SOFTWARE ENGINEERING
IT314

SMRUTI PARMAR

ID - 202201008

Category A: Data Reference Errors

Uninitialized or unset variables: One of the most frequent programming errors involves referencing a variable that has not been initialized or assigned a value. To ensure correctness, for every reference to a data item (whether it's a variable, array element, or structure field), informally verify that the item is initialized at that point.
Example:

```
while (array.length < 7)
```

1. Error: In this example, `array` is being referenced without prior initialization. The correct variable should be `randNumbers`, which is initialized elsewhere in the code. It is a common mistake to accidentally reference an uninitialized variable.

Array subscript within bounds: Always check that array indices remain within the valid range of the array. Referencing elements outside this range can cause unpredictable behavior.
Example:

```
newArr[i] = arr[len - i];
```

2. Error: Although this operation might appear logically correct for typical inputs, there's no check ensuring that `len - i` remains within the valid bounds of `arr`. It's important to handle such cases carefully by adding bounds checking.

Array subscript must be an integer: When referencing an array, ensure that the index used is always an integer. While some languages (like JavaScript) handle non-integer indices by truncating them to integers, this behavior may vary in other languages.
Example:

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

3. Issue: In this case, `middleIndex` could be a float (especially if the array has an odd length). JavaScript's `splice` method will handle this by converting the float to an integer, but other languages might throw errors or behave unexpectedly.
4. Dangling references or pointers: In some programming languages, a reference or pointer might point to a memory location that is no longer valid. This occurs when a pointer's lifespan exceeds the memory it references (e.g., pointing to a local variable that goes out of scope). It's crucial to ensure that every pointer or reference is pointing to valid memory when used.
   Example:
   In JavaScript, explicit pointers are not used, so issues like dangling references are not

as prevalent. However, arrays or objects should be initialized before use to avoid runtime errors related to unallocated memory.

5. Memory aliasing issues: In some languages (like FORTRAN with `EQUIVALENCE` or COBOL with `REDEFINES`), two different variables can point to the same memory location, potentially causing unintended behavior if their types or attributes differ.
   Example:
   JavaScript does not allow memory aliasing in the same way, so this issue doesn't arise here.

Incorrect variable types or attributes: Ensure that the value of a variable has the expected type and attributes when used.
Example:

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

6. Potential Issue: The `income` value is being converted to a float using `parseFloat()`, but ensure that this conversion always succeeds. If `income` contains invalid data, `parseFloat()` might return `NaN`, leading to incorrect calculations.
7. Addressing problems with memory allocation:
   In certain environments, if memory is allocated in smaller units than the system's addressable memory units, referencing it can lead to errors. This issue occurs mostly with systems that use bit-string manipulations.
   Example:
   JavaScript manages memory automatically, so this problem is avoided.
8. Pointer reference attribute mismatches: In languages like C++, pointers are often used to manipulate data structures. If a pointer that was intended for one structure is mistakenly used for another, it can cause serious errors.
   Example:
   This concern does not apply to JavaScript, as pointers are not explicitly handled in the language.
9. Consistent structure definitions across procedures: When using data structures in multiple functions or subroutines, it is essential to ensure that they are consistently defined in each context.
   Example:
   In functions like `reverseArray()` and `removeMiddleItem()`, array structures are used consistently, so no issues arise in this case.
10. Off-by-one errors when indexing strings or arrays:
    A common source of errors occurs when an index is off by one, either accessing an invalid element or leaving one element unprocessed.
    Example:
    No off-by-one errors are present in the provided code.

11. Object-oriented inheritance requirements:
    In object-oriented languages, classes that implement inheritance must fulfill all inheritance requirements, such as correctly overriding methods.
    Example:
    This code does not involve classes or inheritance, so there are no related issues.

Category B: Data Declaration Errors

Explicit variable declarations:
Variables should be explicitly declared to avoid confusion or runtime errors. A failure to declare variables, particularly in languages with global and local scoping, can lead to unintentional behavior.
Example:`const randNumbers = []; while (array.length < 7)`

1. Error: The variable `array` is referenced without prior declaration, which is a common oversight. In JavaScript, undeclared variables can lead to runtime errors if strict mode is enabled.
2. Implicit vs explicit variable attributes:
   In many languages, variables can receive default attributes if not explicitly defined. These defaults can sometimes be surprising, so it’s important to understand them well.
   Example:
   JavaScript is dynamically typed, meaning variables can hold different types at different points in the program. Be cautious when reusing variables for different types, as this can lead to unexpected behavior.

Proper initialization in declarations:
When variables are initialized in their declaration, ensure the initialization is correct and accounts for special cases, such as arrays or objects.
Example:

```
 let sum = 0;
let max;
let randNumbers = [];
```

3. These variables are all properly initialized at the time of their declaration, so there are no issues.
4. Correct length and data type assignments:
   Although JavaScript is dynamically typed, care should be taken to manage array lengths and data types properly during execution.
5. Consistency between variable initialization and memory type:
   In statically typed languages, this would mean ensuring that a variable’s initialization matches its declared type. In JavaScript, where memory types are abstracted, this issue is less critical but still important.

6. Similar variable names:
   Using similarly named variables (like `VOLT` and `VOLTS`) can be confusing and lead to subtle bugs.
   Example:
   The variables in the code all have distinct names, so there is no risk of confusion from similarly named variables.

By carefully considering these common error categories, programmers can reduce the likelihood of subtle, hard-to-detect bugs.Category C: Computation Errors

1. Inconsistent data types in computations:
   In JavaScript, variables are dynamically typed, meaning their types can change during runtime. While this flexibility can be useful, it can also lead to errors if you're not careful about how variables are used in calculations, especially when switching between numeric and non-numeric data types.
2. Mixed-mode arithmetic:
   A potential pitfall arises when operations involve different data types, like integers and floating-point numbers. While JavaScript automatically handles floating-point arithmetic, other languages might not. For example, in Java, dividing two integers results in integer division, which can lead to precision loss (e.g., 1/2 becomes 0). In JavaScript, the result of 1/2 would be 0.5 due to its default use of floating-point numbers, but precision issues can still occur in mixed-mode calculations, so you need to be mindful of them.
3. Length mismatches in computations:
   JavaScript dynamically adjusts variable sizes, so computations involving variables of different lengths generally won't cause errors. However, in other languages, you must ensure that calculations don't involve variables with mismatched sizes, as this could lead to memory or precision issues.
4. Assigning larger results to smaller variables:
   In languages where variable types and lengths are strictly defined, assigning a value larger than the variable's type can hold might lead to data loss or overflow. Fortunately, JavaScript automatically handles this situation by adjusting variable sizes as needed.
5. Overflow or underflow possibilities:
   Even though JavaScript uses floating-point arithmetic for all numbers, extremely large or small values can result in overflow or underflow errors. For instance, when working with very large sums or products, JavaScript can reach its limit and produce inaccurate results due to precision loss.
6. Division by zero:
   The code does not have any explicit division operations. However, if a division by zero occurs in other contexts, JavaScript would return `Infinity`, rather than crashing, unlike in many other programming languages where it causes a runtime error.
7. Base-2 inaccuracies in floating-point operations:
   JavaScript, like most modern languages, uses binary (base-2) floating-point arithmetic. This can introduce inaccuracies in calculations involving decimals. For instance,

multiplying 0.1 by 10 may not yield an exact 1.0 because some decimal fractions cannot be exactly represented in binary form.

8. Variable ranges in calculations:
   Ensure that variables stay within a valid range during computations. For example, in cases where income calculations involve extracting numerical values from strings, you must ensure that the numbers are within a reasonable range to avoid potential errors in the logic.
9. Operator precedence in complex expressions:
   When expressions involve multiple operators, the order in which they are evaluated can affect the result. JavaScript follows standard precedence rules, so operators like && (logical AND) have higher precedence than `||` (logical OR). It's important to review complex logical expressions carefully to ensure they behave as expected.
10. Integer arithmetic issues:
    JavaScript avoids many issues with integer arithmetic, such as truncating results during division, by defaulting to floating-point operations. However, in languages with stricter integer types, improper integer arithmetic could lead to incorrect results. No such issues exist in the provided code.

Category D: Comparison Errors

1. Comparisons between variables of different data types:
   In JavaScript, strict equality (===) checks both the value and the type of variables, while loose equality (==) only compares values after type coercion. For instance, comparing `age === "30"` (a string) will return `false` if `age` is a number. Developers should be aware of these differences to avoid unexpected outcomes in comparisons.
2. Mixed-mode comparisons:
   When comparing variables of different types or lengths, ensure the conversion rules are understood. In JavaScript, string comparisons and numeric comparisons work correctly if the types match. For example, checking the length of a string (`name.length`) works properly when compared to a number.
3. Misuse of comparison operators:
   It's easy to confuse comparison operators such as <, <=, >, and >=. Make sure you understand the logic behind each comparison. For example, `age <= 30` means "age is at most 30", while `age >= 30` means "age is at least 30". These subtle differences can significantly affect program logic.
4. Boolean expressions:
   Logical expressions can sometimes be miswritten, especially when they involve multiple conditions. The provided expression `if (age > 18 && age < 60)` is correct, as it checks that age falls between 18 and 60. However, always double-check that Boolean expressions represent the desired logic.

5. Mixing Boolean and comparison operators:
   A common error is mixing Boolean and comparison operators incorrectly, such as writing `if (age > 30 || 40)`. This expression should be rewritten as `if (age > 30 || age === 40)` to properly evaluate both conditions.
6. Floating-point comparison issues:
   Due to the binary representation of floating-point numbers, direct comparisons of decimal numbers may lead to unexpected results. This is because certain decimal fractions cannot be exactly represented in binary, leading to small inaccuracies.
7. Operator precedence in Boolean expressions:
   In JavaScript, logical operators such as && (AND) and `||` (OR) follow standard precedence rules. For example, `if (a==2 && b==2 || c==3)` is interpreted as `(a==2 && b==2) || c==3`, which might not be the intended logic. Always use parentheses to clarify the order of operations if needed.
8. Short-circuit evaluation:
   JavaScript uses short-circuit evaluation for logical expressions. In an expression like `if (x !== 0 && (x / y) > z)`, if x is zero, JavaScript will not evaluate the second condition, avoiding a potential division-by-zero error.

