not be what you intended.

8. Does the way in which the compiler evaluates Boolean expressions affect the program? For instance, the statement if((x==0 && (x/y)>z) may be acceptable for compilers that end the test as soon as one side of an and is false, but may cause a division-by-zero error with other compilers.
Ans: JavaScript short-circuits && evaluations, meaning if the first condition (x!==0) is false, it won't evaluate the second condition, preventing division by zero.

Category E: Control-Flow Errors

1. If the program contains a multiway branch such as a computed GO TO, can the index variable ever exceed the number of branch possibilities? For example, in the statement
GO TO (200, 300, 400), i
will i always have the value of 1, 2, or 3?
Ans: This code does not include a multiway branch

2. Will every loop eventually terminate? Devise an informal proof or argument showing that each loop will terminate.
Ans:

- **Example**:

```
for (let i = 0; i < 7; i++) {
    console.log(hash);
}
```

This loop initializes i to 0 and increments i until it reaches 7. Since i is incremented in each iteration, the loop will eventually terminate after 7 iterations.

3. Will the program, module, or subroutine eventually terminate?
Ans: Your functions like fizzBuz,findMax, etc., will terminate as they run through a defined number of iterations or return values after processing.

4. Is it possible that, because of the conditions upon entry, a loop will never execute? If so, does
this represent an over- sight? For instance, if you had the following loops headed by the following statements:

```
for (i==x ; i<=z; i++) {
   ...
 }
while (NOTFOUND) {
  ...
 }
```

what happens if NOTFOUND is initially false or if x is greater than z?

Ans: If x is greater than z, this loop will not execute.
If NOTFOUND is initially false, this loop will not execute.

5. For a loop controlled by both iteration and a Boolean condition (a searching loop, for example) what are the consequences of loop fall-through? For example, for the psuedo-code loop headed by

```
DO I=1 to TABLESIZE WHILE (NOTFOUND)
```

what happens if NOTFOUND never becomes false?
Ans: If NOTFOUND remains true, this loop will execute for TABLESIZE iterations. If NOTFOUND never becomes false, it can lead to unnecessary iterations

6. Are there any off-by-one errors, such as one too many or too few iterations? This is a common error in zero-based loops. You will often forget to count "0" as a number. For example, if you want to create Java code for a loop that counted to 10, the following would be wrong, as it counts to 11:

```
for (int i=0; i<=10;i++) {
   System.out.println(i);
 }
```

Correct, the loop is iterated 10 times:

```
for (int i=0; i<=9;i++) {
  System.out.println(i);
```

Ans: There is no off-by-one error.

7. If the language contains a concept of statement groups or code blocks (e.g., do-while or {...}), is there an explicit while for each group and do the do's correspond to their appropriate groups? Or is there a closing bracket for each open bracket? Most modern compilers will complain of such mismatches.
Ans: JavaScript uses {} for code blocks.

8. Are there any non-exhaustive decisions? For instance, if an input parameter's expected values are 1, 2, or 3, does the logic assume that it must be 3 if it is not 1 or 2? If so, is the assumption valid?
Ans: The code accounts for various input cases. For example, the prime-checking function handles edge cases like numbers less than 2. All expected inputs have been considered, and no assumptions about unhandled cases exist.

Category F: Interface Errors

1. Does the number of parameters received by this module equal the number of arguments sent by each of the calling modules? Also, is the order correct?
Ans: Yes, the number of parameter received by this module is equal to the number of arguments sent by each module.

2. Do the attributes (e.g., data type and size) of each parameter match the attributes of each corresponding argument?
Ans: This code matches all attributes.

3. Does the units system of each parameter match the units system of each corresponding argument? For example, is the parameter expressed in degrees but the argument expressed in Radians?
Ans:This category typically applies to cases where units matter, like measurement conversions. This code doesn't seem to involve units that need conversion, so this point is not applicable.

4. Does the number of arguments transmitted by this module to another module equal the number of parameters expected by that module?
Ans: No functions in this code call other modules (functions), so this point does not apply.

5. Do the attributes of each argument transmitted to another module match the attributes of the corresponding parameter in that module?
Ans: Since there are no calls to external modules, this point does not apply.

6. Does the units system of each argument transmitted to another module match the units system of the corresponding parameter in that module?
Ans: Again, no external module calls, so this point does not apply.

7. If built-in functions are invoked, are the number, attributes, and order of the arguments Correct?
Ans: `Math.floor()` and `String.endsWith()` are invoked correctly with the appropriate number of arguments and correct data types.

8. Does a subroutine alter a parameter that is intended to be only an input value?
Ans: All functions handle parameters appropriately without altering input parameters that are intended as inputs.

9. If global variables are present, do they have the same definition and attributes in all modules that reference them?
Ans: There are no global variables defined or used in your code, so this point does not apply.

Category G: Input / Output Errors

1. If files are explicitly declared, are their attributes correct?
Ans: Since there are no file operations in this code.

2. Are the attributes on the file's OPEN statement correct?
Ans: Since there are no file operations in this code.

3. Is there sufficient memory available to hold the file your program will read?
Ans: This code does not read from files, so there are no memory constraints regarding file operations to consider.

4. Have all files been opened before use?
Ans: No files are opened in This code.

5. Have all files been closed after use?
Ans: No files are closed in this code since no files are opened.

6. Are end-of-file conditions detected and handled correctly?
Ans: There are no file operations in this code.

7. Are I/O error conditions handled correctly?
Ans: This code does not perform any I/O operations, so there are no I/O error conditions to handle.

8. Are there spelling or grammatical errors in any text that is printed or displayed by the program?
Ans: There are a few text outputs that could use review for clarity or consistency:

- In **`fizzBuz`**, "FizBuzz" should be "FizzBuzz" for consistency with the traditional game.
- In **`modifyArray`**, the return value `'Not Found'` might be better expressed as `'Not enough elements'` or a similar phrase for clarity.

Category H: Other Checks

1. If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.
Ans: This code does not include a cross-reference listing

2. If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.
Ans: Since JavaScript does not have strict data type enforcement, there are no specific attribute listings to check.

3. If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully. Warning messages are indications that the compiler suspects that you are doing something of questionable validity; all of these suspicions should be reviewed. Informational messages may list undeclared variables or language uses that impede code optimization.
Ans: This code should be run in a JavaScript environment to check for warnings or messages.

4. Is the program or module sufficiently robust? That is, does it check its input for validity?
Ans: This code does not perform thorough input validation

5. Is there a function missing from the program?
Ans: The code seems fairly complete for the tasks outlined.

Program Inspection

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

- **Logic Error in `countWords1`:** The regular expression pattern does not account for whole word matches. It can count substrings rather than complete words (e.g., counting "love" in "lovely"). This can lead to incorrect occurrences being reported.
- **Logic Error in `countWords2`:** The use of `includes()` checks for substrings instead of exact word matches. This may lead to inflated counts when the target word is part of another word.
- **Comment Mistake in `agesGreaterEighteen`:** The comment suggests an incorrect output with a trailing comma in the expected return value, which could confuse readers regarding the intended functionality.
- **Uninitialized `middleIndex` in `removeMiddleItem`:** If the function is called with an empty array, the variable `middleIndex` is undefined, potentially causing an error when used with the `splice` method.
- **Variable Reassignment Issue in `removeAll`:** The function attempts to reassign `todoList`, which is declared as a `const`. This will result in a runtime error. The array should be cleared using `todoList.length = 0;`.
- **Lack of Input Validation in multiple functions:** Functions like `addProduct` and `editTask` do not validate input parameters, which can lead to potential undefined behavior when incorrect values are passed (e.g., a non-existent index).
- **Incorrect Logic in `toggleAll`:** The logic inside the `toggleAll` function could be simplified. The nested conditions make it less readable and introduce complexity that can lead to errors in toggling all tasks.

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

- **Category B: Logic Errors** would be particularly effective here, as many errors arise from logic missteps (like miscounting words or handling array indices incorrectly).
- **Category E: Interface Errors** is also crucial since ensuring that functions receive the correct types and number of arguments can prevent many issues before they arise.

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

- **Logical Errors:**Issues like incorrect outputs (e.g., counting "love" in "lovely") often go unnoticed until runtime or testing.
- **Runtime Errors:**Errors caused by incorrect data types or accessing undefined indices will only show up during execution, not in a static inspection.
- **Performance Issues:**Inefficient algorithms (e.g., repeated array traversals) may not be identified through inspection but could lead to performance bottlenecks.

4. Is the program inspection technique is worth applicable?
Ans: Yes, it is worth applying**.**

- **Benefits:**
    - Helps identify many syntactical and some logical errors early in the development process.
    - Facilitates code understanding among team members, improving collaboration.
- **Limitations:**
    - Alone, it might miss subtle logical and runtime errors.
    - It should be part of a broader testing strategy, including unit tests and integration tests, to ensure comprehensive coverage.

