

Lab - 01 Assignment

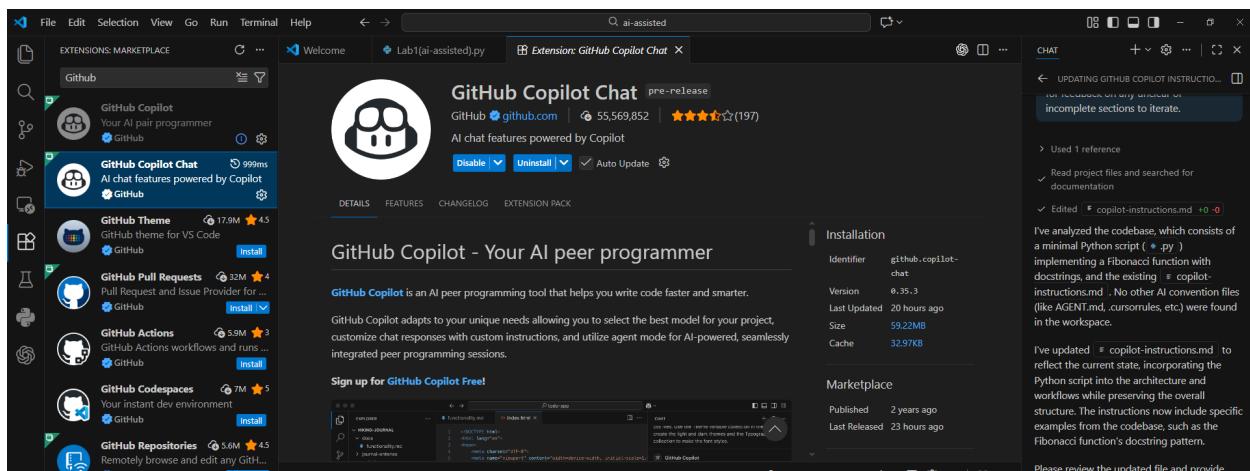
Name: Zobiya Fatima

Roll No: 2303A51879

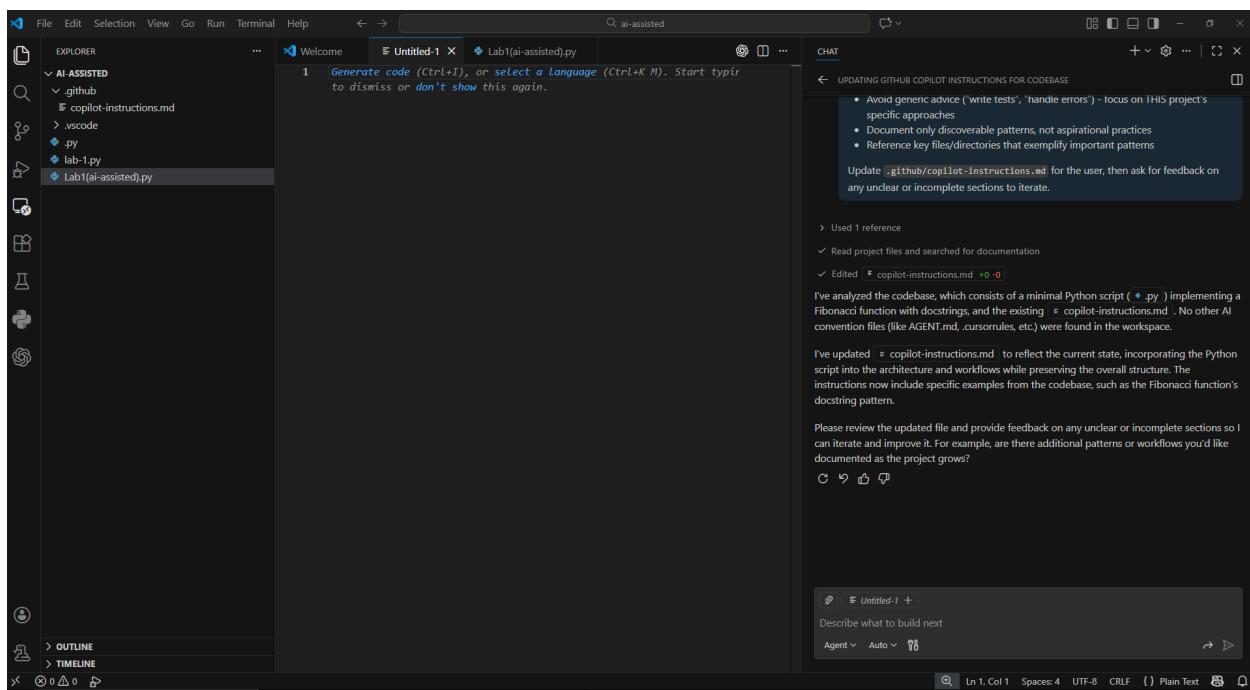
Batch No: 14

Task 0

- Install and configure GitHub Copilot in VS Code. Take screenshots of each step.



