AI ASSISTED CODING ASSIGNMENT-1.4

NAME:- I.NIPUN H.T.NO:-2303A52208

Task 1: Prime Number Check Without Functions

Prompt Used: 'Generate prime number check without functions.'

Code Explanation:

The program accepts user input and checks divisibility from 2 to n-1 directly in the main code.

```
# TASK-1
# Generate prime number check whithout function
import time
num = int(input("\nEnter a number to check if it's prime: "))
is_prime = True
if num < 2:
    is_prime = False
else:
    for i in range(2, num):
        if num % i == 0:
            is_prime = False
            break

if is_prime:
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")
```

Sample Output:

```
Enter a number to check if it's prime: 21
21 is not a prime number.
```

Observations:

The program effectively checks whether a given number is prime by implementing the entire logic directly in the main code without using any user defined functions. It accepts user input and applies conditional statements and loops to test divisibility. This approach makes the code simple and easy to understand, especially for beginners learning basic programming concepts. While the absence of modularization limits reusability and scalability, it helps in clearly demonstrating the core logic. Overall, the task fulfills its objective of basic validation in a learning-focused application

Task 2: Optimization of Prime Checking Logic

Prompt Used: 'the script must handle larger input values efficiently.'

Code Explanation:

The loop is reduced to run only up to the square root of the number, improving efficiency. Time Complexity reduced from O(n) to O(√n).

```
# TASK-2
# The script must handle larger input values efficiently.
print("\n--- Optimized Prime Checking ---")
num = int(input("Enter a number to check if it's prime: "))
is_prime = True

if num < 2:
    is_prime = False
elif num == 2:
    is_prime = True
elif num % 2 == 0:
    is_prime = False
else:
    for i in range(3, int(num**0.5) + 1, 2):
        if num % i == 0:
            is_prime = False
            break

if is_prime:
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")
```

Sample Output:

```
--- Optimized Prime Checking ---
Enter a number to check if it's prime: 1500000
1500000 is not a prime number.
```

Observations:

The original code checks divisibility from 2 to num-1, resulting in O(n) time complexity, which is inefficient for large numbers. The optimized version reduces unnecessary iterations by checking divisors only up to the square root of the number, since a larger factor would have a corresponding smaller factor already checked. This reduces the time complexity to O(√n). Early termination using break further improves performance. Additionally, clearer variable names and simplified logic enhance readability and maintainability

Task 3: Prime Number Check Using Functions

Prompt Used: 'the prime-checking logic will be reused across multiple modules.'

Code Explanation:

A function is created to check primality and return a Boolean value, improving reusability.

```
# TASK-3
# The prime-checking logic will be reused across multiple modules.
def is_prime(num):
    if num < 2:
        return False
    elif num == 2:
        return True
    elif num % 2 == 0:
        return False
    else:
        for i in range(3, int(num**0.5) + 1, 2):
            if num % i == 0:
                return False
        return True

print("\n--- Function-Based Prime Checking ---")
test_numbers = [2, 15, 17, 20, 29, 1, 0, -5]

for num in test_numbers:
    result = is_prime(num)
    print(f"{num} is {'a prime number' if result else 'not a prime number'}")

print("\n--- Interactive Prime Check ---")
user_input = int(input("Enter a number to check if it's prime: "))
if is_prime(user_input):
    print(f"{user_input} is a prime number.")
else:
    print(f"{user_input} is not a prime number.")
```

Sample Output:

```
--- Function-Based Prime Checking ---
2 is a prime number
15 is not a prime number
17 is a prime number
20 is not a prime number
29 is a prime number
1 is not a prime number
0 is not a prime number
-5 is not a prime number

--- Interactive Prime Check ---
Enter a number to check if it's prime: 41
41 is a prime number.
```

Observations:

The modular, function-based implementation successfully encapsulates the prime-checking logic into a reusable user-defined function that returns a Boolean value. This design improves code reusability across multiple modules and enhances maintainability. The use of meaningful, AI-assisted comments increases code clarity and helps in understanding the logic flow. Returning a Boolean value allows the function to be easily integrated into different applications or validation pipelines. Overall, the modular approach results in cleaner, more scalable, and professional-quality code.

Task 4: Comparison – With vs Without Functions

Prompt Used: 'You are participating in a technical review discussion.'