LOC : 700

```
console.log('==================== BEGIN Q1
============================ ')
/* === Question 1 === */
// 1. 1
const printHashes = () => {
  let hash = ''
  for (let i = 0; i < 7; i++) {
    hash += '#'
    console.log(hash)
  }
}

printHashes()
console.log('------------------------------------------------------------ ')

// 1. 2
const multiplicationTable = n => {
  for (let i = 0; i < n; i++) {
    console.log(`${i} * ${i} = ${i * i}`)
  }
}

multiplicationTable(11)
console.log('------------------------------------------------------------ ')

// 1. 3
const exponentialTable = () => {
  console.log(`i\ti^2\t^3`)
  for (let i = 0; i < 11; i++) {
    console.log(`${i}\t${i ** 2}\t${i ** 3}`)
  }
}

exponentialTable()
console.log('------------------------------------------------------------ ')

// 1. 4
const countries = [
  'ALBANIA',
  'BOLIVIA',
  'CANADA',
  'DENMARK',
  'ETHIOPIA',
  'FINLAND',
  'GERMANY',
  'HUNGARY',
```

```
  'IRELAND',
  'JAPAN',
  'KENYA'
]
// const createArrayOfArrays = arr => {
//   const newArray = []
//   for (const element of arr) {
//     let name = element[0] + element.slice(1).toLowerCase()
//     newArray.push([name, element.slice(0, 3), element.length])
//   }
//   return newArray
// }

const createArrayOfArrays = arr =>
  arr.map(country => {
    let name = country[0] + country.slice(1).toLowerCase()
    return [name, country.slice(0, 3), country.length]
  })

console.log(createArrayOfArrays(countries))

console.log('==================== END Q1
============================ ')

console.log('==================== BEGIN Q2
============================ ')
/* === Question 2 === */
// 2. 1
const products = [
  { product: 'banana', price: 3 },
  { product: 'mango', price: 6 },
  { product: 'potato', price: ' ' },
  { product: 'avocado', price: 8 },
  { product: 'coffee', price: 10 },
  { product: 'tea', price: '' }
]

const printProductItems = arr => {
  for (const { product, price } of arr) {
    let formattedPrice = typeof price == 'number' ? price : 'unknown'
    console.log(`The price of ${product} is ${formattedPrice}`)
  }
}

printProductItems(products)
console.log('---------------------------------------------------------- ')

// 2. 2
const sumOfAllPrices = arr => {
  let total = 0
```

```
  for (const { price } of arr) {
    if (typeof price == 'number') {
      total += price
    }
  }
  return total
}

console.log('the sum of all  prices using for of', sumOfAllPrices(products))
console.log('---------------------------------------------------------- ')

// 2. 3
const total = products
  .map(prod => prod.price)
  .filter(price => typeof price == 'number')
  .reduce((curr, acc) => curr + acc)

console.log('total from method chaining', total)
console.log('---------------------------------------------------------- ')

// 2. 4
const totalUsingReduce = products.reduce((accu, curr) => {
  let price = typeof curr.price == 'number' ? curr.price : 0
  return accu + price
}, 0)

console.log('reduce total', totalUsingReduce)
console.log(sumOfAllPrices(products))
console.log('==================== END Q2
============================ ')

console.log('==================== BEGIN Q3
============================ ')
/*=== Question 3 === */

// 3. 1
const howManyDaysInMonth = () => {
  const userInput = prompt('Enter a month: ')
    .trim()
    .toLowerCase()

  const firstLetter = userInput[0].toUpperCase()
  const remainingStr = userInput.slice(1)
  const month = firstLetter + remainingStr

  let statement
  let days
  switch (userInput) {
    case 'february':
      days = 28
```

```
      statement = `${month} has ${days} days.`
      break
    case 'april':
    case 'june':
    case 'september':
    case 'november':
      days = 30
      statement = `${month} has ${days} days.`
      break
    case 'january':
    case 'march':
    case 'may':
    case 'july':
    case 'august':
    case 'october':
    case 'december':
      days = 31
      statement = `${month} has ${days} days.`
      break
    default:
      return 'The given value is not a month'
  }
  return statement
}

console.log(howManyDaysInMonth())
console.log('------------------------------------------------------------ ')

// 3. 2
const generate = (type = 'id') => {
  const randomId = (n = 6) => {
    const str =
'0123456ABCDEFGHIJKLMNOPKRSTUVWXYZabcdefghihjklmnopqrstuvwxyz'
    let id = ''
    for (let i = 0; i < n; i++) {
      let index = Math.floor(Math.random() * str.length)
      id = id + str[index]
    }
    return id
  }

  const hexaColor = function() {
    let st = '0123456789abcdef'.split('')
    let color = '#'
    for (let i = 0; i < 6; i++) {
      let index = Math.floor(Math.random() * st.length)
      color += st[index]
    }
    return color
  }
```

```
  const rgbColor = function() {
    let redColor = Math.floor(Math.random() * 256)
    let greenColor = Math.floor(Math.random() * 256)
    let blueColor = Math.floor(Math.random() * 256)

    const rgb = `rgb(${redColor},${greenColor},${blueColor})`
    return rgb
  }
  switch (type.toLowerCase()) {
    case 'id':
      return randomId()
    case 'hexa':
    case 'hexadecimal':
      return hexaColor()
    case 'rgb':
      return rgbColor()
    default:
      return 'Not a valid format'
  }
}

console.log(generate())
console.log(generate('id'))
console.log(generate('hexa'))
console.log(generate('hexadecimal'))
console.log(generate('rgb'))
console.log(generate('RGB'))
console.log(generate('rgb'))

console.log('==================== END Q3
============================ ')

console.log('==================== BEGIN Q4
============================ ')

/*=== Question 4 ==== */

// 4. 1
const countries2 = [
  'ALBANIA',
  'BOLIVIA',
  'CANADA',
  'DENMARK',
  'ETHIOPIA',
  'FINLAND',
  'GERMANY',
  'HUNGARY',
  'IRELAND',
  'JAPAN',
```

```
  'KENYA'
]

const c = ['ESTONIA', 'FRANCE', 'GHANA']
countries2.splice(4, 3, ...c)
console.log(countries2)
console.log('---------------------------------------------------------- ')

// 4. 2
const checkUniqueness = arr => {
  for (const element of arr) {
    if (arr.indexOf(element) !== arr.lastIndexOf(element)) {
      return false
    }
  }
  return true
}
const arrOne = [1, 4, 6, 2, 1]
console.log(checkUniqueness(arrOne)) // false
const arrTwo = [1, 4, 6, 2, 3]
console.log(checkUniqueness(arrTwo)) // true
console.log('---------------------------------------------------------- ')

// 4. 3
const checkDataTypes = (arr, type) => arr.every(elem => typeof elem === type)

const numbers = [1, 3, 4]
const names = ['Asab', '30DaysOfJavaScript']
const bools = [true, false, true, true, false]
const mixedData = ['Asab', 'JS', true, 2019, { name: 'Asab', lang: 'JS' }]
const obj = [{ name: 'Asab', lang: 'JS' }]
console.log(checkDataTypes(numbers, 'number')) // true
console.log(checkDataTypes(numbers, 'string')) // false
console.log(checkDataTypes(names, 'string')) // true
console.log(checkDataTypes(bools, 'boolean')) // true
console.log(checkDataTypes(mixedData, 'boolean')) // false
console.log(checkDataTypes(obj, 'object')) // true

console.log('==================== END Q4
============================ ')

console.log('==================== BEGIN Q5
============================ ')
/*=== Question 5 ==== */

// 5. 1
const fullStack = [
  ['HTML', 'CSS', 'JS', 'React'],
  ['Node', 'Express', 'MongoDB']
]
```

```
const [frontEnd, backEnd] = fullStack
console.log(frontEnd, backEnd)
console.log('---------------------------------------------------------- ')

// 5. 2
const student = ['David', ['HTM', 'CSS', 'JS', 'React'], [98, 85, 90, 95]]
const [
  name,
  [html, css, js, react],
  [htmlScore, cssScore, jsScore, reactScore]
] = student
console.log(
  name,
  html,
  css,
  js,
  react,
  htmlScore,
  cssScore,
  jsScore,
  reactScore
)
console.log('---------------------------------------------------------- ')

// 5. 3
const students = [
  ['David', ['HTM', 'CSS', 'JS', 'React'], [98, 85, 90, 95]],
  ['John', ['HTM', 'CSS', 'JS', 'React'], [85, 80, 85, 80]]
]

// one way
// const convertArrayToObject = arr => {
//   const newArray = []
//   for (const [name, skills, scores] of arr) {
//     newArray.push({ name, skills, scores })
//   }
//   return newArray
// }

// another way
const convertArrayToObject = arr =>
  arr.map(([name, skills, scores]) => {
    return { name, skills, scores }
  })

console.log(convertArrayToObject(students))
console.log('==================== END Q5
============================ ')

console.log('==================== BEGIN Q6
```

```
============================ ')
/*=== Question 6 ==== */

// 6. 1
const sumOfAllNums = (...args) => {
  let sum = 0
  for (const element of args) {
    sum += element
  }
  return sum
}

console.log(sumOfAllNums(2, 3, 1)) // 6
console.log(sumOfAllNums(1, 2, 3, 4, 5)) // 15
console.log(sumOfAllNums(1000, 900, 120)) // 2020

function sumOfAllNums2() {
  let sum = 0
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i]
  }
  return sum
}

console.log(sumOfAllNums2(2, 3, 1)) // 6
console.log(sumOfAllNums2(1, 2, 3, 4, 5)) // 15
console.log(sumOfAllNums2(1000, 900, 120)) // 2020
console.log('------------------------------------------------------------ ')

// 6. 2
const [x, y] = [2, a => a ** 2 - 4 * a + 3]
const valueOfX = x // 2
const valueOfY = y // (a) => a ** 2 - 4 * a + 3
console.log(valueOfX, valueOfY)
console.log(y(x)) // -1
console.log('------------------------------------------------------------ ')

// 6. 3

const studentObj = {
  name: 'David',
  age: 25,
  skills: {
    frontEnd: [
      { skill: 'HTML', level: 10 },
      { skill: 'CSS', level: 8 },
      { skill: 'JS', level: 8 },
      { skill: 'React', level: 9 }
    ],
    backEnd: [
```

```
      { skill: 'Node', level: 7 },
      { skill: 'GraphQL', level: 8 }
    ],
    dataBase: [{ skill: 'MongoDB', level: 7.5 }],
    dataScience: ['Python', 'R', 'D3.js']
  }
}
const devOps = [
  { skill: 'AWS', level: 7 },
  { skill: 'Jenkin', level: 7 },
  { skill: 'Git', level: 8 }
]
const copiedStudentObj = {
  ...studentObj,
  skills: {
    ...studentObj.skills,
    frontEnd: [...studentObj.skills.frontEnd, { skill: 'Bootstrap', level: 8 }],
    backEnd: [...studentObj.skills.backEnd, { skill: 'Express', level: 8 }],
    dataBase: [...studentObj.skills.dataBase, { skill: 'SQL', level: 8 }],
    dataScience: [...studentObj.skills.dataScience, 'SQL'],
    devOps: [...devOps]
  }
}
console.log(copiedStudentObj)
console.log('==================== END Q6
============================ ')

console.log('==================== BEGIN Q7
============================ ')
/*=== Question 7 ==== */

// 7. 1
const showDateTime = (format = 'dd/mm/yyyy') => {
  const months = [
    'January',
    'February',
    'March',
    'April',
    'May',
    'June',
    'July',
    'August',
    'September',
    'October',
    'November',
    'December'
  ]
  const now = new Date()
  const year = now.getFullYear()
  const month = months[now.getMonth()]
```

```
  const mm = now.getMonth() + 1
  const date = now.getDate()
  const dd = now.getDate()
  let hours = now.getHours()
  let hh = now.getHours()
  let minutes = now.getMinutes()

  let seconds = now.getSeconds()
  if (hours < 10) {
    hours = '0' + hours
  }
  if (minutes < 10) {
    minutes = '0' + minutes
  }
  if (seconds < 10) {
    seconds = '0' + seconds
  }

  const dateMonthYear = `${month} ${date}, ${year}`

  const time = hours + ':' + minutes
  const fullTime = dateMonthYear + ' ' + time
  //   return fullTime + `:${seconds}`
  switch (format) {
    case 'dd/mm/yyyy':
      return `${dd}/${mm}/${year}`
    case 'dd-mm-yyyy':
      return `${dd}-${mm}-${year}`
    case 'dd/mm/yyyy hh:mm':
      return `${dd}/${mm}/${year} ${hours}:${minutes}`
    case 'dd-mm-yyyy hh:mm':
      return `${dd}-${mm}-${year} ${hours}:${minutes}`
    case 'MMM DD, YYYY':
      return `${month} ${dd}, ${year}`
    case 'MMM DD, YYYY hh:mm':
      return `${month} ${dd}, ${year} ${hours}:${minutes}`

    default:
      return `${dd}/${mm}/${year}`
  }
}

console.log(showDateTime())
console.log(showDateTime('dd-mm-yyyy'))
console.log(showDateTime('dd-mm-yyyy hh:mm'))
console.log(showDateTime('dd/mm/yyyy hh:mm'))
console.log(showDateTime('Month DD, YYYY'))
console.log(showDateTime('MMM DD, YYYY hh:mm'))
console.log('---------------------------------------------------------- ')
// 7. 1
```

```
const todoList = [
  {
    task: 'Prepare JS Test',
    time: '5/4/2020 8:30',
    completed: true
  },
  {
    task: 'Give JS Test',
    time: '6/4/2020 10:00',
    completed: false
  },
  {
    task: 'Assess Test Result',
    time: '4/3/2019 1:00',
    completed: false
  }
]

const addTask = task => {
  const time = showDateTime('dd-mm-yyyy hh:mm')
  const completed = false
  todoList.push({ task, time, completed })
}

const removeTask = index => {
  todoList.splice(index, 1)
}

const editTask = (index, newTask) => {
  todoList[index].task = newTask
  todoList.push({ task, time, completed })
}
const toggleTask = index => {
  todoList[index].completed = !todoList[index].completed
}

const toggleAll = () => {
  const checkCompleted = todoList.filter(todo => todo.completed === true).length
  if (checkCompleted.length === todoList.length) {
    for (const item of todoList) {
      item.completed = !item.completed
    }
  } else {
    for (const item of todoList) {
      item.completed = true
    }
  }
}
```

```
console.log(todoList)

toggleAll()
console.log(todoList)

const removeAll = () => {
  // todoList = []
  todoList.splice()
}
console.log('==================== END Q7
============================ ')

console.log('==================== BEGIN Q8
============================ ')

/*=== Question 8 ==== */

// 8. 1

// generic function to sort items both for string and number
const sortItems = (arr, key) => {
  const copiedArr = [...arr]
  copiedArr.sort((a, b) => {
    if (a[key] > b[key]) return -1
    if (a[key] < b[key]) return 1
    else return 0
  })
  return copiedArr
}

const largestCountries = async (n = 10) => {
  const API_URL = 'https://restcountries.eu/rest/v2/all'
  const countriesArea = []
  const response = await fetch(API_URL)
  const data = await response.json()

  for (const { name, area } of data) {
    countriesArea.push({ country: name, area })
  }

  const countriesSortedByArea = sortItems(countriesArea, 'area').slice(0, n)
  console.log(`${n} most largest countries`, countriesSortedByArea)
}

largestCountries(10)

const numberOfLanguages = async () => {
  const API_URL = 'https://restcountries.eu/rest/v2/all'
  const langSet = new Set()
  const response = await fetch(API_URL)
```

```
  const data = await response.json()

  for (const { languages } of data) {
    for (const { name } of languages) {
      langSet.add(name)
    }
  }

  console.log(Array.from(langSet).sort())
  console.log(
    'Total number of langauges in the countries API:',
    Array.from(langSet).length
  )
  console.log('------------------------------------------------------- ')
}
numberOfLanguages()

const mostSpokenLanguages = async (n = 10) => {
  const API_URL = 'https://restcountries.eu/rest/v2/all'
  const langSet = new Set()
  const allLangArr = []
  const languageFrequency = []

  try {
    const response = await fetch(API_URL)
    const data = await response.json()

    for (const { languages } of data) {
      for (const { name } of languages) {
        allLangArr.push(name)
        langSet.add(name)
      }
    }

    for (l of langSet) {
      const x = allLangArr.filter(ln => l == ln)
      languageFrequency.push({
        lang: l,
        count: x.length
      })
    }

    const sortedLanguages = sortItems(languageFrequency, 'count').slice(0, n)
    console.log(`${n} most spoken languages`, sortedLanguages)
  } catch {
    console.log('Something goes wrong')
  }
  console.log('------------------------------------------------------- ')
}
```

```
console.log('Most spoken languages', mostSpokenLanguages(15))

// 8.2

const add = (a, b) => {
  if (b) {
    return a + b
  }
  if (!a) {
    return 'at least one parameter is required'
  }
  return b => a + b
}
console.log(add(2, 3))
console.log(add())
console.log(add(2)(3))

console.log('==================== END Q8
============================ ')

console.log('==================== BEGIN Q9
============================ ')

/*=== Question 9 ==== */
/*
9.1 What is the difference between forEach, map, filter and reduce? (1 pt)
9.2 What is the difference between find and filter? (1pt)
9.3 Which of the following mutate array: map, filter, reduce, slice, splice, concat, sort,
some? (2pt) only splice and sort modify an array
*/

const generateColor = (type = 'hexa', n = 1) => {
  const hexaColor = function() {
    let st = '0123456789abcdef'.split('')
    let color = '#'
    for (let i = 0; i < 6; i++) {
      let index = Math.floor(Math.random() * st.length)
      color += st[index]
    }
    return color
  }

  const rgbColor = function() {
    let redColor
    let greenColor
    let blueColor

    redColor = Math.floor(Math.random() * 256)
    greenColor = Math.floor(Math.random() * 256)
    blueColor = Math.floor(Math.random() * 256)
```

```
    const rgb = `rgb(${redColor},${greenColor},${blueColor})`
    return rgb
  }

  const rgbColors = []
  const hexaColors = []
  if (n > 1) {
    if (type == 'hexa') {
      for (let i = 0; i < n; i++) {
        hexaColors.push(hexaColor())
      }
    } else if (type == 'rgb') {
      for (let i = 0; i < n; i++) {
        rgbColors.push(rgbColor())
      }
    }
  }

  switch (type) {
    case 'hexa':
      const hexa = n > 1 ? hexaColors : hexaColor()
      return hexa
    case 'rgb':
      const rgb = n > 1 ? rgbColors : rgbColor()
      return rgb
    default:
      return hexaColor()
  }
}

console.log(generateColor())
console.log(generateColor('hexa'))
console.log(generateColor('rgb'))
console.log(generateColor('hexa', 3))
console.log(generateColor('rgb', 3))
console.log('==================== END Q9
============================ ')

console.log('==================== BEGIN Q10
============================ ')

/*=== Question 10 ==== */

const para = `Studies that estimate and rank the most common words in English
examine texts written in English. Perhaps the most comprehensive such analysis is
one that was conducted against the Oxford English Corpus (OEC), a very large
collection of texts from around the world that are written in the English language. A
text corpus is a large collection of written works that are organised in a way that
makes such analysis easier.`
```

```
const cleanText = txt => {
  const pattern = /[^\w ]/g
  return txt.replace(pattern, '')
}

const mostFrequentWord = (txt, n = 10) => {
  const cleanedText = cleanText(txt)
  const words = cleanedText.split(' ')
  const map = new Map()
  for (const word of words) {
    if (map.has(word)) {
      let count = map.get(word.toLowerCase()) + 1
      map.set(word.toLowerCase(), count)
    } else {
      map.set(word.toLowerCase(), 1)
    }
  }

  return Array.from(map)
    .map(([word, count]) => {
      return { word, count }
    })
    .slice(0, n)
}

console.log('Most frequent words', sortItems(mostFrequentWord(para), 'count'))
console.log('------------------------------------------------------------ ')

// 10. 2
const sentence = `@He@ @%met%^$##%# his mom at no@on and s@he was
watching %^$#an epso@ide%^$# of the begni@nging of her Sol@os. Her
te@net%^$# hel@ped %^$#her to le@vel up her stats. %^$#After that h@e went to
%^$#kayak driving Civic %^$#Honda.`

const cleanedText = cleanText(sentence)
console.log(cleanedText)
console.log('------------------------------------------------------------ ')

// 10. 3
const checkPalindrome = param => {
  const word = typeof param == 'number' ? param.toString() : param
  let invertedWord = ''
  for (let i = word.length - 1; i >= 0; i--) {
    invertedWord += word[i]
  }
  return word.toLowerCase() === invertedWord.toLowerCase()
}

console.log('Check palindrome the number 123:', checkPalindrome(323))
console.log('Check palindrome the word anna:', checkPalindrome('anna'))
```

```
console.log('Check palindrome the word He:', checkPalindrome('He'))
console.log('---------------------------------------------------------- ')

// 10.4
const words = cleanText(sentence).split(' ')
const palindromes = words.filter(word => checkPalindrome(word))
const numberOfPalindromes = palindromes.length
console.log(
  'Total number of palindrome words in the text:',
  numberOfPalindromes
)

console.log('==================== END Q10
============================ ')
```

