Code Review Report

Report Information

Generated	2025-10-12 23:03:46
Files Reviewed	2
Model Used	qwen/qwen-2.5-coder-32b-instruct:free
Total Tokens	1521

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 easy to understand. The function names (`calculateFactorial`, `validateInput`) are descriptive and convey their purpose.
• **Naming Conventions**: The naming conventions are consistent and follow JavaScript conventions.
• **Structure**: The code is well-structured with a clear separation between input validation, factorial calculation, and main execution logic.

`fibonacci.py`
• **Clarity**: The code is clear and easy to understand. The function names (`calculate_fibonacci`, `main`) are descriptive and convey their purpose.
• **Naming Conventions**: The naming conventions are consistent and follow Python conventions.
• **Structure**: The code is well-structured with a clear separation between input validation, Fibonacci calculation, and main execution logic.

Modularity & Architecture

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

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

Potential Bugs

`factorial.js`
• **Negative Input Handling**: The code correctly throws an error for negative inputs.
• **Non-Integer Input Handling**: The code correctly throws an error for non-integer inputs.
• **Edge Cases**: The code handles edge cases for `n = 0` and `n = 1` correctly.

`fibonacci.py`
• **Negative Input Handling**: The code does not handle negative inputs. It should throw an error or handle them appropriately.
• **Non-Integer Input Handling**: The code correctly handles non-integer inputs by catching `ValueError`.
• **Edge Cases**: The code handles edge cases for `n = 0` and `n = 1` correctly.

Security Issues

`factorial.js`
• **Input Validation**: The code validates the input to ensure it is a non-negative integer.
• **Error Handling**: The code uses try-catch blocks to handle errors gracefully.

`fibonacci.py`
• **Input Validation**: The code validates the input to ensure it is an integer.
• **Error Handling**: The code uses try-except blocks to handle errors gracefully.

Performance Analysis

`factorial.js`
• **Efficiency**: The recursive approach is simple but not efficient for large values of `n` due to repeated calculations and stack overflow risk.
• **Optimization**: Consider using an iterative approach or memoization to improve performance.

`fibonacci.py`
• **Efficiency**: The recursive approach is simple but not efficient for large values of `n` due to repeated calculations and exponential time complexity.
• **Optimization**: Consider using an iterative approach or memoization to improve performance.

Best Practices

`factorial.js`
• **Use of `const` and `let`**: Use `const` for variables that do not change and `let` for those that do.
• **Error Messages**: Provide clear and descriptive error messages.
• **Comments**: Add comments to explain complex logic if necessary.

`fibonacci.py`
• **Docstrings**: Use docstrings to describe the purpose and usage of functions.
• **Error Messages**: Provide clear and descriptive error messages.
• **Comments**: Add comments to explain complex logic if necessary.

Improvement Suggestions

`factorial.js`
• **Iterative Approach**: Replace the recursive approach with an iterative one to improve performance and avoid stack overflow.
• **Memoization**: Implement memoization to cache results of previously computed factorials.
• **Code Comments**: Add comments to explain the logic, especially if the code becomes more complex.
• **Testing**: Add unit tests to verify the correctness of the `calculateFactorial` function.

`fibonacci.py`

• **Iterative Approach**: Replace the recursive approach with an iterative one to improve performance.
• **Memoization**: Implement memoization to cache results of previously computed Fibonacci numbers.
• **Negative Input Handling**: Add a check to handle negative inputs and provide an appropriate error message.
• **Code Comments**: Add comments to explain the logic, especially if the code becomes more complex.
• **Testing**: Add unit tests to verify the correctness of the `calculate_fibonacci` function.