- Install and configure GitHub Copilot in VS Code. Take screenshots of each step.



Task 1: AI-Generated Logic Without Modularization (Prime Number Check Without Functions)

❖ Scenario

- You are developing a basic validation script for a numerical learning Application.

❖ Task Description

Use GitHub Copilot to generate a Python program that:

- Checks whether a given number is prime
- Accepts user input
- Implements logic directly in the main code
- Does not use any user-defined functions

❖ Expected Output

- Correct prime result

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows a folder named "RUN AND DEBUG" containing a file "ai-assisted.py".
- Editor:** Displays the following Python code:

```
1  """Checks whether a given number is prime or not.
2  Implements logic directly in the main code and does not use any user-defined functions.
3  """
4
5  try:
6      n = int(input("Enter an integer: ").strip())
7  except Exception:
8      print("Invalid input")
9  else:
10     if n <= 1:
11         print(f"{n} is not prime")
12     elif n == 2:
13         print(f"{n} is prime")
14     elif n % 2 == 0:
15         print(f"{n} is not prime")
16     else:
17         is_prime = True
18         i = 3
19         import math
20         limit = int(math.sqrt(n))
21         while i <= limit:
22             if n % i == 0:
23                 is_prime = False
24                 break
25             i += 2
26         print(f"{n} is prime" if is_prime else 'not prime')
```
- Terminal:** Shows the command line output:

```
PS D:\ai-assisted> *c
PS D:\ai-assisted>
PS D:\ai-assisted> d; cd 'd:\ai-assisted'; & 'D:\WinPython\WinPython64-3.12.10.isilin\Py64-312101\python\python.exe' 'c:\Users\VHP\vscode\extensions\ms-python.python\debugpy-2025.18.0-win32-x64\bundled\launcher' '51348' '--' 'D:\ai-assisted\ai-assisted.py'
Enter an Integer: 3
3 is prime
PS D:\ai-assisted>
```
- Breakpoints:** A single breakpoint is set at line 17.
- References:** A sidebar on the right lists references, including "Used 2 references" and "Used 1 reference".
- Chat:** A sidebar on the right shows a conversation about the task requirements.

➤ Non-prime result

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows a folder named "RUN".
- Editor:** Displays a Python file named "Lab1(ai-assisted).py" containing code to check if a number is prime. The code includes logic to handle invalid input and numbers less than 2.
- Terminal:** Shows the command line output of running the script, which correctly identifies 4 as not prime.
- Breakpoints:** A breakpoint is set at line 4.
- Chat:** An AI-generated logic without modularization is shown in the chat, along with references to existing code.
- Code Snippet:** A snippet of AI-generated code for a prime number check is visible on the right.

➤ Screenshots showing Copilot-generated code suggestions

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows a folder named "RUN".
- Editor:** Displays a Python file named "Lab1(ai-assisted).py" with a code suggestion overlay. The suggestion is to accept user input and check if the number is prime.
- Terminal:** Shows the command line output of running the script, which correctly identifies 4 as not prime.
- Code Snippet:** A snippet of AI-generated code for a prime number check is visible on the right.

Task 2: Efficiency & Logic Optimization (Cleanup)

❖ Scenario

The script must handle larger input values efficiently.

❖ Task Description

Review the Copilot-generated code from Task 1 and improve it by:

- Reducing unnecessary iterations
- Optimizing the loop range (e.g., early termination)
- Improving readability
- Use Copilot prompts like:
 - “Optimize prime number checking logic”
 - “Improve efficiency of this code”

Hint:

Prompt Copilot with phrases like

“optimize this code”, “simplify logic”, or “make it more readable”

❖ Expected Output

➤ Original

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows a folder named "RUN AND DEBUG".
- Run and Debug View:** Shows a configuration named "Run and Debug".
- Editor:** Displays the following Python code:

```
py > ...
1  """Checks whether a given number is prime or not.
2  Implements logic directly in the main code and does not use any user-defined functions.
3  """
4
5  try:
6      n = int(input("Enter an Integer: ").strip())
7  except Exception:
8      print("Invalid input")
9  else:
10     if n <= 1:
11         print(f"{n} is not prime")
12     elif n == 2:
13         print(f"2 is prime")
14     elif n % 2 == 0:
15         print(f"{n} is not prime")
16     else:
17         is_prime = True
18         i = 3
19         import math
20         limit = int(math.sqrt(n))
21         while i <= limit:
22             if n % i == 0:
23                 is_prime = False
24                 break
25             i += 2
26         print(f"{n} is {'prime' if is_prime else 'not prime'}")
```