Category A: Data Reference Errors

1. Unset or Uninitialized Variable:

Error: A referenced variable has an unset or uninitialized value.

Example: while (array.length < 7)

Correction: The variable 'array' is referenced but not initialized. The correct variable should be 'randNumbers', which is initialized.

2. Array Subscript Out of Bounds:

Error: Array references should always have subscript values within the defined bounds.

Example: newArr[i] = arr[len - i];

Correction: Ensure that len - i remains within bounds for array arr. Consider bounds checking during similar operations.

3. Non-integer Array Subscript:

Error: In some languages, array subscripts may not have integer values, which can lead to issues.

Example: arr.splice(middleIndex, 2, itemOne, itemTwo, itemThree);

Explanation: middleIndex may become a float when dividing an odd array length, which is handled by JavaScript, but other languages may have issues.

4. Dangling Reference:
Error: Memory referenced via pointers may no longer be allocated, leading to the 'dangling reference' problem.

Example: JavaScript doesn't have pointer variables, so this is less common. Ensure arrays or objects are initialized before use.

5. Memory Alias with Incorrect Attributes:
Error: When a memory area has alias names with different attributes, the data value might be interpreted incorrectly.

Example: JavaScript does not have aliasing issues like EQUIVALENCE in FORTRAN or REDEFINES in COBOL.

6. Variable Value with Incorrect Type or Attribute:
Error: A variable's value may have a type or attribute other than what the compiler expects.

Example: incomes.forEach((income, i) => { sum += parseFloat(income) * 12; });

Correction: Ensure parseFloat always produces a valid float and does not return NaN.

7. Addressing Problems:
JavaScript abstracts memory management, so this issue doesn't apply directly.

8. Referenced Memory with Unexpected Attributes:
Since JavaScript does not support low-level memory pointers, this issue is irrelevant. You are safe from this type of error, and the objects in your code are handled correctly.

Category B: Code Errors

1. Incorrect Loop Conditions:
Error: The loop condition may lead to unintended iterations or infinite loops.

Example: for (let i = 0; i < n; i++) { console.log(`${i} * ${i} = ${i * i}`); }

Correction: Ensure that the variable 'n' is defined and has the expected value before entering the loop.

2. Incorrect String Interpolation:
Error: Using template literals incorrectly can lead to unexpected results.

Example: console.log(`${i} * ${i} = ${i * i}`);

Correction: Verify that the variable 'i' is properly defined within scope and has a valid value.

3. Undefined Variables:
Error: Using variables that have not been defined or initialized.

Example: const total = products.map(prod => prod.price).filter(price => typeof price == 'number').reduce((curr, acc) => curr + acc);

Correction: Ensure all variables, such as 'products', are defined before use.

4. Potential Infinite Loop in Input Function:
Error: If user input is not properly validated, it could lead to an infinite loop.

Example: const userInput = prompt('Enter a month: ').trim().toLowerCase();

Correction: Implement validation to check for valid month input before proceeding.

5. Non-Returning Function:
Error: A function may not return a value as expected.

Example: const howManyDaysInMonth = () => { ... }

Correction: Ensure that the function properly returns a value in all code paths.

6. Incorrect Data Structure Access:
Error: Accessing elements of data structures incorrectly can lead to undefined errors.

Example: console.log(createArrayOfArrays(countries));

Correction: Verify the structure of 'countries' before accessing its elements.

7. Invalid Type Comparison:
Error: Comparing values of different types can lead to unexpected results.

Example: let formattedPrice = typeof price == 'number' ? price : 'unknown';

Correction: Ensure type checks are appropriate for the expected data types.

8. Potentially Undefined Return Values:
Error: Functions that may not always return a defined value.

Example: return 'Not a valid format';

Correction: Ensure all return paths provide valid return values, especially in switch statements.

Category C: Resource Management Errors

1. Memory Leak:
Error: Memory that is allocated dynamically is not deallocated, leading to increased memory usage over time.

Example: let obj = {}; obj = null; // Memory leak if not handled correctly.

Correction: Ensure all dynamically allocated memory is released when no longer needed.

2. Resource Exhaustion:
Error: The application consumes resources (e.g., file handles, database connections) without releasing them, leading to exhaustion.

Example: const fs = require('fs'); let file = fs.createReadStream('file.txt');

Correction: Close file streams and connections using appropriate methods when done.

3. Double Free Error:
Error: Attempting to free memory that has already been released leads to undefined behavior.

Example: let arr = new Array(5); delete arr; delete arr;

Correction: Ensure memory is freed only once.

4. Invalid Memory Access:
Error: Accessing memory that has already been freed can cause application crashes or unpredictable behavior.

Example: let arr = new Array(5); arr = null; console.log(arr[0]);

Correction: Avoid accessing memory after it has been freed.

5. File Handle Leak:
Error: Failing to close file handles can lead to the maximum file limit being reached.

Example: fs.open('file.txt', 'r', (err, fd) => { /* no fs.close() */ });

Correction: Always ensure file handles are closed after operations.

6. Network Resource Leak:
Error: Not closing network connections leads to resource leaks.

Example: const socket = new WebSocket('ws://example.com'); // No socket.close()

Correction: Ensure to close network connections when done.

7. Improper Resource Handling in Loops:
Error: Resources allocated within a loop without proper cleanup can lead to leaks.

Example: for (let i = 0; i < 1000; i++) { let resource = allocateResource(); }

Correction: Release resources properly after their use within the loop.

8. Concurrent Resource Access Issues:
Error: Multiple threads accessing shared resources without synchronization can lead to race conditions.

Example: const sharedResource = { value: 0 }; setInterval(() => { sharedResource.value++; }, 100);

Correction: Use proper locking mechanisms to prevent concurrent access issues.

Category D: Control Flow Errors

1. Infinite Loop:
Error: A loop that never terminates, causing the program to run indefinitely.

Example: while (true) { console.log('Hello'); }

Correction: Ensure that the loop has a valid termination condition.

2. Incorrect Loop Condition:
Error: A loop with a condition that does not behave as expected, leading to either premature termination or infinite looping.

Example: for (let i = 0; i < 10; i--) { console.log(i); }

Correction: Verify loop conditions to ensure they are logically sound.

3. Misplaced Break/Continue Statement:
Error: A break or continue statement placed incorrectly can lead to unexpected control flow.

Example: for (let i = 0; i < 10; i++) { if (i === 5) break; console.log(i); }

Correction: Check the placement of break and continue statements within the loop.

4. Unreachable Code:
Error: Code that can never be executed due to the flow of control.

Example: if (false) { console.log('This will never execute.'); }

Correction: Remove or modify unreachable code segments.

5. Improper Exception Handling:
Error: Failing to handle exceptions properly can lead to program crashes or incorrect program flow.

Example: try { throw new Error('Oops!'); } catch (e) { console.log('Error caught!'); }

Correction: Ensure that exceptions are caught and handled appropriately.

6. Improperly Nested Control Structures:
Error: Control structures that are nested incorrectly can lead to confusion and errors in logic.

Example: if (condition1) { if (condition2) { console.log('Nested!'); } }

Correction: Ensure that nesting of control structures is logical and clear.

7. Missing Return Statement:
Error: Functions that are expected to return a value but do not.

Example: function add(a, b) { a + b; }

Correction: Ensure that all functions return the expected values.

8. Improper Use of Asynchronous Code:
Error: Failing to handle asynchronous code properly can lead to unexpected results.

Example: async function fetchData() { const data = await getData(); console.log(data); }

Correction: Ensure proper use of async/await or callbacks.

Category E: Control-Flow Errors

1. Multiway Branch Index Variable Exceeding Limits:

Error: If the program contains a multiway branch, the index variable may exceed the number of branch possibilities.

Example: In the statement GO TO (200, 300, 400), i, can i always have the value of 1, 2, or 3?

Correction: Validate the index variable to ensure it remains within defined branch limits.

2. Loop Termination:

Error: It is crucial to ensure that every loop eventually terminates.

Example: for (let i = 0; i < 7; i++) { console.log(hash); }

Correction: Verify that the loop variable increments correctly to guarantee termination after a defined number of iterations.

3. Function Termination:

Error: Ensure that the program, module, or subroutine eventually terminates.

Example: Functions like fizzBuzz and findMax will terminate after processing a defined number of iterations or returning values.

Correction: Implement termination conditions within functions to ensure they complete.

4. Non-Execution of Loops:

Error: Conditions upon entry may cause a loop to never execute.

Example: For loops like for (i==x; i<=z; i++) or while (NOTFOUND)

Correction: Ensure initial conditions allow for loop execution and handle scenarios where conditions are false at the start.

5. Loop Fall-Through:

Error: A loop controlled by both iteration and a Boolean condition can lead to fall-through.

Example: In the pseudo-code DO I=1 to TABLESIZE WHILE (NOTFOUND)

Correction: Ensure that the Boolean condition will eventually resolve to false to prevent unnecessary iterations.

6. Off-by-One Errors:

Error: Look for off-by-one errors in loops.