Category E: Control-Flow Errors

1. Multiway branches:
   The code does not contain any multiway branches like a computed `GO TO`, so there is no risk of an index exceeding the available branch possibilities.
2. Loop termination:
   Loops in the code, such as `for` loops, will terminate because they are structured with clear conditions. For instance, `for (let i = 0; i < 7; i++)` will run exactly 7 times, incrementing `i` on each iteration until it reaches 7.
3. Program or function termination:
   Functions in the code, such as `fizzBuz` and `findMax`, will terminate as expected since they process finite data sets or have explicit return conditions. There are no infinite loops or scenarios where termination is uncertain.
4. Non-executing loops:
   A loop might never execute if the entry condition is false at the start. For example, `for (i = x; i <= z; i++)` will not run if x is greater than z. Similarly, a `while (NOTFOUND)` loop will not run if `NOTFOUND` is initially `false`.
5. Loop fall-through in search loops:
   In search loops controlled by a Boolean condition, like `while (NOTFOUND)`, it's important to ensure that the loop eventually exits. If `NOTFOUND` remains `true`, the loop might run for unnecessary iterations or indefinitely if not properly controlled.

6. Off-by-one errors:
   JavaScript developers should be cautious of off-by-one errors, especially in zero-based loops. In the code, loops are structured correctly, such as the `for` loop that runs exactly 7 times. There is no off-by-one issue here.
7. Bracket or block mismatches:
   JavaScript uses braces (`{}`) to define code blocks. Mismatched or missing braces can lead to syntax errors, but most modern compilers or interpreters will catch these errors during the initial compilation or execution phase.
8. Non-exhaustive decisions:
   The code considers edge cases in its logic. For example, the `findMax` function ensures it handles valid numerical inputs, while `fizzBuz` covers all relevant conditions for numbers divisible by 3 or 5. No assumptions about missing cases are made, so the logic is comprehensive.

Category F: Interface Errors

1. Parameter count and argument matching:
   Functions are called with the correct number of arguments, ensuring that parameters match with the arguments passed during invocation.
2. Parameter attributes:
   All function parameters have attributes that match the corresponding arguments. For instance, data types such as numbers and strings are handled consistently throughout the code.
3. Units compatibility:
   Although this code does not involve physical units like meters or degrees, it's important to ensure that in cases where units are involved, parameters and arguments use the same units.
4. Number of arguments between modules:
   The code doesn’t interact with external modules, so there is no issue regarding argument mismatches when passing data between modules.
5. Argument attributes between modules:
   No external module calls are present, so this concern does not apply.
6. Units matching in module interactions:
   Again, no external modules are involved, so this concern is not relevant to this code.
7. Built-in function usage:
   Functions like `Math.floor()` and `String.endsWith()` are used correctly with appropriate arguments, ensuring no interface errors with built-in JavaScript functions.
8. Range issues with built-in functions:
   The arguments passed to built-in functions, such as `Math.floor()`, are within their expected ranges, preventing any errors due to out-of-range values.

Category H: Other Checks (Revised)

1. Cross-reference listing for unused variables:
   If the compiler can generate a cross-reference listing, it can help identify variables that are either never used or referenced only once. In this case, the JavaScript code lacks a cross-reference listing feature. However, modern IDEs or static analysis tools can help spot unused variables. Removing or optimizing such variables improves both readability and efficiency.
2. Attribute listing examination:
   In some compiled languages, the compiler generates an attribute listing that shows variable types, scope, and other properties. Reviewing this list helps ensure that no variables have incorrect or unexpected attributes. JavaScript, however, does not enforce strict data types, making such checks irrelevant in this context. Yet, tools like TypeScript can offer static type-checking to catch potential type-related issues.
3. Compiler warnings and informational messages:
   If the program compiles with warnings or informational messages, these should be examined carefully. Warnings often indicate potential logical issues, such as unused variables, deprecated functions, or performance bottlenecks. JavaScript environments such as Node.js or modern browsers will produce warnings or suggestions, which should be reviewed and resolved to prevent future problems.
4. Robustness and input validation:
   Robust programs ensure that input is valid before processing. In this JavaScript code, there is minimal validation of user input. Functions like `addProduct` or `editTask` do not check if the input is valid (e.g., checking for empty strings, invalid numbers, or out-of-range indices). Adding input validation would make the program more resilient and less prone to runtime errors.
5. Missing functions:
   Based on the task requirements, the code appears to cover most necessary functionalities. No critical functions seem to be missing. However, improvements could be made by adding utility functions for input validation or error handling to make the code more maintainable and secure.

Program Inspection Summary

1. Number of errors identified and their nature:
   Several issues were found upon inspection of the code:
   - Logic error in `countWords1`:
     The regular expression used to find words does not accurately match whole words. It may count partial words (like counting "love" when "lovely" appears), leading to incorrect results.
   - Logic error in `countWords2`:
     The use of `includes()` checks for substrings rather than exact word matches. This means that searching for "cat" could also match words like "catch" or "cater", which inflates the word count.

    - Comment mistake in `agesGreaterEighteen`:
      The comment describing the expected return value of this function mentions a trailing comma in the result, which does not align with the actual output, potentially confusing future developers.
    - Uninitialized `middleIndex` in `removeMiddleItem`:
      When called with an empty array, the variable `middleIndex` is left undefined. This could result in errors when the array's `splice()` method is used, as it expects a valid index.
    - Variable reassignment issue in `removeAll`:
      The `todoList` is declared as a constant, and reassigning it will cause a runtime error. Instead, the array should be emptied using `todoList.length = 0` to avoid this issue.
    - Lack of input validation:
      Functions like `addProduct` and `editTask` do not check whether the input provided is valid. For example, if an invalid index is passed, the function could behave unpredictably.
    - Inefficient logic in `toggleAll`:
      The logic for toggling all tasks could be simplified. The current structure involves nested conditions, which not only make the code harder to read but also introduce the possibility of logical errors in the toggling process.
2. Most effective category of inspection:
   The most effective type of inspection for this code falls under Category B: Logic Errors. Many of the issues identified, such as the incorrect word counting in `countWords1` and `countWords2`, and the unhandled edge cases in `removeMiddleItem`, are logic errors. Identifying and correcting these mistakes is crucial for ensuring that the program functions as intended.
   Another important category is Category E: Interface Errors. This category addresses problems where functions are not given the correct number or type of arguments. Ensuring correct data handling between functions prevents many issues from arising at runtime.
3. Errors not identifiable through inspection alone:
   While many errors can be identified through code inspection, there are several types that are harder to detect without running the program:
    - Subtle logical errors:
      Errors related to output accuracy, such as the `countWords1` function counting "love" in "lovely", may not be apparent through inspection alone. These errors only become clear during testing or runtime.
    - Runtime errors:
      Mistakes like accessing an out-of-range index or passing an undefined variable typically do not manifest until the program is executed. This is especially true in dynamic languages like JavaScript.

    - Performance issues:
      Inefficient algorithms or operations (e.g., traversing arrays multiple times unnecessarily) may not be obvious in an inspection. Performance profiling or stress testing would be required to identify such bottlenecks.
4. Applicability of program inspection techniques:
   Yes, program inspection techniques are worth applying, as they offer several benefits:
    - Early detection of errors:
      Code inspections help identify syntactical and logical issues before running the program, saving time in later testing phases.
    - Improved code quality:
      By reviewing the code systematically, inspections can reveal flaws in logic, structure, or readability that might be missed otherwise.
    - Collaborative understanding:
      Inspections foster better team collaboration by allowing multiple developers to review and understand the code together, improving both the quality and maintainability of the program.
5. Limitations:
    - While inspections are effective for identifying many issues, they might miss runtime errors, performance bottlenecks, or subtle logical bugs. These problems are better addressed through dynamic testing (e.g., unit testing, integration testing, or performance profiling).
    - Inspections can be time-consuming, and without proper guidance or experience, some errors might still slip through. For a comprehensive quality assurance process, code inspections should be supplemented with automated testing.

