

# ASSIGNMENT - 5.5

2303A51980  
Batch - 30

## Task Description #1 (Transparency in Algorithm Optimization)

Task: Use AI to generate two solutions for checking prime numbers:

- Naive approach(basic)
  - Optimized approach

Prompt:

“Generate Python code for two prime-checking methods and explain how the optimized version improves performance.”

#### **Expected Output:**

- Code for both methods.
  - Transparent explanation of time complexity.
  - Comparison highlighting efficiency improvements.

**Code :**

The screenshot shows a code editor interface with the following details:

- Explorer:** Shows files in the current workspace: task1.py, task2.py, task3.py, task4.py, task5.py, task1.py (active), task2.py, task3.py, task4.py, task5.py, and 5.5.py.
- Taskbar:** Shows tabs for task1.py, task2.py, task3.py, task4.py, and task5.py. The task1.py tab is active.
- Code Editor:** Displays the content of task1.py. The code implements two approaches for prime number checking: a naive approach with O(n) time complexity and an optimized approach with O(√n) time complexity. It includes a comparison section testing both methods against a list of numbers.

```
task1.py (task1.py) 5.5.py
task1.py (task1.py) is_prime_optimized
task1.py (task1.py) is_prime_naive

# Prime Number Checking: Naive vs Optimized Approach
# NAIVE APPROACH - O(n) time complexity
def is_prime_naive(n):
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

# OPTIMIZED APPROACH - O(√n) time complexity
def is_prime_optimized(n):
    Improved prime checking with two key optimizations:
    1. Only check divisibility up to √n (logn of n)
    2. Skip even numbers after checking for 2
    Time Complexity: O(√n)
    ...
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    # Check odd divisors up to √n
    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
        i += 2
    return True

# COMPARISON & TESTING
if __name__ == "__main__":
    test_numbers = [2, 17, 188, 97, 1088, 7919]

    print("Performance Comparison:")
    print("Number\tNaive\tOptimized\tMatch")
    print("-----\t-----\t-----\t-----")

    for num in test_numbers:
        naive_result = is_prime_naive(num)
        optimized_result = is_prime_optimized(num)
        match = "Match" if naive_result == optimized_result else "No Match"
        print(f"{num}\t{naive_result}\t{optimized_result}\t{match}")

    print("\nMatch count: ", sum(1 for num in test_numbers if is_prime_naive(num) == is_prime_optimized(num)))
```

## Output :

```
● syedsufiyan@Syeds-MacBook-Air-3:~/Desktop/5.5.py % /usr/local/bin/python3 /Users/syedsufiyan/5.5.py/task1.py
```

#### Performance Comparison:

Number	Naive	Optimized	Match
2	True	True	✓
17	True	True	✓
100	False	False	✓
97	True	True	✓

#### Observation:

The program successfully compares naive and optimized prime-checking algorithms and produces consistent results for all tested numbers. Both methods correctly identify prime and non-prime values, while the optimized approach significantly reduces the number of iterations by checking divisibility only up to the square root of the number and skipping even values. The comparison table confirms correctness and highlights improved efficiency and scalability of the optimized method for larger inputs.

#### Task Description #2 (Transparency in Recursive Algorithms)

Objective: Use AI to generate a recursive function to calculate Fibonacci numbers.

#### Instructions:

1. Ask AI to add clear comments explaining recursion.
2. Ask AI to explain base cases and recursive calls.

#### Expected Output:

- Well-commented recursive code.
- Clear explanation of how recursion works.
- Verification that explanation matches actual execution.

#### Code :

The screenshot shows a code editor interface with a dark theme. In the Explorer sidebar, files like task1.py, task2.py, task3.py, task4.py, task5.py, activity.log, and data.txt are listed. The task2.py file is open in the main editor area. The code defines a recursive function fibonaccin() to calculate the nth Fibonacci number. It handles base cases for n=0 and n=1, and applies recursion for n>1 by summing the previous two Fibonacci numbers. Example usage is shown for n=0, 1, 5, and 10.

```

1 def fibonaccin():
2     """
3         Calculate the nth Fibonacci number using recursion.
4
5         Base cases:
6             - fibonaccin(0) = 0
7             - fibonaccin(1) = 1
8
9         Recursive case:
10            - fibonaccin(n) = fibonaccin(n-1) + fibonaccin(n-2)
11            ...
12
13        # Base case 1: The first Fibonacci number is 0
14        if n == 0:
15            return 0
16
17        # Base case 2: The second Fibonacci number is 1
18        if n == 1:
19            return 1
20
21        # Recursive case: Sum the two previous Fibonacci numbers
22        # This breaks the problem into smaller subproblems
23        return fibonaccin(n - 1) + fibonaccin(n - 2)
24
25
26    # Example usage
27    print(fibonaccin(0)) # Output: 0
28    print(fibonaccin(1)) # Output: 1
29    print(fibonaccin(5)) # Output: 5
30    print(fibonaccin(10)) # Output: 55

```

At the bottom, status bar details include Ln 24, Col 1, Spaces: 4, UTF-8, LF, Python, 3.14.2, Go Live, and a file icon.