Example: for (int i=0; i<=10; i++) { System.out.println(i); }

Correction: Adjust loop boundaries to account for zero-based indexing correctly.

7. Mismatched Code Blocks:

Error: Ensure that the structure of code blocks matches their corresponding control statements.

Example: JavaScript uses {} for code blocks.

Correction: Maintain proper pairing of opening and closing brackets to prevent control flow issues.

8. Non-Exhaustive Decisions:

Error: Check for non-exhaustive decisions in conditional statements.

Example: If expected input values are 1, 2, or 3, does logic assume it must be 3 if it is neither 1 nor 2?

Correction: Ensure all expected input values are handled appropriately to avoid logical fallacies.

Category F: Interface Errors

1. Parameter Mismatch:

Error: The number of parameters received by a module does not equal the number of arguments sent.

Example: If a function is defined as `function example(a, b)` but called with `example(1)`.

Correction: Ensure that all calling modules provide the correct number of arguments.

2. Attribute Mismatch:

Error: The data type and size of parameters do not match their corresponding arguments.

Example: Passing a string to a function expecting an integer.

Correction: Validate and cast types if necessary before passing to functions.

3. Units System Mismatch:
Error: Mismatch in units between parameters and arguments.

Example: A function expects angles in degrees but receives them in radians.

Correction: Ensure consistent units are used or convert as necessary.

4. Incorrect Argument Transmission:
Error: The number of arguments sent to another module does not match the parameters expected.

Example: Calling `function foo(x, y)` with `foo(1)` will cause an error.

Correction: Ensure the correct number of arguments is passed.

5. Argument Attribute Mismatch:
Error: The attributes of each argument do not match the attributes of the corresponding parameters.

Example: A function expecting a float receives an integer instead.

Correction: Check and enforce the expected types and attributes of arguments.

6. Units System of Arguments Mismatch:
Error: Arguments sent to another module do not match the expected units of that module.

Example: Passing a distance in kilometers to a function expecting meters.

Correction: Convert the units before calling the function.

7. Built-in Functions Argument Issues:
Error: Incorrect number or type of arguments passed to built-in functions.

Example: Using `Math.max()` with non-numeric arguments.

Correction: Ensure that built-in functions are invoked with the correct argument types and counts.

8. Parameter Modification:
Error: A subroutine alters a parameter that should be treated as an input value only.

Example: Modifying a parameter intended as read-only in a function.

Correction: Use const or ensure parameters are not modified within the function.

9. Global Variable Consistency:
Error: Global variables are inconsistently defined across modules.

Example: A global variable defined in one module is used differently in another.

Correction: Ensure consistent definitions and attributes for global variables across all modules.

Category G: Input / Output Errors

1. File Declaration Attributes:
Error: If files are explicitly declared, their attributes may be incorrect.

Example: Not applicable as there are no file operations in this code.

Correction: Ensure file attributes match the intended operations.

2. File OPEN Statement Attributes:
Error: Attributes on the file's OPEN statement may be incorrect.

Example: Not applicable as there are no file operations in this code.

Correction: Validate OPEN statement attributes for correctness.

3. Sufficient Memory for File Read:
Error: Insufficient memory available to hold the file that the program will read.

Example: Not applicable since this code does not read from files.

Correction: Ensure adequate memory is available before reading files.

4. File Opening Before Use:
Error: All files must be opened before use.

Example: No files are opened in this code.

Correction: Open all necessary files before performing operations.

5. File Closure After Use:
Error: Failing to close files after use can lead to resource leaks.

Example: No files are closed in this code since no files are opened.

Correction: Always close files after use to prevent leaks.

6. End-of-File Conditions Detection:

Error: End-of-file conditions must be detected and handled correctly.

Example: There are no file operations in this code.

Correction: Implement end-of-file detection for file operations.

7. I/O Error Handling:

Error: Input/Output error conditions must be handled correctly.

Example: This code does not perform any I/O operations, so there are no I/O error conditions to handle.

Correction: Ensure proper error handling mechanisms for I/O operations.

8. Spelling or Grammatical Errors in Output:

Error: Spelling or grammatical errors in any text printed or displayed by the program.

Example: There are a few text outputs that could use review for clarity or consistency.

Correction: Review and correct text outputs for spelling and grammatical accuracy.

Category H: Other Checks

1. Cross-reference Listing:

Check: If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.

Ans: This code does not include a cross-reference listing.

2. Attribute Listing:
Check: If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.

Ans: Since JavaScript does not have strict data type enforcement, there are no specific attribute listings to check.

3. Compiler Warnings and Messages:
Check: If the program compiled successfully, but the computer produced one or more 'warning' or 'informational' messages, check each one carefully.

Ans: This code should be run in a JavaScript environment to check for warnings or messages.

4. Input Validity Checks:
Check: Is the program or module sufficiently robust? That is, does it check its input for validity?

Ans: This code does not perform thorough input validation.

5. Missing Functions:
Check: Is there a function missing from the program?

Ans: The code seems fairly complete for the tasks outlined.

Code:

420 LOC

```
/* ============================== QUESTION 1
=============================================
  Write a function which count the number of occurrence of a word in a paragraph or a
sentence.
  The function countWords takes a paragraph and word as parameters.

=========================================================================
====================== */
const paragraph =
  'I love teaching. If you do not love teaching what else can you love. I love JavaScript if you
do not love something which can give life to your application what else can you love.';
//Method one
const countWords1 = (para, wrd) => {
  const pattern = new RegExp(wrd, 'gi'); //creating regex object using RegExp constructor
  const matches = para.match(pattern) || []; // if the para.match returns null, matches result will
be en empty array
  return matches.length; // geting the length of the array
};
console.log(countWords1(paragraph, 'love'));

//Method
const countWords2 = (para, wrd) => {
  let count = 0;
  const words = para.split(' '); // spliting the paragraph into array of words
  //iterating through words array and checking if the target word is in the array
  for (const word of words) {
    //includes or startsWith could give the same result
    if (word.toLowerCase().includes(wrd.toLowerCase())) {
      console.log(word);
      count++;
    }
  }
  return count;
};
console.log(countWords2(paragraph, 'love'));

/* ====================================== QUESTION 2
=======================================================
    Write a function which takes an array as a parameter and returns an array of the data types
of each item:

=========================================================================
================================== */
const arr = ['Asabeneh', 100, true, null, undefined, { job: 'teaching' }];
const checkDataTypes = arr => {
  const dataTypes = []; // creating an empty array
```

```
  let type; // will be used in the loop to store the data type of each element in the array
  for (const element of arr) {
    type = typeof element; // type checking of each elements
    dataTypes.push(type);
  }
  return dataTypes; // returning the above array which contains all the datatypes of the
elements
};
console.log(checkDataTypes(arr));

/* ========================= QUESTION 3 ============================
   Create a function which filter ages greater than 18 from ages array.
   const ages = [35, 30, 17, 18, 15, 22, 16, 20];
   console.log(agesGreaterEighteen(ages));
   [35, 30, 22, , 20];
  ================================================================= */

const ages = [35, 30, 17, 18, 15, 22, 16, 20];
const agesGreaterEighteen = ages => {
  const agesGreater18 = []; // creating an empty array to store ages which are abover 18
  //iterating through the ages array
  for (const age of ages) {
    //checking if the age if it is greater than 18
    if (age > 18) {
      agesGreater18.push(age); //
    }
  }
  return agesGreater18;
};
console.log(agesGreaterEighteen(ages));
/* ========================= QUESTION 4 ============================
   Write a function which calculate the average age of the group.
  ================================================================= */
const averageAge = ages => {
  let sum = 0; // an accumulator to sum all the ages
  for (const age of ages) {
    // adding each age to the sum varaible
    sum += age;
  }
  return Math.round(sum / ages.length); // rounding the number to the closest integer
};
console.log(averageAge(ages));

/* ========================= QUESTION 5 ============================
  Write a function which remove an item or items from the middle of the array and replace with
two items
  ================================================================= */
const products = ['Milk', 'Coffee', 'Tea', 'Honey', 'Meat', 'Cabage'];
const removeMiddleItem = (arr, ...items) => {
  let middleIndex; // to store the middle index of the array,
```

```
  if (arr.length % 2 === 0) {
    //if the array length is even
    middleIndex = arr.length / 2 - 1;
    //splice takes starting point, number of items to remove and items to replace
    arr.splice(middleIndex, 2, ...items);
  } else {
    //if the array length is odd
    middleIndex = Math.floor(arr.length / 2);
    console.log(middleIndex);
    arr.splice(middleIndex, 1, ...items);
  }

  return arr;
};
console.log(removeMiddleItem(products, 'potato', 'banana'));

/* ====================== QUESTION 6
================================================
   Write a function which can generate a random Finnish car code(Car plate number).
   console.log(genCarPlateNum())
   AFG-205
   console.log(genCarPlateNum())
   JCB-586

================================================================================
========= */

const genCarPlateNum = () => {
  const letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  const numbers = '0123456789';
  let lettersPart = ''; // variable to store, the letters part
  let numbersPart = ''; // variable to store, the letters part
  let indexOne; // random index to extract one of the letter at a time from letters array
  let indexTwo; // random index to extract one of the number at a time from numbers array
  for (let i = 0; i < 3; i++) {
    indexOne = Math.floor(Math.random() * letters.length);
    indexTwo = Math.floor(Math.random() * numbers.length);
    lettersPart += letters[indexOne];
    numbersPart += numbers[indexTwo];
  }
  return `${lettersPart}-${numbersPart}`;
};
console.log(genCarPlateNum());
/* ======================================== QUESTION 7
============================================
  The following shopping cart has four products.
  Create an addProduct, removeProduct ,editProduct , removeAll functions to modify the
shopping cart.
    const shoppingCart = ['Milk','Coffee','Tea', 'Honey'];
    addProduct( "Meat");
```

```
    ["Milk", "Coffee", "Tea", "Honey", "Meat"]
    editProduct(3, "Sugar" );
    ["Milk", "Coffee", "Tea", "Sugar", "Meat"]
    removeProduct(0);
    ["Coffee", "Tea", "Sugar", "Meat"]
    removeProduct(3);
    ["Coffee", "Tea", "Sugar"]

================================================================================ */
const shoppingCart = ['Milk', 'Coffee', 'Tea', 'Honey'];
const addProduct = (products, product) => {
  products.push(product);
  return products;
};
addProduct(shoppingCart, 'Meat');
console.log(shoppingCart);
const editProduct = (products, index, product) => {
  products[index] = product;
  return products;
};
editProduct(shoppingCart, 3, 'Sugar');
console.log(shoppingCart);
const removeProduct = index => {
  shoppingCart.splice(index, 1);
  return shoppingCart;
};
removeProduct(0);
console.log(shoppingCart);
removeProduct(3);
console.log(shoppingCart);

/* ========================= QUESTION 8
=========================================
    Write a function which can generate a random Finnish social security number.
      console.log(genSocialSecurityNum())
    220590 - 255H
    console.log(genSocialSecurityNum())
    190395 - 225J

================================================================================
====== */
const genSocialSecurityNum = () => {
  const letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  const numbers = '0123456789';
  let index = Math.floor(Math.random() * letters.length);
  const letter = letters[index]; // getting a letter from the letters array
  let date = Math.floor(Math.random() * 31) + 1; // date from 1 to 31
  let month = Math.floor(Math.random() * 12) + 1; // month from 1 to 12
```

```
  //if the date or month is less than 10
  if (date < 10) date = '0' + date;
  if (month < 10) month = '0' + month;

  let year = Math.floor(Math.random() * 2019);
  if (year > 100) {
    year = year.toString().substr(-2);
  } else if (year < 10) {
    year = '0' + year;
  }

  let suffix = '';
  for (let i = 0; i < 3; i++) {
    let randomIndex = Math.floor(Math.random() * numbers.length);
    suffix += numbers[randomIndex];
  }

  return `${date}${month}${year}-${suffix}${letter}`;
};
console.log(genSocialSecurityNum());

/* ========================================== QUESTION 9
=========================================================
  The following todoList has three tasks.
  Create an addTask, removeTask, editTask, toggleTask, toggleAll, removeAll functions to
modify the todoList.
  const todoList = [
  {
    task:'Prepare JS Test',
    time:'4/3/2019 8:30',
    completed:true

  },
  {
    task:'Give JS Test',
    time:'4/3/2019 10:00',
    completed:false

  },
  {
    task:'Assess Test Result',
    time:'4/3/2019 1:00',
    completed:false

  }]
=========================================================================
================================== */
const todoList = [
  {
    task: 'Prepare JS Test',
```

```
    time: '4/3/2019 8:30',
    completed: true
  },
  {
    task: 'Give JS Test',
    time: '4/3/2019 10:00',
    completed: false
  },
  {
    task: 'Assess Test Result',
    time: '4/3/2019 1:00',
    completed: false
  }
];

// function to generate date, month, year, hour and minute : day/month/year hh:mm
const displayDateTime = () => {
  const now = new Date();
  const year = now.getFullYear();
  const month = now.getMonth() + 1;
  const date = now.getDate();
  const hours = now.getHours();
  let minutes = now.getMinutes();
  if (minutes < 10) {
    minutes = '0' + minutes;
  }
  return `${date}/${month}/${year} ${hours}:${minutes}`;
};

const addTask = task => {
  const time = displayDateTime();
  const completed = false;
  const newTask = { task, time, completed };
  todoList.push(newTask);
};
const editTask = (index, task) => {
  todoList[index].task = task;
};

const removeTask = index => {
  todoList.splice(index, 1);
};
const toggleTask = (index, task) => {
  todoList[index].completed = !todoList[index].completed;
};
const toggleAll = arr => {
  let completedTodos = 0;
  for (let i = 0; i < arr.length; i++) {
    if (arr[i].completed) {
      completedTodos++;
```

```
    }
    if (completedTodos === arr.length) {
      for (let i = 0; i < arr.length; i++) {
        arr[i].completed = !arr[i].completed;
      }
    } else {
      for (let i = 0; i < arr.length; i++) {
        arr[i].completed = true;
      }
    }
  }
  return arr;
};
console.log(toggleAll(todoList));
const removeAll = () => {
  todoList = [];
  return todoList;
};
console.log(todoList);

/* ========================= QUESTION 10 =========================
  Write a function which check if items of an array are unique?
  const arrOne = [1, 4, 6, 2, 1];
  console.log(checkUniqueness(arrOne));
  false
  const arrTwo = [1, 4, 6, 2, 3]
  console.log(checkUniqueness(arrTwo));
  true
================================================================== */
const checkUniqueness = arr => {
  let uniquenesFlag = true;
  for (const element of arr) {
    if (arr.indexOf(element) !== arr.lastIndexOf(element)) {
      uniquenesFlag = false;
      break;
    }
  }
  return uniquenesFlag;
};
const arrOne = [1, 4, 6, 2, 1];
console.log(checkUniqueness(arrOne));

const arrTwo = [1, 4, 6, 2, 3];
console.log(checkUniqueness(arrTwo));
/* ====================== QUESTION 11
============================================================
   Write a function named shuffle, it takes an array parameter and it returns a shuffled array.

============================================================================
====================== */
```

```
const shuffle = arr => {
  const shuffledArray = []; // empty array to store shaffled array
  let index; // random index which we use to take element from the original array
  let element; // the item we get using the random index will be stored in element
  while (shuffledArray.length !== arr.length) {
    index = Math.floor(Math.random() * arr.length);
    element = arr[index];
    if (shuffledArray.indexOf(element) === -1) {
      shuffledArray.push(element);
    }
  }
  return shuffledArray;
};
console.log(shuffle([1, 2, 3, 4, 5]));

/* ============================== Bonus
==========================================
    Write a function which filter users who has scoresGreaterThan85.
    Write a function which addUser to the user array only if the user does not exist.
    Write a function which addUserSkill which can add skill to a user only if the user exist.
    Write a function which editUser if the user exist in the users array.

==========================================================================
================== */
const users = [
  {
    name: 'Brook',
    scores: 75,
    skills: ['HTM', 'CSS', 'JS'],
    age: 16
  },
  {
    name: 'Alex',
    scores: 80,
    skills: ['HTM', 'CSS', 'JS'],
    age: 18
  },
  {
    name: 'David',
    scores: 75,
    skills: ['HTM', 'CSS'],
    age: 22
  },
  {
    name: 'John',
    scores: 85,
    skills: ['HTM'],
    age: 25
  },
  {
```

```
    name: 'Sara',
    scores: 95,
    skills: ['HTM', 'CSS', 'JS'],
    age: 26
  },
  {
    name: 'Martha',
    scores: 80,
    skills: ['HTM', 'CSS', 'JS'],
    age: 18
  },
  {
    name: 'Thomas',
    scores: 90,
    skills: ['HTM', 'CSS', 'JS'],
    age: 20
  }
];

const scoresGreaterThan85 = arr => {
  const scores = [];
  for (const element of arr) {
    if (element.scores > 85) {
      scores.push(element.scores);
    }
  }
  return scores;
};

console.log(scoresGreaterThan85(users));
const newUser = {
  name: 'Asabeneh',
  scores: 800,
  skills: ['HTML', 'CSS', 'JS'],
  age: 250
};
const addUser = (arr, newUser) => {
  for (const user of arr) {
    if (user['name'] === newUser.name) {
      return ' A user does exist';
    }
  }
  arr.push(newUser);
  return arr;
};
console.log(addUser(users, newUser));

const addUserSkill = (arr, name, skill) => {
  let found = false;
  for (const user of arr) {
```

```
    if (user['name'] === name) {
      user.skills.push(skill);
      found = true;
      break;
    }
  }
  if (!found) {
    return 'A user does not exist';
  }

  return arr;
};
console.log(addUserSkill(users, 'Brook', 'Node'));

const editUser = (arr, name, newUser) => {
  let found = false;
  for (const user of arr) {
    if (user['name'] === name) {
      user.name = newUser.name;
      user.scores = newUser.scores;
      user.skills = newUser.skills;
      user.age = newUser.age;
      found = true;
      break;
    }
  }
  if(!found) {
      return 'A user does not exist';
    }

  return arr;
};
console.log(editUser(users, "Brook", newUser));
console.log(users);
```