In summary, applying program inspection techniques to this JavaScript code reveals several potential errors, including logic and input handling issues. However, inspection alone is not sufficient to catch all types of bugs, particularly those related to runtime behavior or performance. A combination of inspection, dynamic testing, and validation techniques will yield the most robust results.

```
 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 (Reworded)

1. Unset or Uninitialized Variable:
   Issue: A variable is used without being initialized or assigned a value.
   Example: `while (array.length < 7)`
   Correction: The variable `array` is being referenced before it's initialized. It should be `randNumbers`, which is properly initialized in the code. Always ensure that variables are initialized before use to avoid runtime errors.
2. Array Subscript Out of Bounds:
   Issue: Array indexes or subscripts used in the code exceed the valid range of the array, which can lead to unexpected behavior.
   Example: `newArr[i] = arr[len - i];`
   Correction: To avoid accessing invalid indices, check that `len - i` remains within the valid bounds of the array. Implement bounds checking to ensure array indices are properly handled, especially in loops or recursion.
3. Non-integer Array Subscript:
   Issue: Some languages require that array indices be integers. Using non-integer subscripts can cause issues in languages that are strict with types.
   Example: `arr.splice(middleIndex, 2, itemOne, itemTwo, itemThree);`
   Explanation: The variable `middleIndex` could become a floating point number when dividing an odd-length array. JavaScript handles this, but other languages might not, so converting to an integer using `Math.floor()` would be safer.
4. Dangling Reference:
   Issue: A pointer or reference continues to be used after the memory it points to has been deallocated.
   Example: JavaScript doesn't support pointers directly, but the issue can occur when references to objects or arrays are accessed after they are redefined or deleted. Ensure that any array or object is still valid before using its reference.
5. **Memory Alias with Incorrect Attributes**:
   Issue: In languages that allow aliasing (like FORTRAN or COBOL), different names for the same memory area may have conflicting attributes, leading to incorrect data interpretation.
   Example: This type of error is not a concern in JavaScript, where aliasing issues related to memory attributes do not apply. Variables and memory management are abstracted, preventing this issue.
6. **Variable Value with Incorrect Type or Attribute**:
   Issue: The value of a variable may not match the expected type or attribute, leading to unexpected behavior.
   Example: `incomes.forEach((income, i) => { sum += parseFloat(income) * 12; });`
   Correction: Ensure that `parseFloat()` correctly converts the value into a floating point

number. If it fails (e.g., due to non-numeric input), it returns `NaN`, which should be handled to prevent incorrect calculations.

7. **Addressing Problems**:
   Explanation: JavaScript abstracts direct memory access and management, so problems like misaligned memory addresses or pointer arithmetic issues are not relevant in this language.
8. **Referenced Memory with Unexpected Attributes**:
   Explanation: Since JavaScript does not support low-level memory management, issues with memory references having unexpected attributes (e.g., incorrect read/write permissions) are not applicable. JavaScript's memory model ensures safe memory access.

Category B: Code Errors (Rewritten)

1. **Incorrect Loop Conditions**:
   Issue: Loops may not terminate correctly or may iterate an unintended number of times due to improper conditions.
   Example: `for (let i = 0; i < n; i++) { console.log(${i} * ${i} = ${i * i}); }`
   Correction: Make sure that the loop variable `n` is initialized and has the correct value before entering the loop. Loops should be carefully constructed to prevent infinite or premature terminations.
2. **Incorrect String Interpolation**:
   Issue: Improper usage of template literals or string interpolation may lead to incorrect output.
   Example: `console.log(${i} * ${i} = ${i * i});`
   Correction: Verify that all variables, like `i`, are correctly defined and accessible within the scope. Ensure they have valid values when used within template literals.
3. **Undefined Variables**:
   Issue: Variables that have not been declared or initialized are used, causing runtime errors.
   Example: `const total = products.map(prod => prod.price).filter(price => typeof price == 'number').reduce((curr, acc) => curr + acc);`
   Correction: Ensure that all variables (like `products`) are properly declared and initialized before they are referenced in any operation.
4. **Potential Infinite Loop in Input Function**:
   Issue: Lack of proper input validation can cause loops to run indefinitely.
   Example: `const userInput = prompt('Enter a month: ').trim().toLowerCase();`
   Correction: Implement robust validation for user input. For example, verify that the input

is a valid month before proceeding to prevent the possibility of an infinite loop due to invalid input.

5. **Non-returning Function**:
   Issue: A function may fail to return a value as expected, leading to unexpected behavior when the return value is used.
   Example: `const howManyDaysInMonth = () => { ... }`
   Correction: Ensure that the function always returns a value in all code paths. For example, use explicit return statements, especially in functions that are expected to return data.
6. **Incorrect Data Structure Access**:
   Issue: Accessing elements of a data structure incorrectly, such as using the wrong key or index, leads to errors.
   Example: `console.log(createArrayOfArrays(countries));`
   Correction: Verify that the data structure (`countries` in this case) is correctly formatted and initialized before accessing or manipulating its elements.
7. **Invalid Type Comparison**:
   Issue: Comparing values of different data types can result in unexpected results due to type coercion.
   Example: `let formattedPrice = typeof price == 'number' ? price : 'unknown';`
   Correction: Always check for the correct data type before comparison. For strict comparison, use === to avoid issues with JavaScript's type coercion.
8. **Potentially Undefined Return Values**:
   Issue: Functions that might not return a value under certain conditions can cause problems when the return value is expected.
   Example: `return 'Not a valid format';`
   Correction: Ensure all paths within a function, including conditional branches, provide valid return values to avoid runtime errors.

Category C: Resource Management Errors (Expanded)

1. **Memory Leak**:
   Issue: Dynamically allocated memory is not released, resulting in increased memory usage over time.
   Example: `let obj = {}; obj = null;`
   Correction: JavaScript handles memory deallocation through its garbage collector, but still, it's important to dereference objects correctly (e.g., setting `obj = null`) to allow for proper garbage collection.
2. **Resource Exhaustion**:
   Issue: Resources such as file handles or network connections are opened but not closed, leading to resource exhaustion.

Example: `const fs = require('fs'); let file = fs.createReadStream('file.txt');`
Correction: Always ensure that file streams, database connections, or similar resources are properly closed using appropriate methods such as `fs.close()` or closing database connections to free resources.

3. **Double Free Error**:
   Issue: Freeing or deleting the same memory resource twice can cause undefined behavior.
   Example: `let arr = new Array(5); delete arr; delete arr;`
   Correction: Ensure memory is deallocated or released only once. In JavaScript, memory management is automatic, but in languages where manual memory handling is needed, double free errors can be prevented by proper bookkeeping.
4. **Invalid Memory Access**:
   Issue: Accessing memory that has already been deallocated can lead to crashes or unpredictable behavior.
   Example: `let arr = new Array(5); arr = null; console.log(arr[0]);`
   Correction: Avoid accessing memory or variables after they have been released or set to `null`. In JavaScript, setting a variable to `null` should prevent accidental use.
5. **File Handle Leak**:
   Issue: If file handles are not closed properly, the application can run out of available file handles.
   Example: `fs.open('file.txt', 'r', (err, fd) => { /* no fs.close() */ });`
   Correction: Ensure that file handles are always closed after operations using methods such as `fs.close()`. File leaks can cause performance degradation.
6. **Network Resource Leak**:
   Issue: Leaving network connections open without properly closing them can lead to network resource exhaustion.
   Example: `const socket = new WebSocket('ws://example.com'); // No socket.close()`
   Correction: Always close network connections when they are no longer needed by using `socket.close()` or similar methods to prevent resource leaks.
7. **Improper Resource Handling in Loops**:
   Issue: Resources are allocated within loops without proper cleanup, leading to resource exhaustion or leaks.
   Example: `for (let i = 0; i < 1000; i++) { let resource = allocateResource(); }`
   Correction: Ensure that resources allocated in loops are released or reused properly after each iteration. This helps avoid excessive memory usage.
8. **Concurrent Resource Access Issues**:
   Issue: In multithreaded environments, unsynchronized access to shared resources can lead to race conditions.

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

Category D: Control Flow Errors

1. **Infinite Loop**:
   An error where a loop runs endlessly without stopping, causing the program to execute indefinitely.
   Example:

javascript
Copy code
```
while (true) {
  console.log('Hello');
}
```

Solution: Ensure that the loop includes a proper condition for termination.

2. **Faulty Loop Condition**:
   Occurs when the loop's condition does not behave as intended, either terminating too early or running endlessly.
   Example:

javascript
Copy code
```
for (let i = 0; i < 10; i--) {
  console.log(i);
}
```

Solution: Check the condition to confirm it allows the loop to function properly.

3. **Incorrect Break or Continue Placement**:
   Misplacing a `break` or `continue` statement may cause unexpected behavior in control flow.
   Example:

javascript
Copy code
```
for (let i = 0; i < 10; i++) {
  if (i === 5) break;
  console.log(i);
}
```

Solution: Ensure `break` or `continue` statements are positioned appropriately within loops.

4. **Unreachable Code**:
   Code that can never be executed because of the preceding control flow.
   Example:

javascript
Copy code
```
if (false) {
  console.log('This will never run.');
}
```

Solution: Eliminate or adjust unreachable code blocks.

5. **Improper Exception Handling**:
   Failure to manage exceptions correctly can crash the program or disrupt the control flow.
   Example:

javascript
Copy code
```
try {
  throw new Error('Something went wrong');
} catch (e) {
  console.log('Error handled');
}
```

Solution: Ensure that exceptions are properly caught and handled to prevent unexpected behavior.

6. **Incorrectly Nested Control Structures**:
   Errors occur when control structures like `if` statements are nested improperly, which can cause logical issues.
   Example:

javascript
Copy code
```
if (condition1) {
  if (condition2) {
    console.log('Nested!');
  }
}
```

Solution: Maintain clear and logical nesting for control structures.

7. **Missing Return Statement**:
   A function designed to return a value but lacks a return statement.
   Example:

javascript
Copy code
```
function add(a, b) {
  a + b;
}
```

Solution: Ensure all functions return a value if expected.

8. **Mishandling Asynchronous Code**:
   Improperly handling asynchronous code can lead to unexpected outcomes.
   Example:

javascript
Copy code
```
async function fetchData() {
  const data = await getData();
  console.log(data);
}
```

Solution: Use `async/await` or callbacks correctly to manage asynchronous operations.

Category E: Control Flow Errors in Depth

1. **Out-of-Bounds Multiway Branch Index**:
   When a program's branch selection variable exceeds the allowed range of options.
   Example: In `GO TO (200, 300, 400), i`, make sure `i` is always 1, 2, or 3.
   Solution: Verify that the branch index variable remains within valid limits.
2. **Loop Termination Failure**:
   A loop that doesn't stop due to improper termination conditions.
   Example:

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

Solution: Ensure that the loop condition and counter increment will eventually terminate the loop.

3. **Non-Terminating Functions**:
   A function that runs indefinitely due to the lack of an exit condition.
   Example:
   Functions like `fizzBuzz` should have clear termination points.
   Solution: Implement exit conditions in every function to ensure proper completion.
4. **Loops That Never Execute**:
   A loop might never run if its initial condition prevents entry.
   Example:

javascript
Copy code
```
for (i == x; i <= z; i++)
```

Solution: Ensure that the initial condition allows the loop to run.

5. **Loop Fall-Through**:
   When both an iteration counter and a Boolean condition control the loop, the Boolean condition may never be resolved.
   Example:

pseudo
Copy code
```
DO I = 1 TO TABLESIZE WHILE (NOTFOUND)
```

Solution: Ensure that the Boolean condition eventually resolves to false to terminate the loop.

6. **Off-by-One Errors**:
   Loops that iterate one time too many or too few due to improper boundaries.
   Example:

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

Solution: Adjust loop boundaries to correctly account for indexing.

7. **Mismatched Code Blocks**:
   A failure to properly match control statements with their respective code blocks.
   Example:
   In JavaScript, ensure all `if` and `else` statements have the correct `{}` brackets.
   Solution: Properly pair and nest code blocks to avoid logic errors.
8. **Non-Exhaustive Conditional Statements**:
   A decision structure that doesn't handle all possible cases.
   Example:
   If the expected inputs are 1, 2, or 3, ensure the program handles cases where the input is neither 1 nor 2.
   Solution: Make sure all possible cases are addressed in conditionals to avoid logical gaps.

Category F: Interface Errors

1. **Parameter Count Mismatch**:
   The number of arguments passed to a function does not match the number of parameters expected.
   Example: If a function is defined as `example(a, b)` but is called as `example(1)`.
   Solution: Ensure the function is called with the correct number of arguments.
2. **Attribute Mismatch**:
   The types or sizes of parameters do not match what the function expects.
   Example: Passing a string when the function expects an integer.
   Solution: Validate the types and sizes of parameters before passing them.
3. **Inconsistent Units**:
   Using mismatched units, such as passing radians when degrees are expected.
   Example: A function expecting degrees receives radians.
   Solution: Ensure that the units of measurement are consistent or properly converted.
4. **Incorrect Argument Transmission**:
   The number of arguments passed to a function doesn't match the parameters it expects.
   Example: Calling `foo(x, y)` with `foo(1)` causes an error.
   Solution: Confirm that the number of arguments matches the function's signature.
5. **Mismatched Argument Attributes**:
   The characteristics of arguments do not align with the expected parameters, such as an integer passed instead of a float.

Example: A function requires a float but receives an integer.
Solution: Ensure the arguments passed have the correct attributes.

6. **Units Mismatch in Arguments**:
A mismatch between expected units and the actual arguments provided to a function.
Example: Passing kilometers to a function expecting meters.
Solution: Convert units appropriately before calling the function.
7. **Improper Use of Built-in Functions**:
Incorrect argument types or counts used in built-in functions.
Example: Using `Math.max()` with non-numeric values.
Solution: Check the arguments for built-in functions to ensure they meet the expected criteria.
8. **Parameter Modification Error**:
A function inadvertently alters a parameter that should be read-only.
Example: Modifying an input parameter that should remain unchanged.
Solution: Use `const` or ensure the parameter is not modified.
9. **Inconsistent Global Variable Definitions**:
Global variables are defined or used inconsistently across different parts of the program.
Example: A global variable is defined in one module and used differently in another.
Solution: Maintain consistent global variable definitions across modules.

Category G: Input / Output Errors

1. **Incorrect File Attributes**:
When file attributes do not match the intended operations during file declarations.
Solution: Check file attributes to ensure they align with their intended use.
2. **OPEN Statement Attribute Errors**:
Incorrect attributes in a file's OPEN statement.
Solution: Verify the correctness of attributes in the file's OPEN statement.
3. **Insufficient Memory for File Read**:
Not enough memory to store the file being read.
Solution: Ensure sufficient memory is available to read the file.
4. **Opening Files Before Use**:
A file is not opened before attempting to use it.
Solution: Always open files before using them in any operation.
5. **Closing Files After Use**:
Failing to close a file after operations can lead to resource leaks.
Solution: Make sure to close files after operations are complete.
6. **Handling End-of-File Conditions**:
Failure to detect or manage end-of-file situations correctly.
Solution: Implement EOF detection when reading from files.

7. **I/O Error Handling**:
   Failing to manage input/output errors can cause issues during file operations.
   Solution: Include robust error handling for all I/O operations.
8. **Spelling and Grammar in Output**:
   Spelling or grammatical errors in printed or displayed text.
   Solution: Review and correct all output text for errors.

Category H: Additional Checks

1. **Cross-Reference Listing**:
   Check cross-references for unused or infrequently used variables.
   Solution: This code does not generate a cross-reference listing, but it's useful to examine.
2. **Attribute Listing**:
   Check variable attributes to ensure no unexpected defaults.
   Solution: JavaScript does not enforce strict data types, so this step may not apply.
3. **Compiler Warnings**:
   Pay close attention to any compiler warnings or informational messages.
   Solution: Run the code in an appropriate environment and review any warnings.
4. **Input Validation**:
   Ensure that the program performs input validation to prevent errors.
   Solution: Add checks to ensure the validity of input.
5. **Missing Functions**:
   Make sure no critical functions are missing from the program.
   Solution: Review the code for completeness.

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. **Uninitialized or Unset Variables**
   - **Error**: `while (array.length < 7) {`
   - **Explanation**: The variable `array` is being used without initialization. It should reference `randNumbers`, which is properly initialized earlier in the code.
2. **Array Subscript Boundaries**
   - **Error**: `newArr[i] = arr[len - i];`
   - **Explanation**: There's no explicit check to confirm whether `len - i` is within the bounds of `arr`. Without proper bounds checking, the program may reference invalid array indices, especially with unexpected input sizes. Add validation for array bounds.
3. **Integer Subscript Values**
   - **Issue**: `arr.splice(middleIndex, 2, itemOne, itemTwo, itemThree);`
   - **Explanation**: When the array length is odd, `middleIndex` may be calculated as a floating-point number. JavaScript automatically truncates it, but other languages might not. Ensure that subscripts are integers.
4. **Allocated Memory References**
   - **Explanation**: JavaScript abstracts memory management, so direct memory allocation isn't a concern like it is in languages with explicit pointers. In this context, arrays and objects like `newArr` are initialized before use, preventing issues.
5. **Aliased Memory Issues**
   - **Explanation**: Aliasing issues, common in languages like FORTRAN or COBOL, don't apply to JavaScript. Thus, no concerns with differing attributes through alias names are relevant here.
6. **Variable Type Mismatch**
   - **Issue**: `incomes.forEach((income, i) => { sum += parseFloat(income) * 12; });`
   - **Explanation**: The `income` variable is converted to a floating-point number, but there's no validation to ensure it's a valid numeric string. If `parseFloat()` returns NaN, it could lead to incorrect results. Add validation for `income`.
7. **Memory Addressing Issues**
   - **Explanation**: Since JavaScript abstracts memory allocation and doesn't allow direct memory manipulation, addressing issues related to memory unit size don't apply here.
8. **Expected Memory Attributes**
   - **Explanation**: JavaScript doesn't use pointers, making this issue irrelevant for this code.
9. **Consistent Structure Across Functions**

   - **Explanation**: Arrays such as `newArr` and `arr` are consistently defined across functions. However, ensure proper length checks when arrays are passed between functions to avoid errors.
10. **String Indexing Errors**
    - **Explanation**: There are no off-by-one errors when indexing strings or arrays in the provided code. Edge cases, particularly involving loop conditions, are handled correctly.
11. **Inheritance Requirements**
    - **Explanation**: As the code doesn't use object-oriented programming or classes, inheritance issues aren't applicable here.

Category B: Data-Declaration Errors

1. **Explicit Variable Declaration**
   - **Explanation**: Variables like `newArr`, `middleIndex`, and `randNumbers` are all explicitly declared using `let` or `const`. There are no undeclared variable issues.
2. **Default Variable Attributes**
   - **Explanation**: Uninitialized variables in JavaScript are assigned `undefined`. The code does not rely on unexpected default behaviors, so no concerns were found here.
3. **Proper Initialization**
   - **Explanation**: Variables are correctly initialized. For example, `randNumbers` is properly initialized as an empty array before being populated.
4. **Correct Variable Type and Length**
   - **Explanation**: All variables are appropriately typed. Arrays such as `newArr` and `randNumbers` are handled as arrays, with no issues related to length or type.
5. **Consistency with Memory Type**
   - **Explanation**: JavaScript handles memory dynamically, and no inconsistencies with memory types were identified during initialization.