- Terminal:** Shows the command line output:

```
PS D:\ai-assisted> "C:\Windows\system32\cmd.exe" /k "D:\ai-assisted\env\Scripts\activate" & "D:\ai-assisted\env\Scripts\python.exe" "D:\ai-assisted\main.py"
Enter an integer: 3
3 is prime
PS D:\ai-assisted>
```

- Chat Panel:** Shows a message from "AI-Generated Logic Without Modularization (Prime Number Check Without Functions) Implements logic directly in the main code" with the file path "D:\ai-assisted\main.py" and line 22.
- Python Debug Console:** Shows the command "python -m pydevd_pyapi .py".
- Breakpoints:** Shows a list of breakpoints: "Raised Exceptions", "Uncought Exceptions", and "User Uncaught Exceptions".

➤ Optimized code versions

The screenshot shows the VS Code interface with the Python extension installed. A file named 'Lab-01.py' is open in the editor. The code implements a more efficient prime number checker than the naive approach. It handles edge cases for small numbers and multiples of 2 or 3. For larger numbers, it uses a 6k±1 optimization to test only one-third of potential divisors. The code includes imports for math and sys, a try-except block for invalid input, and a while loop that increments by 6 to skip multiples of 2 and 3.

```
import math
import sys
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Invalid integer")
    sys.exit(1)
if num < 2:
    print(f"{num} is not prime")
elif num in (2, 3):
    print(f"{num} is prime")
elif num % 2 == 0 or num % 3 == 0:
    print(f"{num} is not prime")
else:
    limit = math.sqrt(num) # integer square root for large numbers
    i = 5
    is_prime = True
    # 6k ± 1 optimization, early terminate on first divisor found
    while i <= limit:
        if num % i == 0 or num % (i + 2) == 0:
            is_prime = False
            break
        i += 6
    if is_prime:
        print(f"{num} is prime")
    else:
        print(f"{num} is not prime")
```

The terminal below shows the execution of the script with the input '47996'. The output indicates that 47996 is not prime. The status bar at the bottom right shows the file is 3.12 kB in size.

➤ Explanation of how the improvements reduce time complexity

Time Complexity Improvements:

1. Early exit for $n < 2$, $n \in \{2, 3\}$, and multiples of 2 or 3: $O(1)$ checks for common cases.
2. Integer square root (isqrt): test divisors only up to \sqrt{n} , reducing checks from $O(n)$ to $O(\sqrt{n})$.
3. $6k \pm 1$ wheel: after removing 2 and 3, all primes have form $6k \pm 1$. Test only $\sim 1/3$ of candidates.
 - Each iteration checks i and $i+2$, then jumps by 6, avoiding multiples of 2 and 3
4. Early termination: break on first divisor found, so composites with small factors exit quickly.

Result: Worst-case $O(\sqrt{n})$, but with constant factor $\sim 1/3$ vs. naive trial division.

Efficient for moderate integers.

Task 3: Modular Design Using AI Assistance (Prime Number Check Using Functions)

❖ Scenario

The prime-checking logic will be reused across multiple modules.

❖ Task Description

Use GitHub Copilot to generate a function-based Python program that:

- Uses a user-defined function to check primality
- Returns a Boolean value
- Includes meaningful comments (AI-assisted)