Category A: Data Reference Errors

1. **Does a referenced variable have a value that is unset or uninitialized?**
    - **Error**: `while (array.length < 7) {`
    - **Ans**: The variable `array` is referenced but not initialized before use. Instead, you should be referencing `randNumbers`, which is initialized earlier in the code.
2. **For all array references, is each subscript value within the defined bounds of the corresponding dimension?**
    - **Error**: `newArr[i] = arr[len - i];`

- **Ans**: No explicit check ensures that `len - i` is within the bounds of array `arr`. A potential issue might arise if `len - i` exceeds the valid array indices, especially for inputs of unexpected sizes. Consider adding bounds checking.

3. **For all array references, does each subscript have an integer value?**
   - **Issue**: `arr.splice(middleIndex, 2, itemOne, itemTwo, itemThree);`
   - **Ans**: The `middleIndex` is calculated as a floating-point number when the array length is odd. Although JavaScript truncates it to an integer, other languages may not handle it similarly. Ensure the subscript is always an integer.
4. **For all references through pointer or reference variables, is the referenced memory currently allocated?**
   - **Ans**: JavaScript abstracts memory management, so it does not have explicit pointers. However, ensure that arrays or objects are initialized before using them. In this context, your array variables are being initialized properly before usage, so no major issue is found here.
5. **When a memory area has alias names with differing attributes, does the data value in this area have the correct attributes when referenced via one of these names?**
   - **Ans**: JavaScript does not suffer from aliasing issues seen in lower-level languages like FORTRAN or COBOL.
6. **Does a variable's value have a type or attribute other than what the compiler expects?**
   - **Issue**: `incomes.forEach((income, i) => { sum += parseFloat(income) * 12; });`
   - **Ans**: `income` is expected to be parsed as a floating-point number, but there's no validation that ensures it is indeed a valid numeric string. If `parseFloat()` returns `NaN`, this could lead to incorrect results in your calculation of `sum`.
7. **Are there any explicit or implicit addressing problems if, on the machine being used, the units of memory allocation are smaller than the units of memory addressability?**
   - **Ans**: JavaScript abstracts memory allocation and does not allow direct memory address manipulation. This issue is irrelevant for this context.
8. **If pointer or reference variables are used, does the referenced memory location have the attributes the compiler expects?**
   - **Ans**: JavaScript doesn't have pointers like C or C++, so this issue does not apply here.
9. **If a data structure is referenced in multiple procedures or subroutines, is the structure defined identically in each procedure?**
   - **Ans**: In your code, arrays (such as `newArr` and `arr`) are consistently defined and used in multiple procedures, and no structural discrepancies exist. However, ensure consistent length checks when passing arrays between functions.
10. **When indexing into a string, are there off-by-one errors in indexing operations or in subscript references to arrays?**

- **Ans**: Based on the code provided, no off-by-one errors were identified. Ensure careful handling of edge cases, especially for loops that manipulate array indices.

11. **For object-oriented languages, are all inheritance requirements met in the implementing Class?**

- **Ans**: Since the code doesn't use object-oriented classes or inheritance, this is not applicable.

Category B: Data-Declaration Errors

1. **Have all variables been explicitly declared?**
    - **Ans:** Most variables, such as `newArr`, `middleIndex`, and `randNumbers`, are explicitly declared with `let` or `const`. No errors related to undeclared variables were found in the code.
2. **If all attributes of a variable are not explicitly stated in the declaration, are the defaults well understood?**
    - **Ans:** In JavaScript, uninitialized variables declared with `let` or `const` will be `undefined` by default until explicitly initialized. The code does not rely on any confusing default behavior, so no issues here.
3. **Where a variable is initialized in a declarative statement, is it properly initialized?**
    - **Ans:** Initialization is handled properly in the code. For instance, `randNumbers` is initialized as an empty array and populated correctly later in the code.
4. **Is each variable assigned the correct length and data type?**
    - **Ans:** Variables appear to have appropriate types. For example, arrays like `newArr` and `randNumbers` are handled as arrays. No discrepancies in length or type assignments were found.
5. **Is the initialization of a variable consistent with its memory type?**
    - **Ans:** JavaScript handles memory dynamically, and no inconsistencies were found with regard to memory types during variable initialization.
6. **Are there any variables with similar names?**
    - **Ans:** There are no confusingly similar variable names like `VOLT` and `VOLTS` in the provided code.

Category C: Computation Errors

1. **Are there any computations using variables having inconsistent (such as non-arithmetic) data types?**
    - **Ans:** No such inconsistencies were found. All variables used in computations are appropriate for their context, such as numeric operations on numeric variables.
2. **Are there any mixed-mode computations?**

Ans: JavaScript allows mixed-type arithmetic, but it handles type coercion internally.

For example: js

Copy code

```
incomes.forEach((income, i) => { sum += parseFloat(income) * 12; });
```

- Here, `parseFloat(income)` ensures the string `income` is converted to a number before the multiplication. This avoids mixed-mode computation issues.

3. **Are there any computations using variables having the same data type but different lengths?**
   - **Ans:** No such issues were found in the code.
4. **Is the data type of the target variable of an assignment smaller than the data type or result of the right-hand expression?**
   - **Ans:** No overflow/underflow risks related to data types were detected. JavaScript's dynamic typing handles this automatically.
5. **Is an overflow or underflow expression possible during the computation of an expression?**
   - **Ans:** No potential overflows or underflows were detected in the provided code. However, large calculations should always be checked, especially in languages with fixed-length data types.
6. **Is it possible for the divisor in a division operation to be zero?**
   - **Ans:** There are no division operations present in the provided code, so division-by-zero issues are not applicable here.
7. **If the underlying machine represents variables in base-2 form, are there any sequences resulting in inaccuracy?**
   - **Ans:** JavaScript can sometimes introduce small floating-point precision errors, but the code doesn't involve complex floating-point arithmetic. No obvious inaccuracy issues were found.
8. **Can the value of a variable go outside the meaningful range?**
   - **Ans:** In general, there are no checks for variable bounds, but the logic itself appears sound. For example, probability or percentage calculations that should stay within certain ranges are not present here.
9. **For expressions containing more than one operator, are the assumptions about the order of evaluation and precedence of operators correct?**
   - **Ans:** The operator precedence in all expressions, such as in the `forEach` loop, follows JavaScript's rules and appears correct.
10. **Are there any invalid uses of integer arithmetic, particularly divisions?**

- **Ans:** Since no integer divisions are present in the code, this issue does not apply.

Category D: Comparison Errors

1. **Are there any comparisons between variables having different data types?**
   - No data type mismatches in comparisons have been found.

2. **Are there any mixed-mode comparisons or comparisons between variables of different lengths?**
    - No such errors appear in the code. All variables being compared seem to be of compatible types.
3. **Are the comparison operators correct?**
    - Yes, all comparison operators (`<`, `<=`, `>`, `>=`) are used correctly.
4. **Does each Boolean expression state what it is supposed to state?**
    - There are no mistakes in the logic. For instance, `if (len === 0)` is correctly checking for empty arrays.
5. **Are the operands of a Boolean operator Boolean?**
    - Yes, the Boolean expressions use valid comparisons. There are no erroneous combinations of comparison and Boolean operators.
6. **Are there any comparisons between fractional or floating-point numbers?**
    - There are no floating-point comparisons in the code provided. The logic operates mainly on arrays and integers.
7. **For expressions with more than one Boolean operator, are assumptions about evaluation order correct?**
    - There are no complex Boolean expressions that combine `&&` and `||` operators, so no issues regarding evaluation precedence.
8. **Does the way the compiler evaluates Boolean expressions affect the program?**
    - No such cases have been found. The logic does not involve operations that could result in short-circuit evaluation issues.