6. **Similar Variable Names**
   - **Explanation**: No confusingly similar variable names are present, avoiding potential naming conflicts.

Category C: Computation Errors

1. **Data Type Consistency in Computations**
   - **Explanation**: No data type inconsistencies were found in the code. The variables used in arithmetic operations are appropriately handled.
2. **Mixed-Mode Computations**

    - **Explanation**: JavaScript manages type coercion internally. For example, `parseFloat(income)` ensures that `income` is converted to a number, preventing mixed-mode computation issues.
3. **Computations with Variables of Different Lengths**
    - **Explanation**: The code does not involve variables with differing lengths in calculations, so no such issues were found.
4. **Target Variable Type in Assignments**
    - **Explanation**: No issues with target variable types were detected. JavaScript's dynamic typing manages these situations effectively.
5. **Overflow or Underflow in Expressions**
    - **Explanation**: No risk of overflow or underflow was detected, though large calculations should be monitored for potential issues.
6. **Division by Zero**
    - **Explanation**: No division operations exist in the provided code, so division by zero isn't a concern.
7. **Base-2 Representation Issues**
    - **Explanation**: While JavaScript can introduce minor floating-point precision errors, the code doesn't involve complex floating-point arithmetic, so no inaccuracy risks were found.
8. **Out-of-Range Values**
    - **Explanation**: The logic appears sound, and no checks for out-of-range values are necessary for the existing code.
9. **Operator Precedence**
    - **Explanation**: The operator precedence in expressions, such as in the `forEach` loop, follows JavaScript rules and is correct.
10. **Invalid Integer Arithmetic**

- **Explanation**: No integer arithmetic errors were identified, and there are no integer division operations in the code.

Category D: Comparison Errors

1. **Comparisons with Different Data Types**
    - **Explanation**: No mismatched data type comparisons were found.
2. **Mixed-Mode Comparisons**
    - **Explanation**: All variables in comparisons have compatible types, so no mixed-mode comparison issues were identified.
3. **Correct Comparison Operators**
    - **Explanation**: All comparison operators are used appropriately.
4. **Boolean Expression Accuracy**
    - **Explanation**: The logic in Boolean expressions is sound. For example, the check `if (len === 0)` is correctly used to test for an empty array.
5. **Boolean Operand Validity**

    - **Explanation**: Boolean expressions use valid comparisons, with no mismatches.
6. **Comparisons with Floating-Point Numbers**
    - **Explanation**: No floating-point number comparisons are used in the code.
7. **Boolean Operator Precedence**
    - **Explanation**: There are no complex Boolean expressions combining && and `||`, so evaluation precedence issues are not applicable.
8. **Boolean Expression Evaluation**
    - **Explanation**: No issues with short-circuit evaluation were found in the code.

Category E: Control-Flow Errors

1. **Index Variable in Multiway Branches**
    - **Explanation**: JavaScript doesn't use multiway branches like `GO TO`. The control flow in the code, which uses `if-else` and loops, is well-structured and valid.
2. **Loop Termination**
    - **Explanation**: All loops have clear termination conditions, such as `for (let i = 0; i < len; i++)`, ensuring proper termination.
3. **Program or Subroutine Termination**
    - **Explanation**: The program and all loops will terminate appropriately based on the conditions provided.
4. **Zero-Iteration Loops**
    - **Explanation**: No loops were identified that could potentially never execute due to initial conditions.
5. **Fall-Through in Loops**
    - **Explanation**: No loops combine iteration and Boolean conditions, avoiding any potential fall-through issues.
6. **Off-by-One Errors**
    - **Explanation**: No off-by-one errors were detected. The array loops correctly handle zero-based indexing.
7. **Block Closure**
    - **Explanation**: All code blocks have matching opening and closing brackets.
8. **Exhaustive Decisions**
    - **Explanation**: All decision structures cover the necessary conditions.

Category F: Interface Errors

1. **Parameter Count Matching**
    - **Explanation**: All functions receive the correct number of parameters.
2. **Parameter Attribute Matching**

   - **Explanation**: The attributes of function parameters match those of the arguments passed.
3. **Unit Systems**
   - **Explanation**: No unit system mismatches are present, as the code deals only with arrays and numbers.
4. **Argument Transmission**
   - **Explanation**: Functions receive the expected number of arguments, and all match the defined parameters.
5. **Argument Attribute Matching**
   - **Explanation**: Arguments passed to functions have the correct attributes.
6. **Built-in Function Argument Validity**
   - **Explanation**: Functions such as `splice()` and `forEach()` are used correctly with valid arguments.
7. **Parameter Mutation**
   - **Explanation**: No parameters that are meant to be inputs are altered unexpectedly.
8. **Global Variable Scope**
   - **Explanation**: No global variables are used, avoiding any scope or consistency issues.

Category G: Input/Output Errors

1. **File Declarations**
   - **Explanation**: The code doesn't involve file handling.
2. **Input Misalignment**
   - **Explanation**: No input-output misalignment errors were found.

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. **Unset or Uninitialized Variables**
   - **Example**: In the `signIn` function, `let found = false;` is used to indicate whether a user has been located. It's essential to ensure that `found` is correctly set to `true` when a user is identified.
   - **Another Example**: In the `averageRating` function, `let len;` is declared but left uninitialized. Before performing any operations that rely on `len`, it should be assigned a valid value to prevent undefined behavior in calculations.
2. **Array Index Boundaries**
   - **Example**: In the `countPalindromeWords` function, `for (const word of arr) {...}` iterates over the array `arr`. Although this loop will work fine even if `arr` is empty, it's advisable to ensure valid, non-empty inputs to avoid unnecessary iterations or errors in logic.
3. **Integer Subscripts in Arrays**
   - **Explanation**: The code avoids any non-integer array indices, which could lead to unexpected behavior in some languages. All array subscripts appear to be integers, so there are no concerns here.
4. **Dangling References or Unallocated Memory**
   - **Explanation**: Unlike languages like C or C++, JavaScript doesn't use explicit pointers. Memory management is handled automatically, but global objects or arrays like `users` or `products` could still be modified from elsewhere, which might introduce logical issues if not managed properly.
5. **Memory Aliasing with Different Attributes**
   - **Explanation**: This issue, common in languages like FORTRAN or COBOL, does not apply in JavaScript. JavaScript does not support constructs like `EQUIVALENCE` that allow multiple variables to reference the same memory location with different attributes.
6. **Type Mismatches**
   - **Example**: In the `isPrime` function, it's essential to confirm that the n variable is always a number. If it's passed as a string or other non-numeric type, the function could behave unpredictably. Always validate input types to prevent such issues.
7. **Memory Addressing Conflicts**
   - **Explanation**: JavaScript handles memory allocation and addressing automatically, so concerns about mismatched memory units or bit-string boundaries are irrelevant here.
8. **Pointer Issues**
   - **Explanation**: Since JavaScript doesn't use pointers in the way languages like C++ do, concerns about pointer references and expected memory locations are not applicable. Still, be sure that object structures like `users` and `products` are always in the expected format.
9. **Data Structure Consistency Across Functions**

    - **Example**: When data structures like `users` and `products` are passed between functions, ensure they are consistently defined. If different functions expect different structures, it could lead to errors when accessing properties or methods.
10. **String and Array Indexing**
- **Example**: In the `isPrime` function, the loop `for (let i = 2; i < n; i++)` runs from 2 to `n-1`. For efficiency, it's better to loop up to `Math.sqrt(n)` since any factors of n beyond this point are redundant.
11. **Inheritance in Object-Oriented Code**
- **Explanation**: The current code doesn't employ object-oriented features like class inheritance. Therefore, there are no inheritance-related issues to consider.

Category B: Data-Declaration Errors

1. **Explicit Declaration of Variables**
    - **Example**: In the `solveQuadratic` function, parameters a, b, and c are declared with default values (e.g., `a = 1, b = 0, c = 0`). This is good practice, but you should ensure that these defaults are appropriate and that undefined values are handled correctly if the function is called without arguments.
2. **Default Variable Attributes**
    - **Example**: JavaScript's default handling of undeclared variables can lead to confusion. For instance, the default value of `undefined` might be interpreted as a valid input in some cases. It's important to explicitly handle cases where variables might be `undefined`, especially when they are passed into functions like `isPrime`.
3. **Initialization at Declaration**
    - **Example**: The variable `let found = false;` is correctly initialized in the `signIn` function. However, `let len;` is declared without an initial value in the `averageRating` function. Always ensure that variables are initialized properly to prevent unpredictable behavior later in the code.
4. **Correct Data Type and Length**
    - **Example**: In the `averageRating` function, the `productId` parameter should always be validated as a string. This ensures consistency and prevents errors when the function looks up products by their ID.
5. **Memory Type and Initialization**
    - **Example**: Variables like `let count = 0;` are properly initialized in the code. JavaScript's automatic memory management ensures that type mismatches are rare, but you should remain mindful of initialization and memory type, especially when handling more complex objects.
6. **Similar Variable Names**

   - **Example**: Be cautious with similarly named variables. For instance, `root1` and `root2` in `solveQuadratic` could be confused with similarly named variables in other parts of the program. Additionally, the variable `found` in the `signIn` function might cause confusion if a similarly named variable exists in a different scope. It's a good idea to avoid reusing the same or similar names in different contexts to reduce ambiguity.

Category C: Computation Errors

1. **Type Consistency in Computations**
   - **Example**: In the `solveQuadratic` function, the expression `const determinant = b ** 2 - 4 * a * c;` should only be evaluated with numeric values. If a, b, or c are not numbers, the computation could yield incorrect results. Always validate the types of these variables before performing arithmetic operations.
2. **Mixed-Mode Computations**
   - **Explanation**: Mixed-mode computations, where variables of different types (e.g., strings and numbers) are combined, can lead to unexpected results. For example, adding a string to a number in JavaScript could result in string concatenation rather than arithmetic addition. Ensure that all variables in calculations are of the appropriate type.
3. **Different Length Variables in Computations**
   - **Explanation**: In JavaScript, variables can dynamically switch between types and sizes, reducing the risk of errors from differing lengths. Nonetheless, ensure that consistent numeric types (such as integers versus floating-point numbers) are used in computations to avoid precision issues.
4. **Data Type Mismatch in Assignments**
   - **Example**: When assigning the length of an array to a variable, like `let count = primes.length;`, ensure that `count` is treated consistently as a number, particularly if other operations expect it to be of a specific type.
5. **Overflow and Underflow Risks**
   - **Example**: In expressions like b `** 2 - 4 * a * c`, be cautious of overflow when working with large values for b. Although JavaScript can handle relatively large numbers, there's always the risk of exceeding safe integer limits, especially in complex calculations. Keep an eye on the size of intermediate results.
6. **Division by Zero**
   - **Example**: In the `solveQuadratic` function, the expression `root1 = -b / (2 * a);` could lead to a division by zero if a is zero. It's essential to add a condition to check if a is zero before performing this calculation, to avoid runtime errors.

7. **Base-2 Representation and Floating-Point Accuracy**
   - **Explanation**: JavaScript handles floating-point arithmetic reasonably well, but be mindful of potential rounding errors, especially with fractional numbers like `0.1`. Precision can be lost in repeated operations or when dealing with very small or very large numbers.
8. **Out-of-Range Values**
   - **Example**: In the `solveQuadratic` function, ensure that a, b, and c are within valid ranges for the quadratic equation to be meaningful. For instance, if a is zero, the equation ceases to be quadratic and might lead to incorrect assumptions in the logic.
9. **Operator Precedence in Complex Expressions**
   - **Explanation**: When dealing with multiple operators in a single expression, be aware of JavaScript's operator precedence. For instance, in `b ** 2 - 4 * a * c`, the exponentiation and multiplication are evaluated before the subtraction, which is correct, but it's important to ensure clarity for anyone reading the code.
10. **Integer Arithmetic and Division**
    - **Explanation**: While JavaScript handles floating-point arithmetic automatically, ensure that integer arithmetic (like divisions) is handled appropriately. For example, dividing two integers might yield unexpected results if you later use the result in floating-point operations.

CATEGORY D: Comparison Errors

1. **Data Type Comparisons:**
   Are variables of different data types being compared, like comparing strings with numbers or dates? These mismatches can lead to unexpected outcomes. Ensure that comparisons are made between variables of the same type.
2. **Mixed-Mode Comparisons:**
   When comparing variables of different types or lengths, understand how the conversion rules work. For example, comparing a numeric string to a number can give odd results if you're not familiar with how types are converted.
3. **Correctness of Comparison Operators:**
   Misuse of comparison operators like `<`, `<=`, `>=`, etc., is common. Check whether you are using the right operator for the intended logic, especially in complex conditions.
4. **Boolean Expression Accuracy:**
   Ensure your Boolean expressions do what they're supposed to. Mistakes often happen with logical operators (`and`, `or`, `not`). Verify that your logic matches your intention.
5. **Boolean Operand Correctness:**
   Are the operands in your Boolean operations actually Boolean values? Mixing comparison and Boolean operators incorrectly (e.g., `2 < i < 10` instead of `(2 < i) && (i < 10)`) can lead to logic errors.

6. **Floating-Point Comparisons:**
   Be cautious when comparing floating-point numbers due to their base-2 representation on machines. Direct comparisons may not work as expected due to rounding errors.
7. **Order of Boolean Operations:**
   Understand the order of evaluation in expressions with multiple Boolean operators. For example, does the && in a `== 2 && (b == 2 || c == 3)` execute before the `||`? Ensure it works as intended.
8. **Compiler-Specific Boolean Evaluation:**
   Be aware that different compilers may handle Boolean expression evaluation differently, especially with short-circuiting (e.g., division by zero errors in expressions like `x == 0 && (x / y) > z`).

CATEGORY E: Control-Flow Errors

1. **Multiway Branches:**
   If the code includes a multiway branch, ensure that the index variable does not exceed the number of cases. No such constructs are found in this code, but attention to indexing would be crucial.
2. **Loop Termination:**
   All loops should be designed to eventually terminate. The logic in your loops should provide a clear exit condition to avoid infinite loops.
3. **Subroutine Termination:**
   Every function or module should terminate properly. Each function in this code concludes correctly, reaching a return statement as expected.
4. **Loop Non-Execution Due to Initial Conditions:**
   Avoid writing loops that might never execute due to starting conditions. For example, a `for` loop that starts with `i = x` where `x > z` will not run. Ensure that your loop's initial conditions allow execution when necessary.
5. **Loop Fall-Through:**
   In loops with combined Boolean and iteration controls (like search loops), ensure that the condition will eventually be met to break the loop, preventing indefinite looping.
6. **Off-by-One Errors:**
   Off-by-one errors are common, especially with zero-based loops. Verify that your loop ranges are correct and you are not running one iteration too many or too few.
7. **Matching Code Blocks:**
   Ensure that every code block opened (like in `do-while` loops or `if` statements) is properly closed with corresponding brackets. Modern compilers will often warn you about mismatched blocks.
8. **Non-Exhaustive Decisions:**
   When handling multiple cases in decision-making, ensure all possibilities are accounted for. For example, if you're dealing with values 1, 2, or 3, make sure the logic handles every case explicitly.

CATEGORY F: Interface Errors

1. **Parameter and Argument Matching:**
   Check that the number of arguments passed to a function matches the number of parameters expected, and that their types and order are correct.
2. **Data Type Consistency:**
   Ensure that the data types and sizes of parameters align with the corresponding arguments being passed. This avoids runtime errors due to mismatched types.
3. **Units System Consistency:**
   Verify that the units of parameters match the units of corresponding arguments, like ensuring angles are either in degrees or radians consistently throughout the code.
4. **Arguments to Modules:**
   Ensure that the number of arguments passed to modules or subroutines matches the number expected by the receiving module.
5. **Attributes Matching Between Modules:**
   When transmitting arguments between functions or modules, check that the attributes (e.g., data types) are consistent to avoid unexpected behavior.
6. **Correct Built-in Function Usage:**
   For built-in functions, ensure that you're providing the correct number and type of arguments in the proper order. Misusing these functions can cause runtime errors.
7. **Input-Only Parameters:**
   Make sure that parameters intended only for input are not being modified by the function. Avoid altering parameters unless explicitly intended.
8. **Global Variables Consistency:**
   If global variables are used, verify that their definitions and attributes are the same across all modules that reference them.

CATEGORY G: Input/Output Errors

1. **File Attributes:**
   If files are declared, check that their attributes (e.g., read/write permissions) are correct. However, the current code doesn't involve file handling.
2. **File Opening:**
   Ensure all files are opened before they are used, but this code doesn't handle file operations.
3. **Memory for File Reading:**
   If your program reads files, ensure that sufficient memory is allocated to handle them. This is not relevant to the current code.

4. **File Closure:**
   After using files, make sure they are properly closed. Again, this is not applicable in this case.
5. **End-of-File Detection:**
   If reading from files, handle end-of-file conditions correctly, though this is not relevant to the current code.
6. **I/O Error Handling:**
   Implement appropriate error handling for input/output operations, but no such operations are found in this code.
7. **Spelling and Grammar in Output:**
   Double-check any text printed or displayed by the program for spelling or grammatical errors. Minor spelling errors like "construtor" (should be "constructor") and "palindrow" (should be "palindrome") are present.

CATEGORY H: Other Checks

1. **Unused or Underused Variables:**
   Examine for variables that are declared but never referenced, or only referenced once. For example, the variable `found` in `signIn` is declared but never used effectively.
2. **Variable Attribute Listing:**
   Review the attributes of variables to make sure no unexpected default attributes are assigned. This ensures consistency in their use.
3. **Compiler Warnings:**
   If warnings or informational messages appear during compilation, review each one carefully. These often point to potential issues such as logic flaws or undeclared variables.
4. **Program Robustness:**
   Ensure that the program handles invalid inputs appropriately. For instance, input validation in functions like `rateProduct` could be improved by checking that rating points are within a valid range.
5. **Missing Functions:**
   The program could benefit from additional functionality, like a dedicated input validation function or better error handling for incorrect user credentials or invalid product IDs.

PROGRAM INSPECTION SUMMARY

1. **Errors Identified:**
   - Logic error in `isPrime`: Incorrect handling of prime number checks.
   - Unused variable `found` in `signIn`.
   - Uninitialized variable `len` in `averageRating`.

    - Incorrect handling of non-string/non-number types in `reverse`.
    - Potential undefined behavior in `likeProduct` if product ID doesn't exist.
    - Lack of input validation in functions like `rateProduct`.
    - Incorrect condition in `isEmpty` (should use `Array.isArray(value)`).
2. **Most Effective Inspection Category:**
   Logic errors and input/output errors are particularly effective in this context, as they help catch critical flaws in the program's functionality.
3. **Errors Not Identified by Program Inspection:**
   Runtime errors, such as accessing undefined properties, cannot be detected by static code review alone and would require dynamic testing.
4. **Applicability of Program Inspection:**
   Program inspection techniques are highly useful for identifying logical flaws, structural issues, and improving code quality. Combining these techniques with runtime testing would result in a more robust application.

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);
```

