Code Review Report

Report Information

Generated	2025-10-12 23:12:47
Files Reviewed	2
Model Used	qwen/qwen-2.5-coder-32b-instruct:free
Total Tokens	1684

Code Files Reviewed

File 1: factorial.js

Size: 709 bytes | Type: text/javascript

```
function calculateFactorial(n) {
if (n < 0) {
throw new Error("Factorial is not defined for negative numbers");
}
if (n === 0 || n === 1) {
return 1;
}
return n * calculateFactorial(n - 1);
}
```

```
function validateInput(input) {
const num = parseInt(input);
if (isNaN(num)) {
throw new Error("Invalid input: not a number");
}
return num;
}
```

```
// Main execution
try {
const userInput = prompt("Enter a number:");
const number = validateInput(userInput);
```

```
const result = calculateFactorial(number);
console.log(`Factorial of ${number} is ${result}`);
} catch (error) {
console.error("Error:", error.message);
}
```

File 2: fibonacci.py

Size: 516 bytes | Type: text/plain

```
def calculate_fibonacci(n):
"""Calculate the nth Fibonacci number"""
if n <= 1:
return n
return calculate_fibonacci(n-1) + calculate_fibonacci(n-2)
```

```
def main():
try:
num = int(input("Enter a number: "))
result = calculate_fibonacci(num)
print(f"Fibonacci({num}) = {result}")
except ValueError:
print("Please enter a valid number")
except Exception as e:
print(f"An error occurred: {e}")
```

```
if __name__ == "__main__":
main()
```

Code Quality & Readability

`factorial.js`
• **Clarity**: The code is clear and straightforward. The function names (`calculateFactorial` and `validateInput`) are descriptive.
• **Naming Conventions**: The naming conventions are consistent and follow JavaScript conventions.
• **Structure**: The code is well-structured with a clear separation between input validation, computation, and error handling.

`fibonacci.py`
• **Clarity**: The code is clear and easy to understand. The function `calculate_fibonacci` has a docstring explaining its purpose.
• **Naming Conventions**: The naming conventions are consistent and follow Python conventions.
• **Structure**: The code is well-structured with a clear separation between input handling, computation, and error handling.

Modularity & Architecture

`factorial.js`
• **Separation of Concerns**: The code is modular with separate functions for factorial calculation and input validation.
• **Reusability**: The `calculateFactorial` and `validateInput` functions can be reused in other parts of the application.

`fibonacci.py`
• **Separation of Concerns**: The code is modular with separate functions for Fibonacci calculation and input handling.
• **Reusability**: The `calculate_fibonacci` function can be reused in other parts of the application.

Potential Bugs

`factorial.js`
• **Edge Cases**: The code handles negative numbers and non-numeric inputs correctly.
• **Recursion Depth**: For large values of `n`, the recursive approach may lead to a stack overflow. Consider using an iterative approach or memoization.

`fibonacci.py`
• **Edge Cases**: The code handles non-integer inputs correctly.
• **Recursion Depth**: For large values of `n`, the recursive approach may lead to a stack overflow. Consider using an iterative approach or memoization.

Security Issues

`factorial.js`
• **Input Validation**: The code validates user input to ensure it is a number, which is good practice.
• **Error Handling**: The code catches errors and logs them, which is a good practice.

`fibonacci.py`
• **Input Validation**: The code validates user input to ensure it is an integer, which is good practice.
• **Error Handling**: The code catches errors and logs them, which is a good practice.

Performance Analysis

`factorial.js`
• **Efficiency**: The recursive approach is not efficient for large `n` due to repeated calculations. An iterative approach or memoization would improve performance.
• **Resource Usage**: The recursive approach uses a lot of stack space for large `n`.

`fibonacci.py`
• **Efficiency**: The recursive approach is not efficient for large `n` due to repeated calculations. An iterative approach or memoization would improve performance.
• **Resource Usage**: The recursive approach uses a lot of stack space for large `n`.

Best Practices

`factorial.js`
• **Error Handling**: Use specific error messages and types for better debugging.
• **Iterative Approach**: Consider using an iterative approach for better performance.
• **Memoization**: Implement memoization to cache results of previous calculations.

`fibonacci.py`
• **Error Handling**: Use specific error messages and types for better debugging.
• **Iterative Approach**: Consider using an iterative approach for better performance.
• **Memoization**: Implement memoization to cache results of previous calculations.

Improvement Suggestions

`factorial.js` 1. **Iterative Approach**:
function calculateFactorial(n) { if (n < 0) { throw new Error("Factorial is not defined for negative numbers"); } let result = 1; for (let i = 2; i <= n; i++) { result *= i; } return result; }

2. **Memoization**:
const factorialCache = {}; function calculateFactorial(n) { if (n < 0) { throw new Error("Factorial is not defined for negative numbers"); } if (n === 0 || n === 1) { return 1; } if (factorialCache[n]) { return factorialCache[n]; } factorialCache[n] = n * calculateFactorial(n - 1); return factorialCache[n]; }

`fibonacci.py` 1. **Iterative Approach**:
def calculate_fibonacci(n): """Calculate the nth Fibonacci number iteratively""" if n <= 1: return n a, b = 0, 1 for _ in range(2, n + 1): a, b = b, a + b return b

2. **Memoization**:
from functools import lru_cache

@lru_cache(maxsize=None) def calculate_fibonacci(n): """Calculate the nth Fibonacci number with memoization""" if n <= 1: return n return calculate_fibonacci(n-1) + calculate_fibonacci(n-2)

By implementing these suggestions, the code will be more efficient, maintainable, and robust.