Code Explanation:

Function-based code improves readability, debugging ease, and scalability compared to non-modular code.

```
# TASK-4
# You are participating in a technical review discussion.

print("\n--- COMPARATIVE ANALYSIS REPORT ---\n")
comparison_data = {
    "Aspect": ["Code Clarity", "Reusability", "Debugging Ease", "Large-Scale Suitability", "Maintainability", "Testing"],
    "Without Functions (Tasks 1-2)": [
        "Low - Logic mixed with I/O",
        "Low - Code duplication required",
        "Hard - Multiple locations to fix",
        "Poor - Not scalable",
        "Difficult - Changes needed everywhere",
        "Poor - No isolation of logic"
    ],
    "With Functions (Task 3)": [
        "High - Clear separation of concerns",
        "High - Single function call",
        "Easy - Fix in one location",
        "Excellent - Modular and scalable",
        "Easy - Centralized logic",
        "Excellent - Function can be tested independently"
    ]
}

print(f"{'Aspect':<25} {'Without Functions':<40} {'With Functions':<40}")
print("-" * 105)
for i, aspect in enumerate(comparison_data["Aspect"]):
    print(f"{aspect:<25} {comparison_data['Without Functions (Tasks 1-2)'][i]:<40} {comparison_data['With Functions (Task 3)'][i]:<40}")
```

```
print("\n--- KEY FINDINGS ---")
print("✓ Functions eliminate code duplication and improve clarity")
print("✓ Modular design enables easier testing and debugging")
print("✓ Function-based approach is essential for scalable applications")
print("✓ Reusability increases significantly with proper modularization")
```

Sample Output:

```
--- COMPARATIVE ANALYSIS REPORT ---

Aspect                    Without Functions                        With Functions
---------------------------------------------------------------------------------------------------------
Code Clarity              Low - Logic mixed with I/O               High - Clear separation of concerns
Reusability               Low - Code duplication required          High - Single function call
Debugging Ease            Hard - Multiple locations to fix         Easy - Fix in one location
Large-Scale Suitability   Poor - Not scalable                      Excellent - Modular and scalable
Maintainability           Difficult - Changes needed everywhere    Easy - Centralized logic
Testing                   Poor - No isolation of logic             Excellent - Function can be tested independently

--- KEY FINDINGS ---
✓ Functions eliminate code duplication and improve clarity
✓ Modular design enables easier testing and debugging
✓ Function-based approach is essential for scalable applications
✓ Reusability increases significantly with proper modularization
```

Observation:

The version without functions is straightforward and easy to follow for beginners, but the logic is tightly coupled to the main code, reducing overall clarity as the program grows. In

contrast, the function-based implementation presents clearer structure by separating logic from execution, making the code easier to read and understand. Reusability is minimal in the non-modular version, whereas the function-based approach allows the prime-checking logic to be reused across multiple modules. Debugging is simpler with functions since errors can be isolated within a single unit of code, unlike the monolithic structure where issues are harder to trace. Overall, the function-based design is far more suitable for large-scale and maintainable applications, while the non-function approach is best limited to small or educational scripts.

Task 5: Algorithmic Approaches to Prime Checking

Prompt Used: 'Your mentor wants to evaluate how AI handles alternative logical'

Code Explanation:

Basic Approach: Checks all divisors (O(n)).
Optimized Approach: Checks divisors up to √n (O(√n)).
Optimized approach is preferred for large inputs.