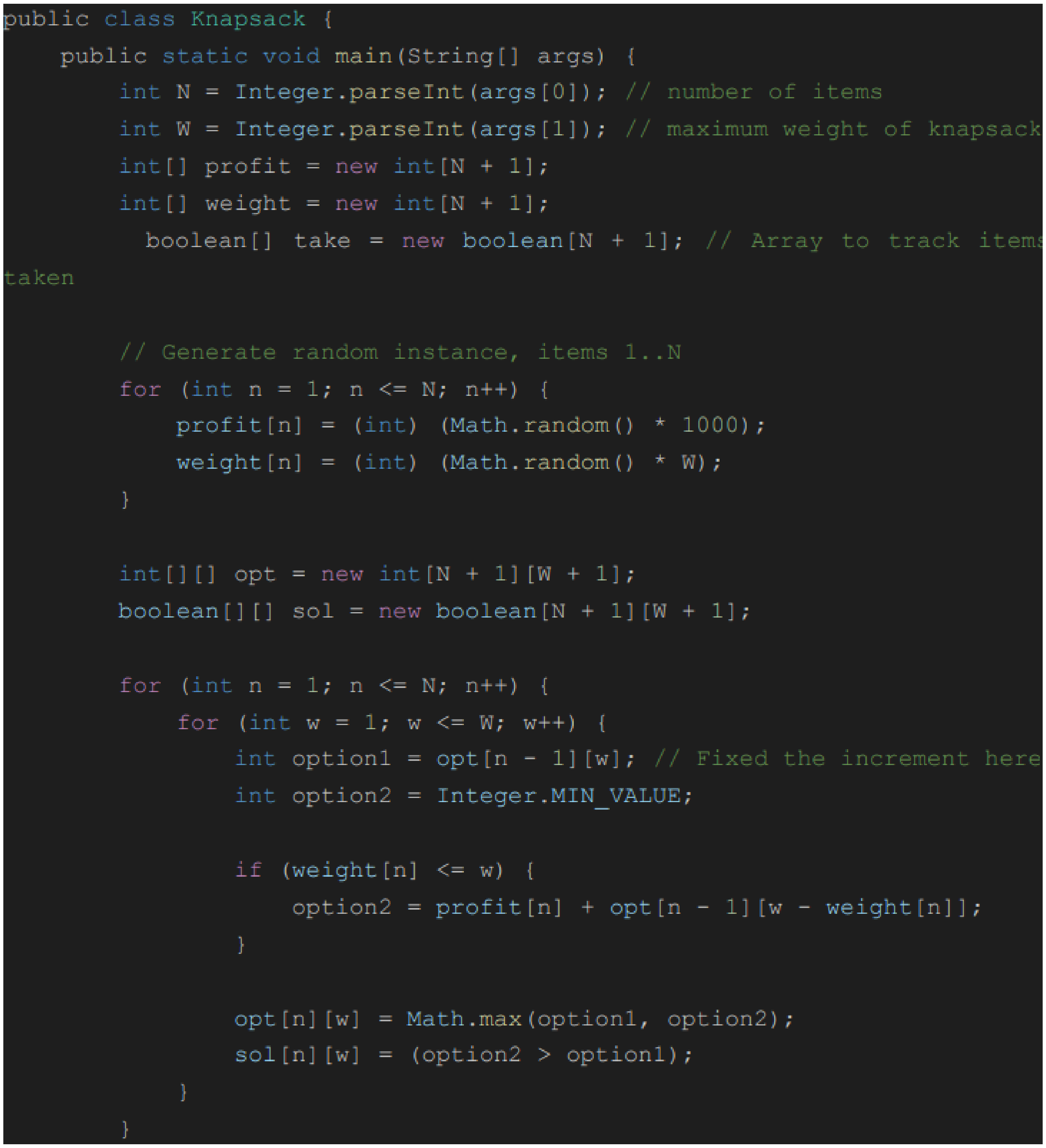
```
if (index !== -1) {

products[i].likes.splice(index, 1);

} else {

products[i].likes.push(userId);

}

}

}

};

console.log(likeProduct('aegfal', 'fg12cy'));

console.log(likeProduct('eedfcf', 'fg12cy'));
```