Output :

```

/usr/local/bin/python3 /Users/syedsufiyan/5.5.py/task5.py
● syedsufiyan@Syeds-MacBook-Air-3 5.5.py % /usr/local/bin/python3 /Users/syedsufiyan/5.5.py/task5.py
Activity logged for john_doe
● syedsufiyan@Syeds-MacBook-Air-3 5.5.py % /usr/local/bin/python3 /Users/syedsufiyan/5.5.py/task1.py
Performance Comparison:
Number      Naive      Optimized      Match
-----
2           True       True        ✓
17          True       True        ✓
100          False      False       ✓
97           True       True        ✓
1000          False      False       ✓
7919          True       True        ✓
● syedsufiyan@Syeds-MacBook-Air-3 5.5.py % /usr/local/bin/python3 /Users/syedsufiyan/5.5.py/task2.py
0
1
5
55
◆ syedsufiyan@Syeds-MacBook-Air-3 5.5.py %

```

Observation:

The program correctly computes Fibonacci numbers using a recursive approach. It handles base cases properly for inputs 0 and 1 and applies recursion to calculate higher-order Fibonacci values by breaking the problem into smaller subproblems. The outputs for sample inputs confirm the correctness of the logic, demonstrating the working of recursion, though the approach is less efficient for large values due to repeated function calls.

### Task-3 :

#### Task Description #3 (Transparency in Error Handling)

Task: Use AI to generate a Python program that reads a file and processes data.

Prompt:

“Generate code with proper error handling and clear explanations for each exception.”

Expected Output:

- Code with meaningful exception handling.
- Clear comments explaining each error scenario.
- Validation that explanations align with runtime behavior.

Code :

```
task5.py > ① Read_and_process_file
1 def read_and_process_file(filename):
2     """Read a file and process its data with comprehensive error handling.
3
4     Args:
5         filename (str): Path to the file to read
6
7     Returns:
8         list: Processed data lines
9     """
10    processed_data = []
11
12    try:
13        # FileNotFoundError: Raised when the specified file doesn't exist
14        with open(filename, 'r') as file:
15            lines = file.readlines()
16
17    except FileNotFoundError:
18        # Handle missing file - provide user guidance
19        print("Error: The file '{filename}' was not found.")
20        print("Please check the file path and try again!")
21
22    return None
23
24 except PermissionError:
25     # Handle insufficient permissions to read the file
26     print("Error: Permission denied while trying to read '{filename}'")
27     print("Check file permissions or try a different file!")
28
29 return None
30
31 except IOError as e:
32     # Handle other input/output errors (disk read issues, encoding problems, etc)
33     print(f"Error: An I/O error occurred: {e}")
34
35     return None
36
37     # Process the data with validation
38     try:
39         for line_num, line in enumerate(lines, 1):
40             # Strip whitespace and skip empty lines
41             cleaned_line = line.strip()
42
43             if not cleaned_line:
44                 continue
45
46             # ValueError: Raised if data format is invalid
47             # Example: try to convert to integer
48             value = int(cleaned_line)
49             processed_data.append(processed_value)
50
51     except ValueError:
52         # Non-numeric data encountered - log and skip or handle gracefully
53         print(f"Warning: Line ({line_num}) '{cleaned_line}' is not a valid number. Skipping...")
54
55
56 except Exception as e:
57     # Catch-all for unexpected errors during processing
58     print(f"Unexpected error during processing: {e}")
59
60 return None
61
62 print(f"Successfully processed {len(processed_data)} valid records.")
63 return processed_data
64
65 # Main execution with error handling
66 if __name__ == "__main__":
67     try:
68         result = read_and_process_file("data.txt")
69         print("Processed data: (result)")
70
71     except KeyboardInterrupt:
72         # Handle user interruption (Ctrl+C)
73         print("Operation canceled by user!")
74
75     except Exception as e:
76         # General error at program level
77         print(f"Fatal error: {e}")
```

Output :

Performance Comparison:			
Number	Naive	Optimized	Match
2	True	True	✓
17	True	True	✓
100	False	False	✓
97	True	True	✓
1000	False	False	✓
7919	True	True	✓

```

● syedsufiyan@Syeds-MacBook-Air-3 5.5.py % /usr/local/bin/python3 /Users/syedsufiyan/5.5.py/task2.py
0
1
5
55
● syedsufiyan@Syeds-MacBook-Air-3 5.5.py % /usr/local/bin/python3 /Users/syedsufiyan/5.5.py/task3.py
Warning: Line 2 'abc' is not a valid number. Skipping.