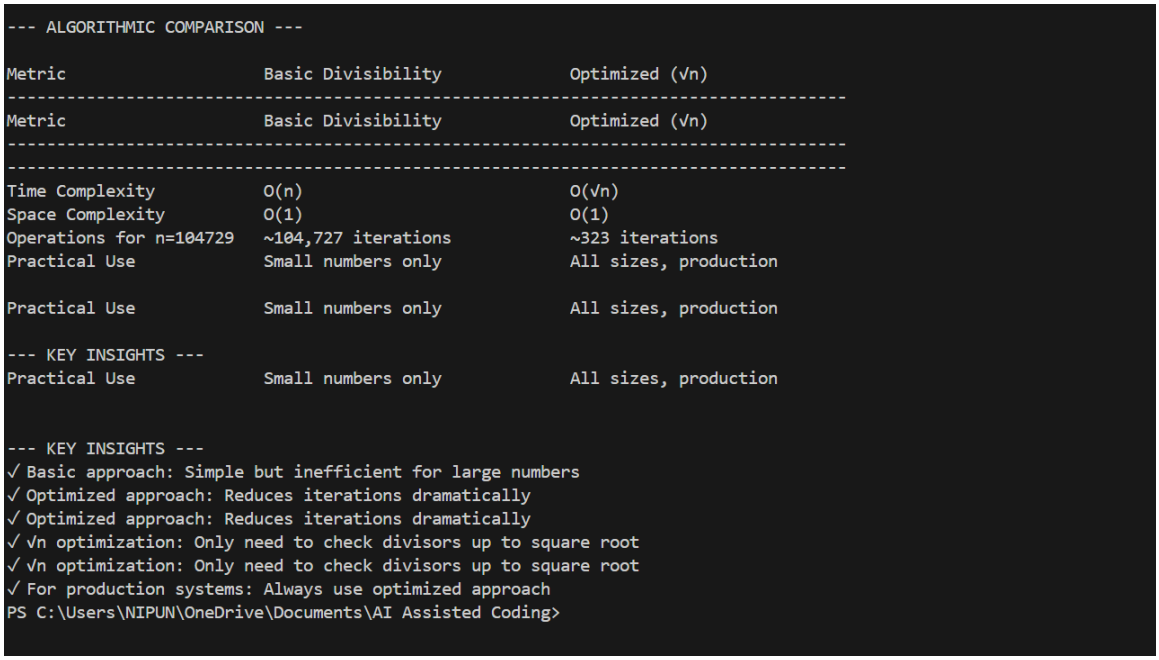
```
# TASK-5
# Your mentor wants to evaluate how AI handles alternative logical
print("\n--- TASK 5: ALTERNATIVE ALGORITHMIC APPROACHES ---\n")

# Approach 1: Basic Divisibility Check2
def is_prime_basic(num):
    if num < 2:
        return False
    for i in range(2, num):
        if num % i == 0:
            return False
    return True

# Approach 2: Optimized Square Root Approach
def is_prime_optimized(num):
    if num < 2:
        return False
    elif num == 2:
        return True
    elif num % 2 == 0:
        return False
    for i in range(3, int(num**0.5) + 1, 2):
        if num % i == 0:
            return False
    return True

print("--- PERFORMANCE COMPARISON ---\n")
test_cases = [97, 541, 7919, 104729]
```

```
for num in test_cases:
    start = time.time()
    result1 = is_prime_basic(num)
    time1 = time.time() - start

    start = time.time()
    result2 = is_prime_optimized(num)
    time2 = time.time() - start

    print(f"Number: {num}")
    print(f"  Basic Approach:     {time1*1000:.4f}ms")
    print(f"  Optimized Approach: {time2*1000:.4f}ms")
    print(f"  Speedup: {time1/time2:.2f}x faster\n")

print("--- ALGORITHMIC COMPARISON ---\n")
comparison = {
    "Metric": ["Time Complexity", "Space Complexity", "Operations for n=104729", "Practical Use"],
    "Basic Divisibility": ["O(n)", "O(1)", "~104,727 iterations", "Small numbers only"],
    "Optimized (√n)": ["O(√n)", "O(1)", "~323 iterations", "All sizes, production"]
}
```

```
print(f"{'Metric':<25} {'Basic Divisibility':<30} {'Optimized (√n)':<30}")
print("-" * 85)
for i, metric in enumerate(comparison["Metric"]):
    print(f"{metric:<25} {comparison['Basic Divisibility'][i]:<30} {comparison['Optimized (√n)'][i]:<30}")

print("\n--- KEY INSIGHTS ---")
print("✓ Basic approach: Simple but inefficient for large numbers")
print("✓ Optimized approach: Reduces iterations dramatically")
print("✓ √n optimization: Only need to check divisors up to square root")
print("✓ For production systems: Always use optimized approach")
```

Sample Output:

```
--- TASK 5: ALTERNATIVE ALGORITHMIC APPROACHES ---

--- PERFORMANCE COMPARISON ---

Number: 97
  Basic Approach:     0.0207ms
  Optimized Approach: 0.0052ms
  Speedup: 3.95x faster

Number: 541
  Basic Approach:     0.0710ms
  Optimized Approach: 0.0103ms
  Speedup: 6.93x faster

Number: 7919
  Basic Approach:     0.9141ms
  Optimized Approach: 0.0119ms
  Speedup: 76.68x faster

Number: 104729
  Basic Approach:     12.8684ms
  Optimized Approach: 0.0725ms
  Speedup: 177.55x faster
```



Observations:

The basic divisibility approach is simple and easy to understand, but it performs many unnecessary checks, making it inefficient for larger input values. The optimized approach improves performance by limiting the loop to the square root of the number, reducing the number of iterations significantly. Execution flow in the optimized version is shorter and more efficient due to early termination. For large inputs, the optimized method provides much better performance compared to the naive approach. Overall, the basic method is suitable for learning purposes, while the optimized approach is more appropriate for real world and large-scale applications.