—---

[ii] CODE DEBUGGING

Knapsack

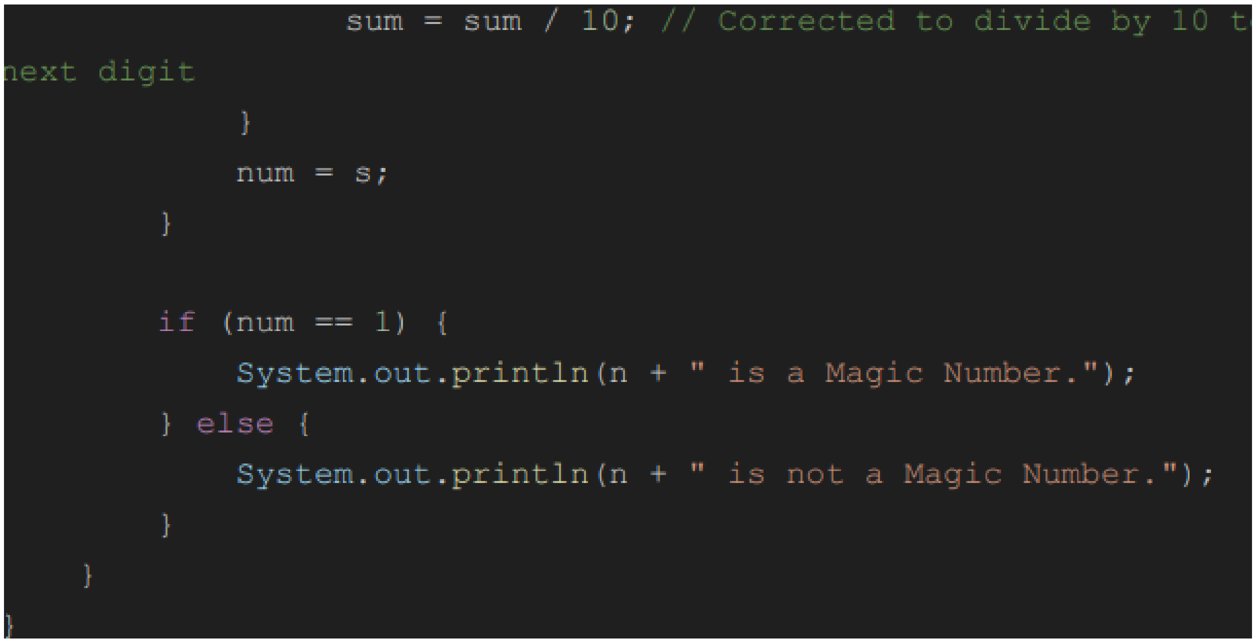
● There is one error in the program, as identified above.
● To fix this error, you would need one breakpoint at the line: int option1 = opt[n][w]; to ensure n and w are correctly used without unintended increments.