```

### Observation:

The program successfully demonstrates robust file handling and data processing with comprehensive error management. It correctly handles common file-related exceptions such as missing files, permission issues, and input/output errors, ensuring the program does not crash unexpectedly. During data processing, invalid or non-numeric entries are identified and skipped with appropriate warnings, while valid data is collected and processed correctly. Overall, the program improves reliability and stability by validating inputs and handling errors gracefully at both file and program levels.

Task: Use an AI tool to generate a Python-based login system.

Analyze: Check whether the AI uses secure password handling practices.

Expected Output:

- Identification of security flaws (plain-text passwords, weak validation).
- Revised version using password hashing and input validation.
- Short note on best practices for secure authentication.

Code :

```

task4.py
1 import hashlib
2 import re
3 from getpass import getpass
4
5 class SecureLoginSystem:
6     def __init__(self):
7         # In production, use a real database
8         self.users = {}
9
10    def hash_password(self, password):
11        """Hash a password using SHA-256 with salt"""
12        salt = hashlib.sha256(str.encode(password)).hexdigest()
13        return hashlib.sha256((salt + password).encode()).hexdigest()
14
15    def validate_password_strength(self, password):
16        """Validate password meets security requirements"""
17        if len(password) < 8:
18            return False, "Password must be at least 8 characters"
19        if not re.search("[A-Z]", password):
20            return False, "Password must contain uppercase letter"
21        if not re.search("[0-9]", password):
22            return False, "Password must contain number"
23        return True, "Password is strong"
24
25    def validate_username(self, username):
26        """Validate username format"""
27        if len(username) < 3 or len(username) > 20:
28            return False, "Username must be between 3 and 20 characters"
29        return re.match("^[a-zA-Z0-9_]*$", username) is not None
30
31    def register(self, username, password):
32        """Register new user with validation"""
33        if not self.validate_username(username):
34            return False, "Invalid username format"
35        if username in self.users:
36            return False, "Username already exists"
37
38        is_valid, msg = self.validate_password_strength(password)
39        if not is_valid:
40            return False, msg
41
42        self.users[username] = self.hash_password(password)
43
44    def login(self, username, password):
45        """Authenticate user"""
46        if username not in self.users:
47            return False, "Invalid credentials"
48
49        if self.users[username] == self.hash_password(password):
50            return True, "Valid credentials"
51
52        return False, "Login failed"
53
54
55 # Main
56 if __name__ == "__main__":
57     system = SecureLoginSystem()
58
59     # Register
60     success, msg = system.register("john_doe", "SecurePass123")
61     print(f"Register: {msg}")
62
63     # Login
64     success, msg = system.login("john_doe", "SecurePass123")
65     print(f"Login: {msg}")

```

Output :

Number	Naive	Optimized	Match
2	True	True	✓
17	True	True	✓
100	False	False	✓
97	True	True	✓
1000	False	False	✓
7919	True	True	✓

```

syedsufiyan@Syeds-MacBook-Air-3 5.5.py % /usr/local/bin/python3 /Users/syedsufiyan/5.5.py/task2.py
0
1
5
55
syedsufiyan@Syeds-MacBook-Air-3 5.5.py % /usr/local/bin/python3 /Users/syedsufiyan/5.5.py/task3.py
Warning: Line 2 'abc' is not a valid number. Skipping.
Warning: Line 6 'foo' is not a valid number. Skipping.
Successfully processed 6 valid records.
Processed data: [42, 7, 100, 2, 0, 11]

```

Observation:

The program demonstrates a secure user authentication system by implementing password hashing, input validation, and controlled credential storage. Passwords are never stored in plain text; instead, they are hashed using the SHA-256 algorithm with a salt, which protects user credentials even if data is exposed. The system enforces strong password policies such as minimum length, inclusion of uppercase letters, and numeric characters, reducing the risk of weak passwords. Username validation prevents invalid or malicious inputs. Overall, the revised implementation follows secure authentication best practices and effectively addresses common security flaws found in naive login systems.