Category E: Control-Flow Errors

1. **If the program contains a multiway branch such as a computed GO TO, can the index variable ever exceed the number of branch possibilities?**
    - No multiway branches like `GO TO` are present in JavaScript. The control flow uses `if-else` and simple loops, and the conditions are adequately checked.
2. **Will every loop eventually terminate?**
    - Yes, loops are well-constructed. For instance, in the `for` loops, termination conditions are clear (`for (let i = 0; i < len; i++)`).
3. **Will the program, module, or subroutine eventually terminate?**
    - Yes, the loops and recursion will terminate based on the given conditions. No infinite loops have been detected.
4. **Is it possible that, because of the conditions upon entry, a loop will never execute?**
    - The conditions allow all loops to execute at least once. There are no zero-iteration loops, such as `for (i == x; i <= z; i++)`, where `x > z`.
5. **For a loop controlled by both iteration and a Boolean condition, what are the consequences of loop fall-through?**
    - No loops in this code combine iteration and Boolean conditions. The loops are either based on indices or simple array traversal.

6. **Are there any off-by-one errors, such as one too many or too few iterations?**
    - No off-by-one errors have been detected. The loops over arrays correctly account for zero-based indexing.
7. **If the language contains statement groups or blocks, is there a closing bracket for each open one?**
    - All code blocks have matching brackets.
8. **Are there any non-exhaustive decisions?**
    - Decisions appear to be exhaustive. For instance, in `removeMiddleItem()`, all conditions are considered when altering the array.

Category F: Interface Errors

1. **Does the number of parameters received by this module equal the number of arguments sent by each of the calling modules?**
    - No parameter mismatches have been detected. All function calls pass the correct number of arguments.
2. **Do the attributes of each parameter match the attributes of each corresponding argument?**
    - Yes, the function parameters and arguments match in both type and quantity.
3. **Does the units system of each parameter match the units system of each corresponding argument?**
    - There are no unit systems in play, but data types align correctly.
4. **Does the number of arguments transmitted by this module to another module equal the number of parameters expected by that module?**
    - Yes, the number of arguments matches in function calls.
5. **Do the attributes of each argument transmitted to another module match the attributes of the corresponding parameter in that module?**
    - Yes, argument and parameter attributes are correctly aligned.
6. **Does the units system of each argument transmitted to another module match the units system of the corresponding parameter in that module?**
    - No unit mismatches exist, as the code deals with arrays and numbers.
7. **If built-in functions are invoked, are the number, attributes, and order of the arguments correct?**
    - Yes, built-in JavaScript functions such as `splice()` and `forEach()` are used correctly with the appropriate number and type of arguments.
8. **Does a subroutine alter a parameter that is intended to be only an input value?**
    - No such side effects are observed. Parameters are handled properly within the function scopes.
9. **If global variables are present, do they have the same definition and attributes in all modules that reference them?**
    - No global variables are used. All variables have localized scope within the functions.

Category G: Input/Output Errors

1. **If files are explicitly declared, are their attributes correct?**
    - No file handling is present in the code.
2. **Are the attributes on the file's OPEN statement correct?**
    - Not applicable, as there is no file I/O in the code.
3. **Is there sufficient memory available to hold the file your program will read?**
    - Not applicable.
4. **Have all files been opened before use?**
    - Not applicable.
5. **Have all files been closed after use?**
    - Not applicable.
6. **Are end-of-file conditions detected and handled correctly?**
    - Not applicable.
7. **Are I/O error conditions handled correctly?**
    - Not applicable.
8. **Are there spelling or grammatical errors in any text that is printed or displayed by the program?**
    - There is no printed or displayed text in the code.

Category H: Other Checks

1. **If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.**
    - No variables are unused in the code. All declared variables are referenced.
2. **If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.**
    - JavaScript doesn't enforce strict type declarations, so no attribute mismatch issues are present.
3. **If the program compiled successfully, but the computer produced warning or informational messages, check each one carefully.**
    - No warnings are expected in this code, assuming it runs in a modern JavaScript environment.
4. **Is the program or module sufficiently robust? Does it check its input for validity?**
    - No explicit input validation is present, which could be improved. For example, checks for array length could avoid potential errors in edge cases.
5. **Is there a function missing from the program?**
    - The functionality seems complete for the tasks it is designed to handle.

Program Inspection

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

Based on the inspection of the code provided, the following errors were identified:

- **Data-Declaration Errors:**
  - Implicit Variable Declaration: If a variable is used without being explicitly declared, it can lead to confusion. This might not apply directly in the code, but it's worth noting that JavaScript allows undeclared variables, which could lead to issues.
- **Computation Errors:**
  - Integer Division: When dividing integers in JavaScript, the result can lead to unexpected behavior if not carefully handled. Although not directly visible in the provided code, this could be a concern if integers were to be involved in future modifications.
- **Control-Flow Errors:**
  - Potential Infinite Loops: The code appears to be structured well, but care must be taken to ensure loops have correct exit conditions, especially when dealing with dynamic data like arrays.
- **Input/Output Errors:**
  - Lack of Input Validation: The functions do not validate input parameters. For instance, they assume the input array is valid without checks for null values or unexpected data types.

Total Identified Errors: 4

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

- The Data-Declaration Errors category would be more effective for this code. Ensuring that all variables are explicitly declared and understood can prevent many common issues, especially in a dynamically typed language like JavaScript. This helps in maintaining clarity in the code, reducing ambiguity regarding variable scopes and types.

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

- Using program inspection, it is challenging to identify logic errors that arise during runtime or specific edge cases not covered by tests. For instance, conditions where a function might not behave as intended due to unforeseen data inputs or states can be difficult to detect through inspection alone. Moreover, performance issues related to time complexity or memory usage may also go unnoticed.

4. Is the program inspection technique worth applicable?

- Yes, the program inspection technique is definitely worth applying. It provides a structured way to evaluate code and helps identify a range of potential errors before the code is executed. This proactive approach can save time and resources by catching issues early, fostering better code quality, and promoting more robust software development practices. However, it should be complemented with other testing techniques, such as unit testing and integration testing, to cover a broader range of errors, including runtime and logical errors.

CODE :

LOC : 420

```
/* ============================== QUESTION 1
=============================================
    Create a function which solve quadratic equation ax2 + bx + c = 0.
    A quadratic equation may have one, two or no solution.
    The function should return a set of the solution(s).

=========================================================================
==================== */
const solveQuadratic = (a = 1, b = 0, c = 0) => {
  if (a === 0) {
    return 'Not a quadratic equation';
  }
  const determinate = b ** 2 - 4 * a * c;
  const solnSet = new Set();

  let root1, root2;
  if (determinate === 0) {
    root1 = -b / (2 * a);
    solnSet.add(root1);
  } else if (determinate > 0) {
    root1 = ((-b + Math.sqrt(determinate)) / 2) * a;
    root2 = ((-b - Math.sqrt(determinate)) / 2) * a;
    solnSet.add(root1);
    solnSet.add(root2);
  } else {
  }
  return solnSet;
};
console.log(solveQuadratic()); //Set(1) {0}
console.log(solveQuadratic(1, 2, 3)); // Set(0) {}
console.log(solveQuadratic(1, 4, 4)); //Set(1) {-2}
console.log(solveQuadratic(1, -1, -2)); //{2, -1}
console.log(solveQuadratic(1, 7, 12)); //Set(2) {-3, -4}
console.log(solveQuadratic(1, 0, -4)); //{2, -2}
console.log(solveQuadratic(1, -1, 0)); //{1, 0}
/* ============================== QUESTION 2
=============================================
    Create a function called isPrime which check if a number is prime or not.

=========================================================================
====================*/
const isPrime = n => {
  let prime = false;
  if (n < 2) {
    prime = false;
```

```
  }
  if (n === 2) {
    prime = true;
  }
  for (let i = 2; i < n; i++) {
    if (n % i === 0) {
      prime = false;
      break;
    } else {
      prime = true;
    }
  }
  return prime;
};

console.log(isPrime(0)); // false
console.log(isPrime(1)); // false
console.log(isPrime(2)); // true
console.log(isPrime(3)); // true
console.log(isPrime(5)); // true

/* ================================ QUESTION 3
================================================
    Write a function rangeOfPrimes.
    It takes two parameters, a starting number and an ending number .
    The function returns an object with properties primes and count.
    The primes value is an array of prime numbers and
    count value is the number of prime numbers in the array.
     See example

=========================================================================
==================== */

const rangeOfPrimes = (start, end) => {
  const primes = [];
  for (let i = start; i <= end; i++) {
    if (isPrime(i)) {
      primes.push(i);
    }
  }
  const count = primes.length;
  return { primes, count };
};
console.log(rangeOfPrimes(2, 11)); //{primes:[2, 3, 5, 7, 11], count:5}
console.log(rangeOfPrimes(50, 60)); //{primes:[53, 59], count:2}
console.log(rangeOfPrimes(95, 107)); //{primes:[97, 101, 103, 107], count:4}
/* ================================ QUESTION 4
================================================
    Create a function called isEmpty which check if the parameter is empty.
    If the parameter is empty, it returns true else it returns false.
```

```
========================================================================
==================== */
const isEmpty = value => {
  return (
    value === null ||
    value === undefined ||
    (typeof value === 'string' && value.trim().length === 0) ||
    (value.construtor === Array && value.length === 0) ||
    (typeof value === 'object' && Object.keys(value).length === 0)
  );
};
console.log(isEmpty('')); // true
console.log(isEmpty(' ')); // true
console.log(isEmpty('Asabeneh')); // false
console.log(isEmpty([])); // true
console.log(isEmpty(['HTML', 'CSS', 'JS'])); // false;
console.log(isEmpty({})); //true
console.log(isEmpty({ name: 'Asabeneh', age: 200 })); // false

/* ================================ QUESTION 5
================================================
   a. Create a function called reverse which take a parameter and it returns the reverse of the
parameter.
   Don't use the built in reverse method.
  b. Create a function called isPalindrome which check if a parameter is a palindrome or not.
    Use the function from a to reverse words.
========================================================================
==================== */
//5a
const reverse = value => {
  let reversed = '';
  if (typeof value === 'number') {
    const formatedValue = value.toString();
    const len = formatedValue.length;
    for (let i = len - 1; i >= 0; i--) {
      reversed += formatedValue[i];
    }
  } else if (typeof value === 'string') {
    const formatedValue = value
      .trim()
      .replace(/\W/g, '')
      .toLowerCase();
    let len = formatedValue.length;
    for (let i = len - 1; i >= 0; i--) {
      reversed += formatedValue[i];
    }
  } else {
    return 'Not a valid parameter';
  }
```

```
  return reversed;
};
console.log(reverse('car'));
console.log(reverse('Cat '));
console.log(reverse('Tuna nut.'));
console.log(reverse('A nut for a jar of tuna.'));
// end of 5a

//5b
const isPalindrome = value => {
  let isPalindrome = false;
  if (typeof value === 'number') {
    const formatedValue = value.toString();
    if (reverse(formatedValue) === formatedValue) {
      isPalindrome = true;
    }
  } else if (typeof value === 'string') {
    const formatedValue = value
      .trim()
      .replace(/\W/g, '')
      .toLowerCase();
    if (reverse(formatedValue) === formatedValue) {
      isPalindrome = true;
    }
  } else {
    return 'Not a valid parameter';
  }
  return isPalindrome;
};
console.log(isPalindrome('Anna')); //true
console.log(isPalindrome(121)); //true
console.log(isPalindrome('Noon')); //true
console.log(isPalindrome('Asa ')); //true
console.log(isPalindrome('Asab')); //false
console.log(isPalindrome('cat')); //false
console.log(isPalindrome('Tuna nut.')); // true
console.log(isPalindrome('A nut for a jar of tuna.')); // true
console.log(isPalindrome('A man, a plan, a canal. Panama'));
/* ============================== QUESTION 6
=============================================
    Create a function called countPalindrowWords which counts the number of palindrome
words in
   the palindoromeWords array or in any array.

==========================================================================
==================== */
const words = [
  'Anna',
  'Asa',
  'Civic',
```

```
  'common',
  'Kayak',
  'Level',
  'Madam',
  'Mom',
  'Noon ',
  'Rotor',
  'Sagas ',
  'Solar',
  'Stats',
  'Tenet ',
  'Wow'
];

const countPalindrowWords = arr => {
  let count = 0;
  for (const word of arr) {
    if (isPalindrome(word)) {
      count++;
    }
  }
  return count;
};
console.log(countPalindrowWords(words)); // 13

/* ================================ QUESTION 7
=============================================
   Count the number of palindrome words in the following sentence.

================================================================================
==================== */
const sentence = `He met his mom at noon and she was watching an epsoide of the
begninging of her Solos. Her tenet helped her to level up her stats. After that he went to kayak
driving Civic Honda.`;
const countPalindrowWords2 = sent => {
  const words = sent.split(' ');
  const palindromeWords = [];
  let count = 0;
  for (const word of words) {
    if (isPalindrome(word)) {
      count++;
      palindromeWords.push(word);
    }
  }
  return { count, words: palindromeWords };
};
console.log(countPalindrowWords2(sentence));
//{count:8, words:["mom", "noon", "Solos.", "tenet", "level", "stats.", "kayak", "Civic"]}

const users = [
```

```
  {
    _id: 'ab12ex',
    username: 'Alex',
    email: 'alex@alex.com',
    password: '123123',
    createdAt: '17/05/2019 9:00 AM',
    isLoggedIn: false
  },
  {
    _id: 'fg12cy',
    username: 'Asab',
    email: 'asab@asab.com',
    password: '123456',
    createdAt: '17/05/2019 9:30 AM',
    isLoggedIn: true
  },
  {
    _id: 'zwf8md',
    username: 'Brook',
    email: 'brook@brook.com',
    password: '123111',
    createdAt: '17/05/2019 9:45 AM',
    isLoggedIn: true
  },
  {
    _id: 'eefamr',
    username: 'Martha',
    email: 'martha@martha.com',
    password: '123222',
    createdAt: '17/05/2019 9:50 AM',
    isLoggedIn: false
  },
  {
    _id: 'ghderc',
    username: 'Thomas',
    email: 'thomas@thomas.com',
    password: '123333',
    createdAt: '17/05/2019 10:00 AM',
    isLoggedIn: false
  }
];

const products = [
  {
    _id: 'eedfcf',
    name: 'mobile phone',
    description: 'Huawei Honor',
    price: 200,
    ratings: [{ userId: 'fg12cy', rate: 5 }, { userId: 'zwf8md', rate: 4.5 }],
    likes: []
```

```
  },
  {
    _id: 'aegfal',
    name: 'Laptop',
    description: 'MacPro: System Darwin',
    price: 2500,
    ratings: [],
    likes: ['fg12cy']
  },
  {
    _id: 'hedfcg',
    name: 'TV',
    description: 'Smart TV:Procaster',
    price: 400,
    ratings: [{ userId: 'fg12cy', rate: 5 }],
    likes: ['fg12cy']
  }
];

/* ============================== QUESTION 8
=============================================
    Imagine you are getting the above users collection from a MongoDB database.
    a. Create a function called signUp which allows user to add to the collection.
    If user exists, inform the user that he has already an account.
    b. Create a function called signIn which allows user to sign in to the application

==========================================================================
=================== */

const randomId = () => {
  const numbersLetters =
'0123456789abcdefghijklmnopqrstuvwzyzABCDEFGHIJKLMNOPQRSTUVWXYZ'.split(
    ''
  );
  let randId = '';
  let randIndex;
  for (let i = 0; i < 6; i++) {
    randIndex = Math.floor(Math.random() * numbersLetters.length);
    randId += numbersLetters[randIndex];
  }
  return randId;
};
const newUser = {
  _id: randomId(),
  username: 'Asabeneh',
  email: 'asabeneh@asabeneh.com',
  password: '123123123',
  createdAt: new Date(),
  isLoggedIn: false
};
```

```
const signUp = () => {
  const { email } = newUser;
  for (const user of users) {
    if (user['email'] == email) {
      return 'An email has already exist. Please log in!';
    }
  }
  users.push(newUser);
  return 'You have successfully signed up!';
};
console.log(users);
console.log(signUp(newUser));
console.log(signUp(newUser));
console.log(users);
const currentUser = {
  email: 'asabeneh@asabeneh.com',
  password: '123123123'
};
const signIn = user => {
  let found = false;
  const { email, password } = user;
  for (let i = 0; i < users.length; i++) {
    if (users[i]['email'] === email && users[i]['password'] === password) {
      users[i].isLoggedIn = true;
      return 'Successfully logged in';
    }
  }
  if (!found) {
    return 'Use does not exist';
  }
};
console.log(signIn(currentUser));
console.log(users);
console.log(signIn({ email: 'asab@asab.com', password: '123456' }));
/* ============================== QUESTION 9
=============================================
    The products array has three elements and each of them has six properties.
    a. Create a function called rateProduct which rates the product
    b. Create a function called averageRating which calculate the average rating of a product

==========================================================================
==================== */
//a
const rateProduct = (productId, userId, ratingPoint) => {
  let found = false;
  for (let i = 0; i < products.length; i++) {
    if (products[i]._id === productId) {
      for (let j = 0; j < products[i].ratings.length; j++) {
        if (products[i].ratings[j].userId === userId) {
          const rate = { userId, rate: ratingPoint };
```

```
          products[i].ratings[j].rate = ratingPoint;
          found = true;
          break;
        }
      }
      if (!found) {
        products[i].ratings.push({ userId, rate: ratingPoint });
      }
    }
  }
};
console.log(products);
rateProduct('eedfcf', 'fg12cy', 5);
rateProduct('aegfal', 'fg12cy', 2.5);
rateProduct('aegfal', 'fg12cy', 2.0);
console.log(products);
//b
const averageRating = productId => {
  let sum = 0;
  let len; // number of ratings
  for (let i = 0; i < products.length; i++) {
    if (products[i]._id === productId) {
      len = products[i].ratings.length;
      for (let j = 0; j < len; j++) {
        if (len === 0) {
          return 0;
        } else {
          sum += products[i].ratings[j].rate;
        }
      }
    }
  }
  console.log(len);
  return sum / len;
};
console.log(averageRating('eedfcf'));

/* ================================ QUESTION 10
=============================================
  Create a function called likeProduct.
  This function will helps to like to the product if it is not liked and remove like if it was liked.

=========================================================================
==================== */
const likeProduct = (productId, userId) => {
  for (let i = 0; i < products.length; i++) {
    if (products[i]._id === productId) {
      const likes = products[i].likes;
      const index = products[i].likes.indexOf(userId);
      if (index !== -1) {
```

```
        products[i].likes.splice(index, 1);
      } else {
        products[i].likes.push(userId);
      }
    }
  }
};
console.log(likeProduct('aegfal', 'fg12cy'));
console.log(likeProduct('eedfcf', 'fg12cy'));
```