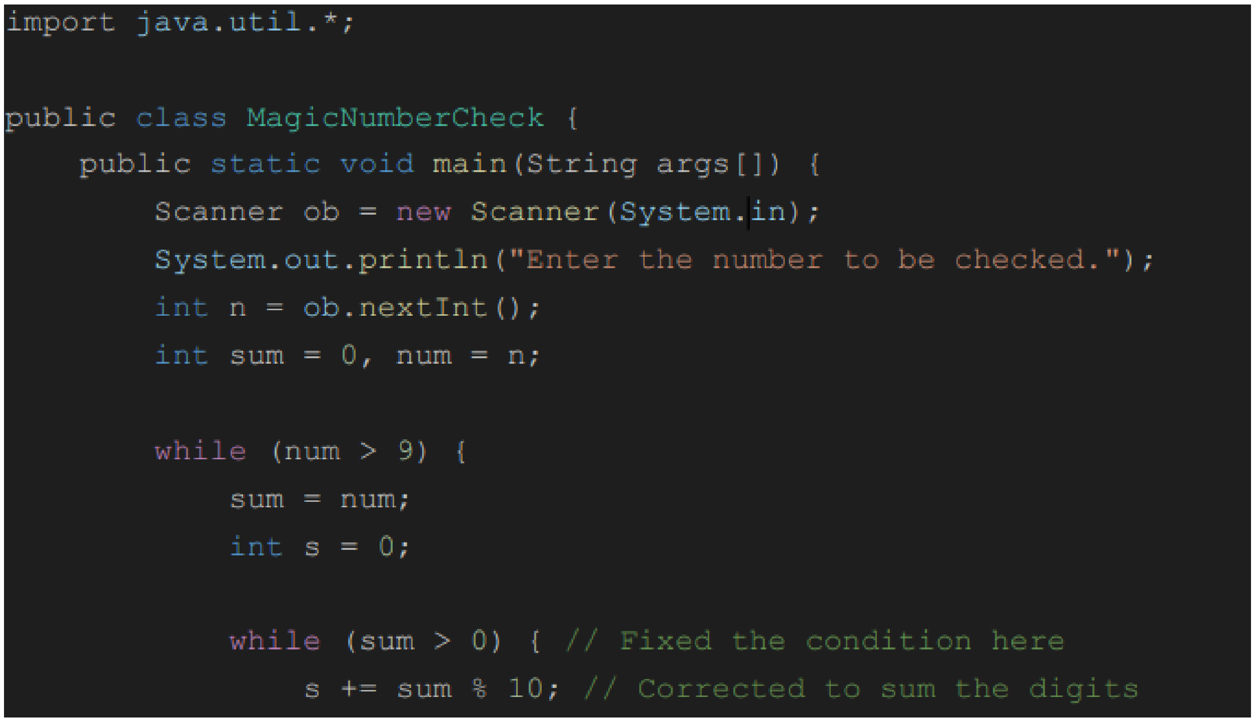
```
public class Knapsack {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]); // number of items
        int W = Integer.parseInt(args[1]); // maximum weight of knapsack
        int[] profit = new int[N + 1];
        int[] weight = new int[N + 1];
          boolean[] 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);
            }
        }
```

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

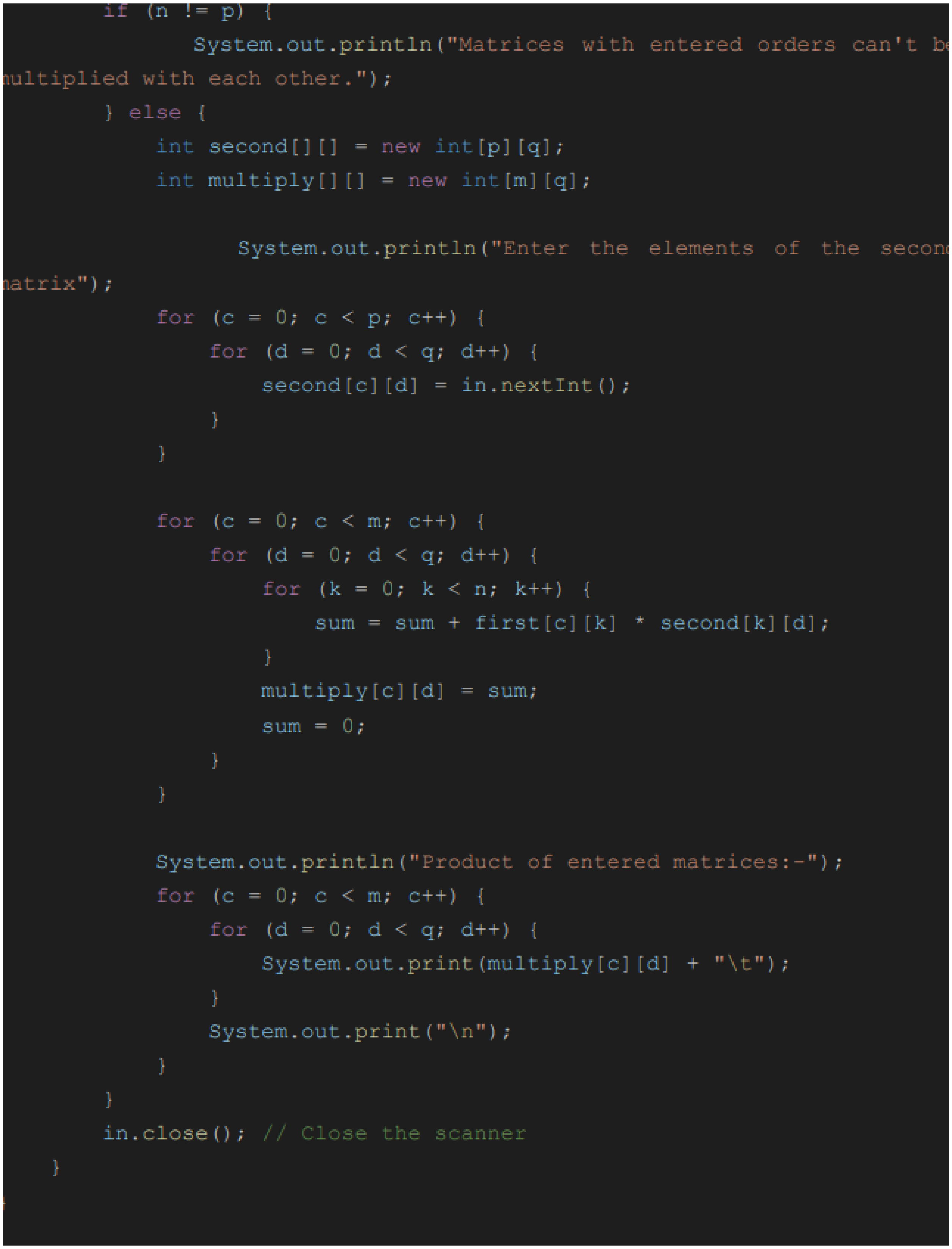


```
                  sum = sum / 10; // Corrected to divide by 10 t
next digit
            }
            num = s;
        }

        if (num == 1) {
            System.out.println(n + " is a Magic Number.");
        } else {
            System.out.println(n + " is not a Magic Number.");
        }
    }
}
```

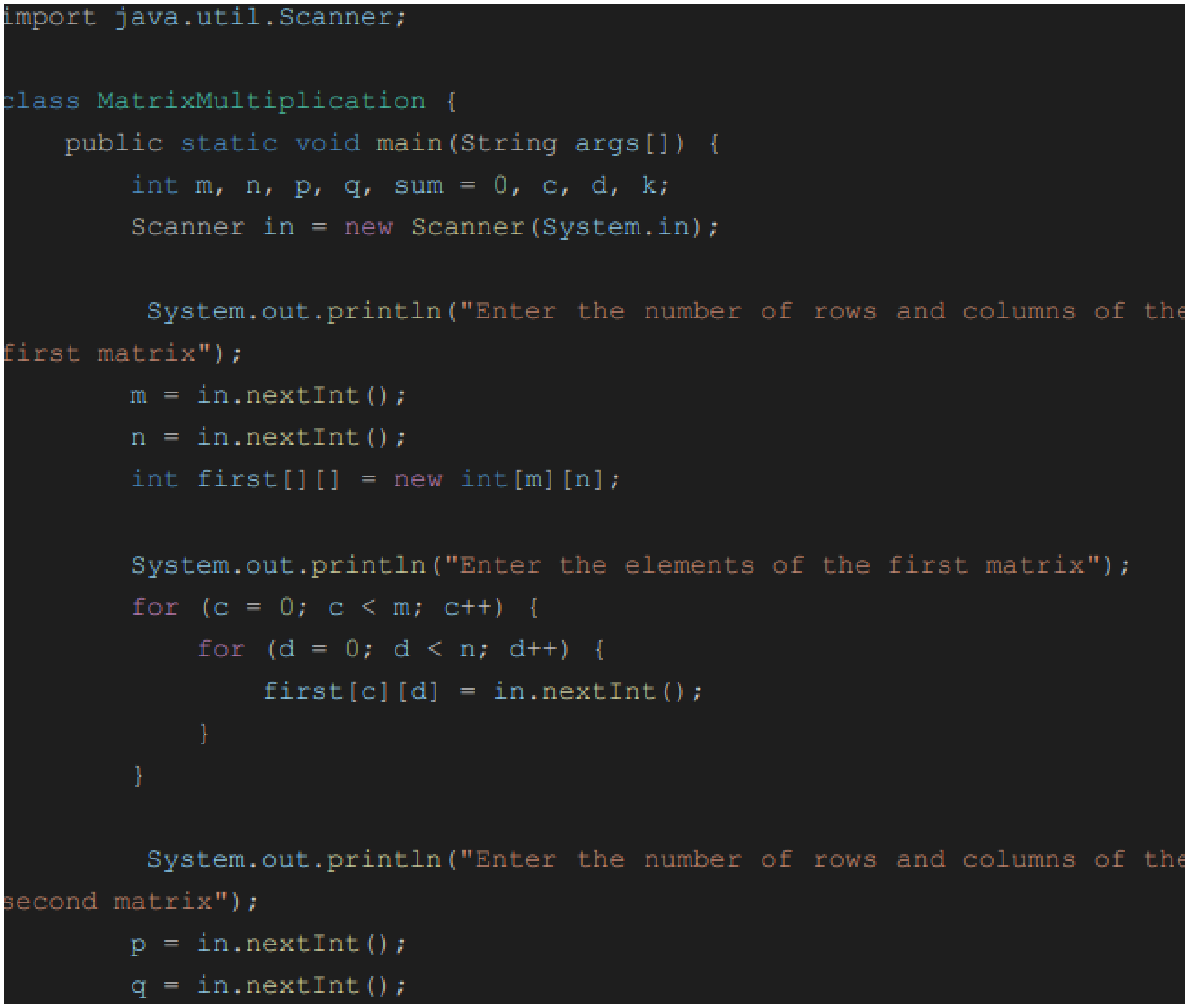


```
import java.util.*;

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
```

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.