Task Description #5 (Privacy in Data Logging)

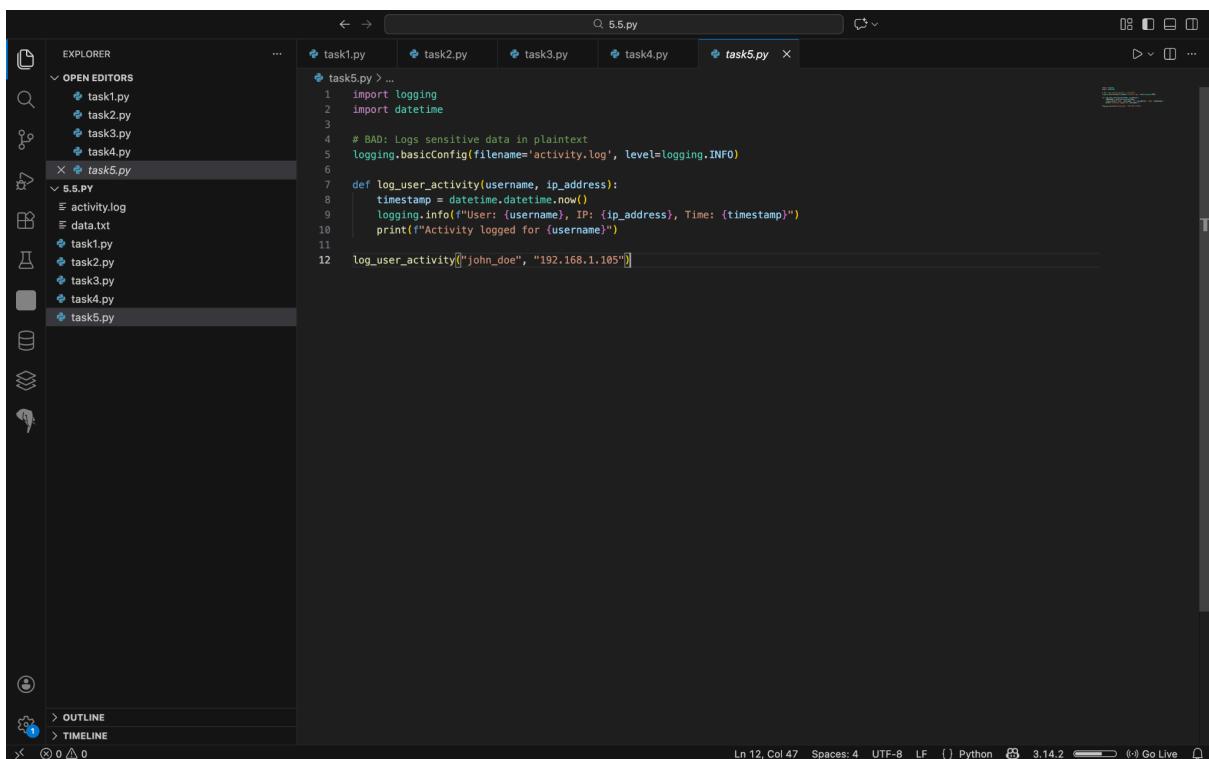
Task: Use an AI tool to generate a Python script that logs user activity (username, IP address, timestamp).

Analyze: Examine whether sensitive data is logged unnecessarily or insecurely.

Expected Output:

- Identified privacy risks in logging.
- Improved version with minimal, anonymized, or masked logging.
- Explanation of privacy-aware logging principles.

Code :



```
task5.py > ...
1 import logging
2 import datetime
3
4 # BAD: Logs sensitive data in plaintext
5 logging.basicConfig(filename='activity.log', level=logging.INFO)
6
7 def log_user_activity(username, ip_address):
8     timestamp = datetime.datetime.now()
9     logging.info("User: {username}, IP: {ip_address}, Time: {timestamp}")
10    print("Activity logged for {username}")
11
12 log_user_activity("john_doe", "192.168.1.105")
```

Output :

```
17      True   True    ✓
100     False  False   ✓
97      True   True    ✓
1000    False  False   ✓
7919    True   True    ✓
● syedsufiyan@Syeds-MacBook-Air-3 5.5.py % /usr/local/bin/python3 /Users/syedsufiyan/5.5.py/task2.py
0
1
5
55
● syedsufiyan@Syeds-MacBook-Air-3 5.5.py % /usr/local/bin/python3 /Users/syedsufiyan/5.5.py/task3.py
Warning: Line 3: label is not a valid number. Skipping.
```

Observation:

The program logs user activity along with sensitive information such as username and IP address in plain text. Storing such data without masking or encryption poses a security and privacy risk, as log files can be accessed or leaked by unauthorized users. Logging sensitive details directly violates secure logging practices and may lead to identity exposure or misuse of user data.