PROGRAM INSPECTION

CATEGORY A : Data Reference Errors

[1] Does a referenced variable have a value that is unset or uninitialized? This probably is the most frequent programming error; it occurs in a wide variety of circumstances. For each reference to a data item (variable, array element, field in a structure), attempt to "prove" informally that the item has a value at that point.

let found = false; in the signIn function
Ensure found is set to true when a user is found.

let len; in the averageRating function
len should be initialized before being used in calculations.

[2] For all array references, is each subscript value within the defined bounds of the corresponding dimension?

For (const word of arr) {...} in countPalindrowWords

Ensure that arr is non-empty before this loop runs. If arr is empty, it will still work fine but ensure valid inputs are passed.

[3] For all array references, does each subscript have an integer value? This is not necessarily an error in all languages, but it is a dangerous practice.

There are no explicit non-integer subscripts in this code.

[4] For all references through pointer or reference variables, is the referenced memory currently allocated? This is known as the "dangling reference" problem. It occurs in situations where the lifetime of a pointer is greater than the lifetime of the referenced memory. One situation occurs where a pointer references a local variable within a procedure, the pointer value is assigned to an output parameter or a global variable, the procedure returns (freeing the referenced location), and later the program attempts to use the pointer value. In a manner similar to checking for the prior errors, try to prove informally that, in each reference using a pointer variable, the reference memory exists.

JavaScript does not use pointers like C/C++. However, global objects or array elements that might be modified elsewhere.

[5] When a memory area has alias names with differing attributes, does the data value in this area have the correct attributes when referenced via one of these names? Situations to look for are the use of the EQUIVALENCE statement in FORTRAN, and the REDEFINES clause in COBOL. As an example, a FORTRAN program contains a real variable A and an integer variable B; both are made aliases for the same memory area by using an EQUIVALENCE statement. If the program stores a value into A and then references variable B, an error is likely present since the machine would use the floating-point bit representation in the memory area as an integer.

Not applicable in JavaScript. There are no aliasing issues typical of FORTRAN or COBOL.

[6] Does a variable's value have a type or attribute other than what the compiler expects? This situation might occur where a C, C++, or COBOL program reads a record into memory and references it by using a structure, but the physical representation of the record differs from the structure definition.

Ensure that n in isPrime is always a number.

[7] Are there any explicit or implicit addressing problems if, on the machine being used, the units of memory allocation are smaller than the units of memory addressability? For instance, in some environments, fixed-length bit strings do not necessarily begin on byte boundaries, but addresses only point to byte boundaries. If a program computes the address of a bit string and later refers to the string through this address, the wrong memory location may be referenced. This situation also could occur when passing a bit-string argument to a subroutine.

Not applicable to JavaScript as it automatically manages memory.

[8] If pointer or reference variables are used, does the referenced memory location have the attributes the compiler expects? An example of such an error is where a C++pointer upon which a data structure is based is assigned the address of a different data structure.

Not applicable in JavaScript, but ensure the structures of objects like users and products are as expected throughout the code.

[9] If a data structure is referenced in multiple procedures or subroutines, is the structure defined identically in each procedure?

Make sure that object structures are consistently defined, such as:
Definition of the users array.
Definition of the products array.

[10] When indexing into a string, are the limits of the string off by-one errors in indexing operations or in subscript references to arrays?

Loop in isPrime:

```
for (let i = 2; i < n; i++) {
```

Consider checking against Math.sqrt(n) for efficiency.

[11] For object-oriented languages, are all inheritance requirements met in the implementing Class?

Not applicable in current code, as it is not using classes or inheritance.

CATEGORY B: Data-Declaration Errors

[1] Have all variables been explicitly declared? A failure to do so is not necessarily an error, but it is a common source of trouble. For instance, if a program subroutine receives an array parameter, and fails to define the parameter as an array (as in a DIMENSION statement, for example), a reference to the array (such as C=A (I)) is interpreted as a function call, leading to the machine's attempting to execute the array as a program. Also, if a variable is not explicitly declared in an inner procedure or block, is it understood that the variable is shared with the enclosing block?

const solveQuadratic = (a = 1, b = 0, c = 0) => {...}
Variables a, b, and c are declared with default values. Ensure you handle cases where they might be undefined when invoked.

const isPrime = n => {...}
n is not explicitly typed, which is fine in JavaScript, but make sure you validate input types.

[2] If all attributes of a variable are not explicitly stated in the declaration, are the defaults well

understood? For instance, the default attributes received in Java are often a source of surprise.

Default values for parameters (a = 1, b = 0, c = 0) are clear but should be documented to ensure maintainability.

JavaScript defaults to undefined for undeclared variables, which could lead to confusion if not handled properly.

[3] Where a variable is initialized in a declarative statement, is it properly initialized? In many languages, initialization of arrays and strings is somewhat complicated and, hence, error prone.

let found = false;
This variable is correctly initialized.

let len;
len is declared but not initialized until used, which could lead to unexpected behavior. Make sure to initialize it before use.

[4] Is each variable assigned the correct length and data type?

Ensure that all functions expect parameters of the right type:
const averageRating = productId => {...}
Confirm that productId is always a valid string.

[5] Is the initialization of a variable consistent with its memory type?

JavaScript automatically manages types and memory, so initialization like let count = 0; (Line 152) is appropriate.

[6] Are there any variables with similar names (VOLT and VOLTS, for example)? This is not necessarily an error, but it should be seen as a warning that the names may have been confused somewhere within the program.

The variable names root1 and root2 in solveQuadratic could be confused with other similar variables in broader contexts.

let found = false; could be confused with a similar found variable if there are nested scopes. Avoid using the same name in different scopes.

CATEGORY C : Computation Errors

[1] Are there any computations using variables having inconsistent (such as non-arithmetic) data types?

const determinate = b ** 2 - 4 * a * c;
Ensure that a, b, and c are numeric. If any are strings, it could lead to unexpected behavior in arithmetic operations.

[2] Are there any mixed-mode computations? An example is the addition of a floating-point variable to an integer variable. Such occurrences are not necessarily errors, but they should be explored carefully to ensure that the language's conversion rules are understood.
Consider the following Java snippet showing the rounding error that can occur when working with integers:
int x = 1;
int y = 2;
int z = 0;
z = x/y;
System.out.println ("z = " + z);

OUTPUT:
z = 0

(string vs. number). If b is a string, this will lead to concatenation rather than arithmetic addition.

[3] Are there any computations using variables having the same data type but different lengths?

This is less of a concern in JavaScript due to dynamic typing, but ensure that you consistently use similar numeric types (integers vs. floating-point) across computations.

[4] Is the data type of the target variable of an assignment smaller than the data type or result of the right-hand expression?

let count = primes.length;
Ensure that count is used consistently as a number, especially if later operations expect a specific type.

[5] Is an overflow or underflow expression possible during the computation of an expression? That is, the end result may appear to have valid value, but an intermediate result might be too big or too small for the programming language's data types.

Consider scenarios where b is large enough that b ** 2 exceeds safe integer limits, although JavaScript can handle large numbers relatively well. Still, be mindful of potential precision issues.

[6] Is it possible for the divisor in a division operation to be zero?

root1 = -b / (2 * a);
Ensure that a is not zero to prevent division by zero errors. Add a check before this computation.

[7] If the underlying machine represents variables in base-2 form, are there any sequences of the resulting inaccuracy? That is, 10 x 0.1 is rarely equal to 1.0 on a binary machine.