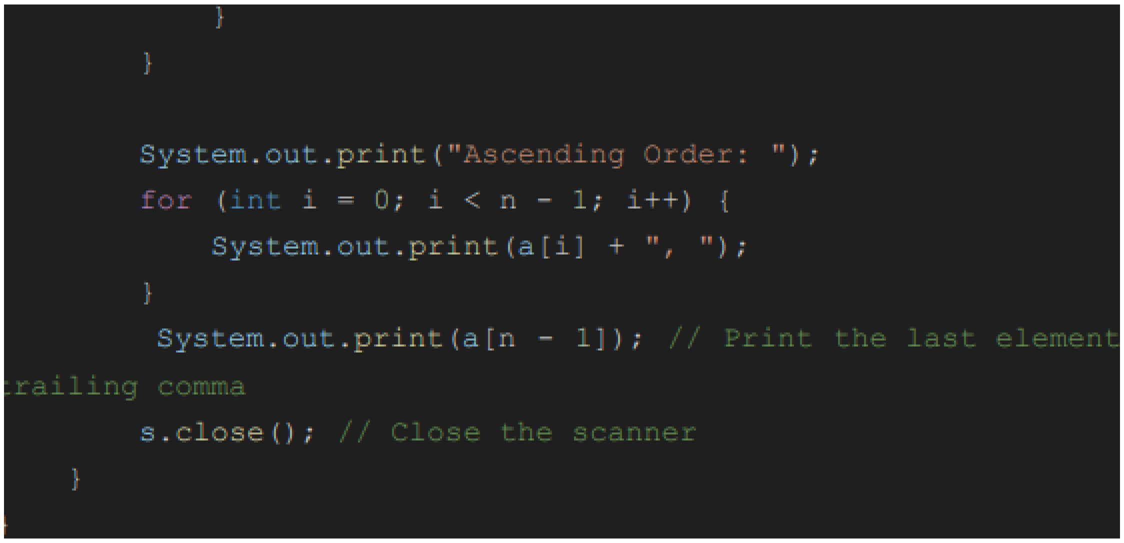


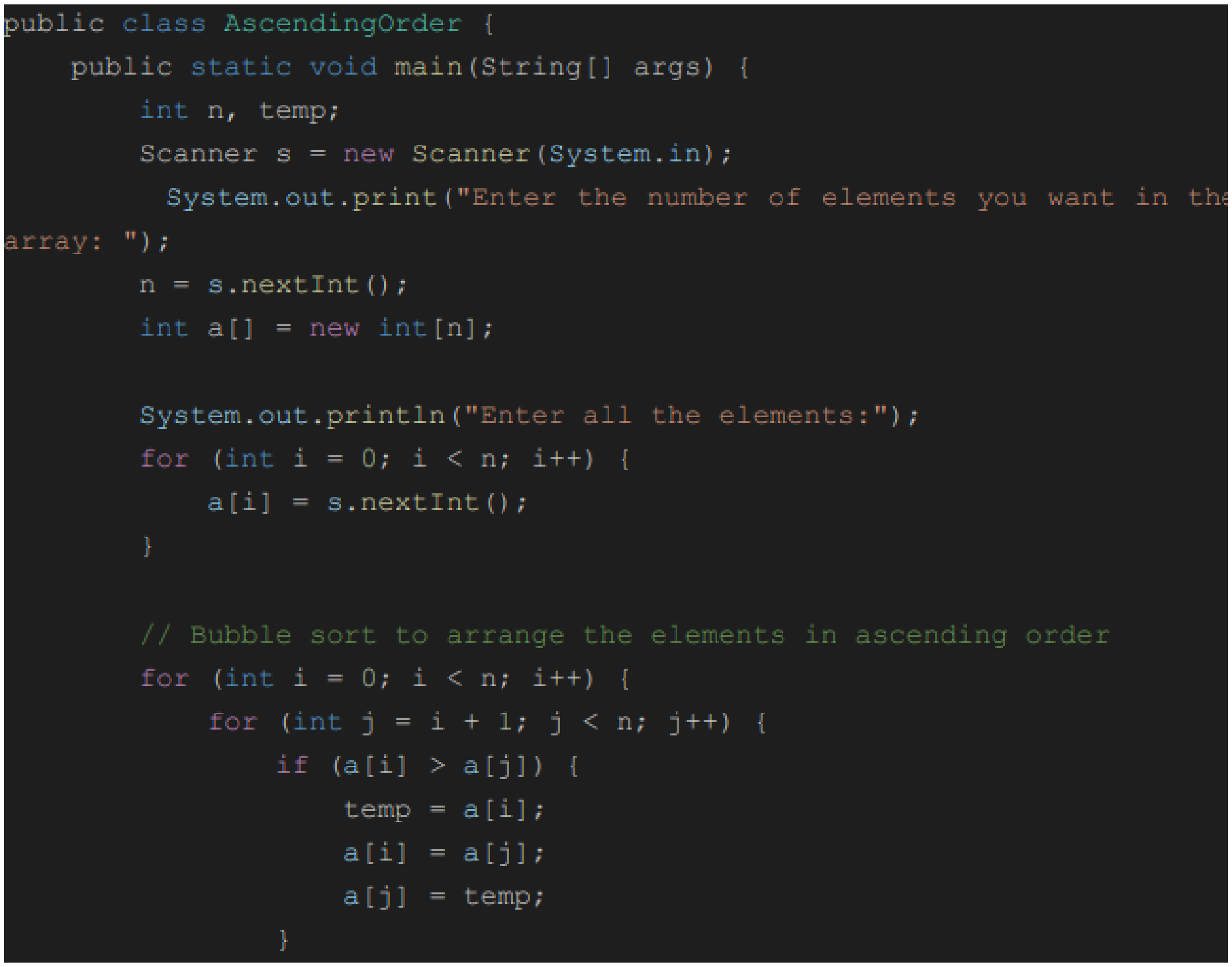
```
        if (n != p) {
               System.out.println("Matrices with entered orders can't be
ultiplied with each other.");
        } else {
            int second[][] = new int[p][q];
            int multiply[][] = new int[m][q];

                  System.out.println("Enter the elements of the secon
atrix");
            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
    }
}
```



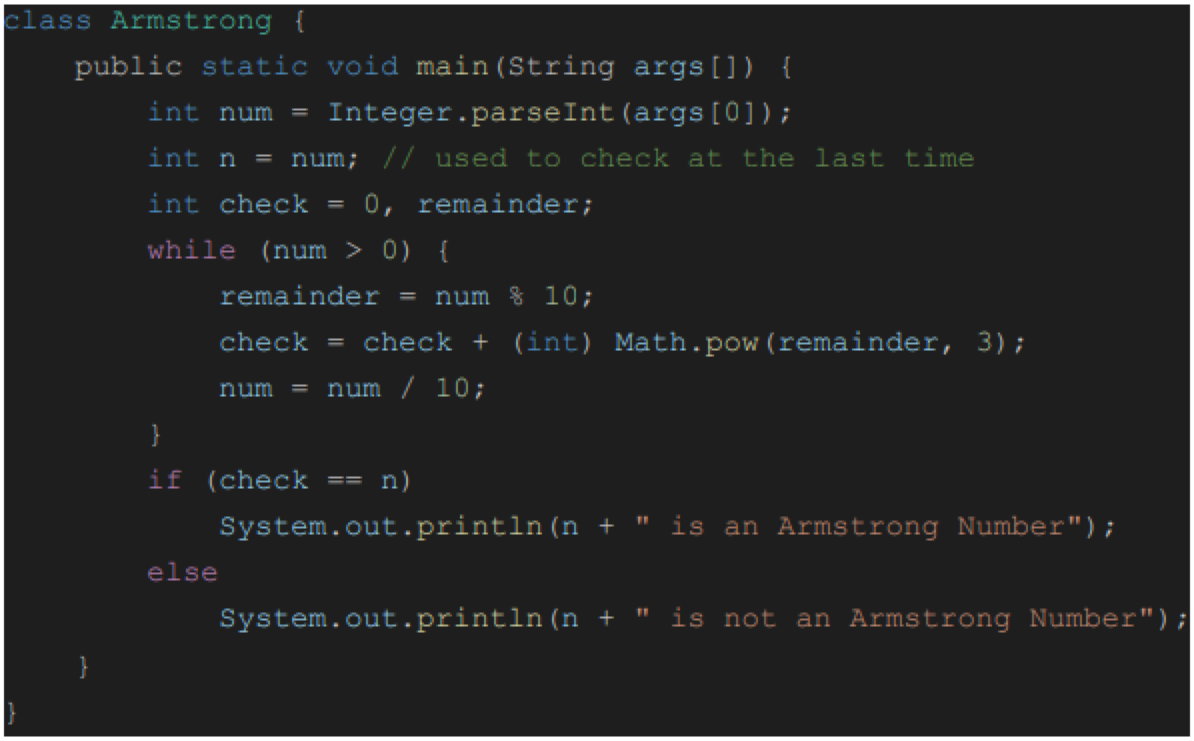
```
import java.util.Scanner;

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();
```

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.


```
            }
        }

        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
railing comma
        s.close(); // Close the 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;
                }
```

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.



```
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");
    }
}
```