JavaScript generally handles floating-point arithmetic correctly, but be aware of precision issues in calculations, especially involving fractions (e.g., 0.1).

[8] Where applicable, can the value of a variable go outside the meaningful range? For example, statements assigning a value to the variable PROBABILITY might be checked to ensure that the assigned value will always be positive and not greater than

Ensure that input values for a, b, and c are within expected ranges. For example, a should not be zero to avoid misinterpretation as a non-quadratic equation.

[9] For expressions containing more than one operator, are the assumptions about the order of evaluation and precedence of operators correct?

Be cautious of operator precedence in mixed operations. For example, the order of operations in b ** 2 - 4 * a * c is correctly prioritized, but ensure this is clear to anyone reading the code.

[10] Are there any invalid uses of integer arithmetic, particularly divisions? For instance, if i is an integer variable, whether the expression 2*i/2 == i depends on whether i has an odd or an even value and whether the multiplication or division is performed first…

When dealing with integers, ensure that calculations are valid, particularly if you later treat results as integers. In your case, since you use floating-point numbers, be cautious about rounding errors.

CATEGORY D : Comparison Errors

[1] Are there any comparisons between variables having different data types, such as comparing a character string to an address, date, or number?

Check if comparing variables of different types (e.g., a number to a string). This can lead to unexpected behavior. Ensure that both sides of the comparison are of the same type.

[2] Are there any mixed-mode comparisons or comparisons between variables of different lengths? If so, ensure that the conversion rules are well understood.

For instance, comparing a string representation of a number to an actual number can yield unexpected results.
If comparing variables of different lengths or types, then make sure the conversion rules are clear.

[3] Are the comparison operators correct? Programmers frequently confuse such relations as at most, at least, greater than, not less than, less than or equal.

Review the logic of your comparisons to confirm that you're using the correct operators (e.g., <, <=, >, >=). It's easy to confuse these, especially in complex conditions.

[4] Does each Boolean expression state what it is supposed to state? Programmers often make mistakes when writing logical expressions involving and, or, and not.

Ensure Boolean expressions accurately reflect your intentions. For instance, verify that conditions involving and and or are constructed logically and convey the correct logic.

[5] Are the operands of a Boolean operator Boolean? Have comparison and Boolean operators been erroneously mixed together? This represents another frequent class of mistakes. Examples of a few typical mistakes are illustrated here. If you want to determine whether i is between 2 and 10, the expression 2<i<10 is incorrect; instead, it should be (2<;i) && (i<10). If you want to determine whether i is greater than x or y, i>x||y is incorrect; instead, it should be (i>x)||(i>y). If you want to compare three numbers for equality, if(a==b==c) does something quite different. If you want to test the mathematical relation x>y>z, the correct expression is (x>y) && (y>z).

Check that the operands used in Boolean operations are indeed Boolean values. Be cautious of incorrect constructs like 2 < i < 10. Instead, it should be written as (2 < i) && (i < 10).

[6] Are there any comparisons between fractional or floating- point numbers that are represented in base-2 by the underlying machine? This is an occasional source of errors because of truncation and base-2 approximations of base-10 numbers.

During comparing floating-point numbers, be mindful of potential inaccuracies due to base-2 representation. It's often better to check if the numbers are "close enough" rather than exactly equal.

[7] For expressions containing more than one Boolean operator, are the assumptions about the order of evaluation and the precedence of operators correct? That is, if you see an expression such as (if((a==2) && (b==2) || (c==3)), is it well understood whether the and or the or is performed first?

For complex Boolean expressions, ensure you understand the order of operations. For example, in an expression like (a == 2 && (b == 2 || c == 3)), confirm that the || operator is correctly grouped.

[8] Does the way in which the compiler evaluates Boolean expressions affect the program? For instance, the statement if((x==0 && (x/y)>z) may be acceptable for compilers that end the test as soon as one side of an and is false, but may cause a division-by-zero error with other compilers.

Be cautious with expressions that could lead to errors, such as if ((x == 0 && (x / y) > z). Different compilers may handle short-circuiting differently, potentially causing division by zero if not handled correctly.

CATEGORY E : Control-Flow Errors

[1] If the program contains a multiway branch such as a computed GO TO, can the index.

The code does not use multiway branching, but if it did, care should be taken to ensure that any index variable used for branching does not exceed the number of available options.

[2] Will every loop eventually terminate? Devise an informal proof or argument showing that each loop will terminate.

All loops in the code (like for and while) are structured to eventually terminate. The loop conditions are clear, ensuring that the iterations will stop based on specified criteria.

[3] Will the program, module, or subroutine eventually terminate?

Each function in the code is designed to terminate properly by reaching a return statement. There are no infinite loops or uncontrolled recursion, so each module or subroutine will conclude as intended.

[4] Is it possible that, because of the conditions upon entry, a loop will never execute? If so, does this represent an over- sight? For instance, if you had the following loops headed by the following statements:

```
for (i==x ; i<=z; i++) { ... }

while (NOTFOUND) { ... }
```

what happens if NOTFOUND is initially false or if x is greater than z?

The code does not contain loops that could potentially never execute due to initial conditions. For example, in the for loop or while loop scenarios, the initial conditions will always allow execution as per the logic defined.

[5] For a loop controlled by both iteration and a Boolean condition (a searching loop, for example) what are the consequences of loop fall-through? For example, for the psuedo-code loop headed by DO I=1 to TABLESIZE WHILE (NOTFOUND) what happens if NOTFOUND never becomes false?

The code does not demonstrate a scenario where a loop might fall through indefinitely. However, if a loop condition (like a search loop) does not become false, it would lead to an infinite loop, which should be avoided by ensuring the condition can change during iterations.

[6] Are there any off-by-one errors, such as one too many or too few iterations? This is a common error in zero-based loops. You will often forget to count "0" as a number. For example, if you want to create Java code for a loop that counted to 10, the following would be wrong, as it counts to 11:

```
for (int i=0; i<=10;i++) {
System.out.println(i);
}
```

Correct, the loop is iterated 10 times:

```
for (int i=0; i<;=9;i++) {
System.out.println(i);
```

There are no apparent off-by-one errors in the loop iterations. All loops correctly count their iterations based on intended ranges. Care has been taken to ensure boundaries are correctly defined, especially in zero-based indexing.

[7] If the language contains a concept of statement groups or code blocks (e.g., do-while or {...}), is there an explicit while for each group and do the do's correspond to their

appropriate groups? Or is there a closing bracket for each open bracket? Most modern compilers will complain of such mismatches.

The code follows proper syntax with matched braces for code blocks. Each opening brace has a corresponding closing brace, which prevents syntax errors and ensures logical grouping of statements.

[8] Are there any non-exhaustive decisions? For instance, if an input parameter's expected values are 1, 2, or 3, does the logic assume that it must be 3 if it is not 1 or 2? If so, is the assumption valid?

The code accounts for various input cases. For example, the prime-checking function handles edge cases like numbers less than 2. All expected inputs have been considered, and no assumptions about unhandled cases exist.

CATEGORY F : Interface Errors

[1] Does the number of parameters received by this module equal the number of arguments sent by each of the calling modules? Also, is the order correct?

Each function correctly receives the number of parameters that match the number of arguments provided in the calls. For instance, solveQuadratic(a, b, c) is called with three arguments, and isPrime(n) is called with one argument, aligning correctly.

[2] Do the attributes (e.g., data type and size) of each parameter match the attributes of each corresponding argument?

The data types and expected sizes of parameters align with the types of the arguments passed. For example, numeric parameters are expected in functions like isPrime and solveQuadratic, and they receive numbers consistently.

[3] Does the units system of each parameter match the units system of each corresponding argument? For example, is the parameter expressed in degrees but the argument expressed in radians?

The code does not involve any units systems that could differ. All parameters and arguments are treated as numbers without any unit conversions, so this aspect is consistent.

[4] Does the number of arguments transmitted by this module to another module equal the number of parameters expected by that module?

When functions are called within the modules, the number of arguments transmitted matches the expected parameters of the receiving modules. This is true for all function calls in the code.

[5] Do the attributes of each argument transmitted to another module match the attributes of the corresponding parameter in that module?

The attributes of arguments passed between functions align properly. For example, when calling `rateProduct(productId, userId, ratingPoint)`, all arguments are of the expected types.

[6] Does the units system of each argument transmitted to another module match the units system of the corresponding parameter in that module?

As with parameters, there are no unit discrepancies among transmitted arguments across functions. All values remain numerical without the introduction of differing units.

[7] If built-in functions are invoked, are the number, attributes, and order of the arguments correct?

All built-in functions (like `Math.sqrt`, `console.log`, etc.) are called with the correct number of arguments and in the appropriate order, following the expected signature.

[8] Does a subroutine alter a parameter that is intended to be only an input value?

The code does not alter parameters that are meant to be inputs. Parameters are treated as inputs only, and any modifications occur on local copies rather than altering the input directly.

[9] If global variables are present, do they have the same definition and attributes in all modules that reference them?

The global variables users and products are referenced consistently across the functions. Their structure and attributes remain the same throughout the code.

CATEGORY G : Input / Output Errors

[1] If files are explicitly declared, are their attributes correct?

The code does not explicitly declare any files for input/output. Hence, this point is not applicable.

[2] Are the attributes on the file's OPEN statement correct?

Since no files are opened or closed in the code, this point is also not applicable.

[3] Is there sufficient memory available to hold the file your program will read?

There are no file read operations, so this aspect does not apply to the current code.

[4] Have all files been opened before use?

No files are utilized, so this point is not relevant.

[5] Have all files been closed after use?

Again, since there are no file operations, this is not applicable.

[6] Are end-of-file conditions detected and handled correctly?

There are no file reads that could encounter end-of-file conditions. This point is not relevant.

[7] Are I/O error conditions handled correctly?

As there are no I/O operations, error handling for input/output is not applicable.

[8] Are there spelling or grammatical errors in any text that is printed or displayed by the program?

There are a few minor spelling errors in the comments and function implementations.

"construtor" should be "constructor" in the isEmpty function.

The term "palindrow" appears multiple times instead of "palindrome" in comments and function names.

CATEGORY H : Other Checks

[1] If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.

The code includes several variables and functions. A thorough inspection of the code indicates that most variables are used appropriately. However, found in the signIn function is never set to true in the case where the user exists, leading to a possible logic error.

[2] If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.

The attributes of variables seem to be in line with expectations. All variables appear to have appropriate types based on their usage (e.g., strings for usernames, numbers for ratings). There are no unexpected default attributes.

[3] If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully. Warning messages are indications that the compiler suspects that you are doing something of questionable validity; all of these suspicions should be reviewed. Informational messages may list undeclared variables or language uses that impede code optimization.

Since this analysis is based on code review rather than actual compilation, there may be potential warnings:

In the isPrime function, the logic could lead to a misleading return value (e.g., if n < 2, it should return false immediately).

In the averageRating function, if no ratings exist, len may not be defined if the product ID does not match. This could result in a division by zero error.

[4] Is the program or module sufficiently robust? That is, does it check its input for validity?

The program checks some inputs, such as ensuring that the user does not sign up with an existing email. However, other areas lack sufficient input validation:

The rateProduct function does not check if ratingPoint is within a valid range (e.g., 1 to 5).

Functions like solveQuadratic assume valid inputs without checking for non-numeric values.

[5] Is there a function missing from the program?

The code covers a range of functionalities but could benefit from:

A function to validate inputs more rigorously, especially for user registration and product ratings.

Error handling functions to manage exceptions more effectively, such as handling invalid user credentials or product IDs.

PROGRAM INSPECTION :

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

Total Errors Identified: 7

- Logic Error in isPrime: The loop incorrectly sets prime to true multiple times without proper checks. It should return false immediately when a divisor is found.
- Unused Variable: The variable found in the signIn function is declared but never used effectively, leading to misleading return values.
- Uninitialized len in averageRating: If no ratings exist for a product, the variable len is undefined when used in division.
- Incorrect Logic in reverse: It does not handle non-string and non-number types properly, returning "Not a valid parameter" instead of gracefully managing the input.
- Potential for Undefined Behavior in likeProduct: If the product ID does not exist, the function silently does nothing without error handling.
- Inefficient Input Validation: Functions like rateProduct and signUp do not adequately check for valid inputs (e.g., negative ratings).
- Incorrect Condition in isEmpty: The condition (value.constructor === Array) should be Array.isArray(value) to correctly check if a value is an array.

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

Effective Category: Logic Errors and Input/Output Errors would be particularly effective for this code. Since many functions rely on proper logical flow and valid inputs, focusing on these categories would help catch the most critical errors affecting functionality.

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

Runtime Errors: Errors that occur during execution, such as attempting to access properties of undefined, will not be captured by static inspection alone. These errors often require dynamic testing to discover.

[4] Is the program inspection technique is worth applicable?

Applicability: Yes, program inspection techniques are valuable. They help identify logical flaws, unused variables, and structural issues early in development, leading to improved code quality. Although they may not catch all runtime errors, they significantly enhance understanding of the code, allowing for better design decisions. Combining inspection techniques with dynamic testing can lead to a more robust and reliable application.