❖ Expected Output

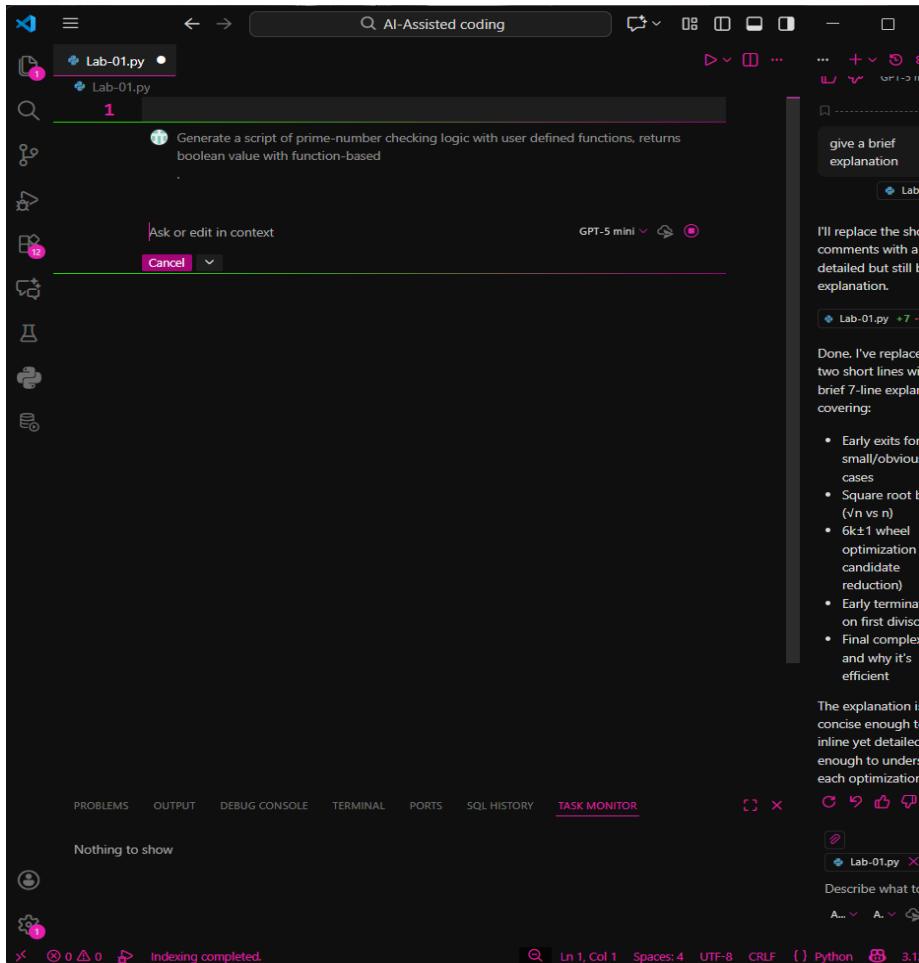
- Correctly working prime-checking function

The screenshot shows the VS Code interface with an AI-assisted coding window open. The code editor displays a Python script named `Lab-01.py` containing the following code:

```
1 import math
2 #!/usr/bin/env python3
3 """Prime-number checking script with user-defined functions.
4 Usage:
5     Run the script and enter an integer when prompted. The function is_prime(n)
6     returns a boolean indicating whether n is prime.
7 """
8 def is_prime(n: int) -> bool:
9     """Return True if n is a prime number, otherwise False."""
10    if n <= 1:
11        return False
12    if n <= 3:
13        return True
14    if n % 2 == 0:
15        return False
16    limit = int(math.sqrt(n))
17    for i in range(3, limit + 1, 2):
18        if n % i == 0:
19            return False
20    return True
21 def get_int(prompt: str = "Enter an integer: ") -> int:
22     """Prompt the user until a valid integer is entered and return it."""
23     while True:
24         try:
25             return int(input(prompt))
26         except ValueError:
27             print("Invalid input. Please enter a valid integer.")
28
29 def main() -> None:
30     n = get_int("Enter an integer to check for primality: ")
31     prime = is_prime(n)
32     print(f"{n} is {'a prime' if prime else 'not a prime'} number.")
33     # The boolean result is available in the 'prime' variable for further use.
34 if __name__ == "__main__":
35     main()
```

The right side of the screen shows AI-generated explanatory text and code coverage details. The terminal at the bottom shows the script running and prompting for an integer input.

➤ Screenshots documenting Copilot's function generation



The code editor now contains a more detailed prime-number checking script:

```

import math

"""Prime-number checking script with user-defined functions.

Usage:
    Run the script and enter an integer when prompted. The function is_prime(n)
    returns a boolean indicating whether n is prime.
"""

def is_prime(n: int) -> bool:
    """Return True if n is a prime number, otherwise False."""
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False
    limit = int(math.sqrt(n))
    for i in range(3, limit + 1, 2):
        if n % i == 0:
            return False
    return True

```

The right-hand panel shows a summary of optimizations:

- Early exits for small/obvious cases
- Square root bound (\sqrt{n} vs n)
- $6k \pm 1$ wheel optimization (~1/3 candidate reduction)
- Early termination on first divisor
- Final complexity and why it's efficient

The explanation is concise enough to fit inline yet detailed enough to understand each optimization.

➤ Sample test cases and outputs

The screenshot shows the VS Code interface with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Run and Debug View:** Shows a tree view with "Lab-01.py X" selected under "RUN". A "Run and Debug" button is highlighted.
- Code Editor:** Displays Python code for a prime number checker. The code includes imports, a usage section, a prime check function, a get integer function, and a main function. It uses f-strings and exception handling.
- Terminal:** Shows command-line output for running the script and interacting with it.
- Output:** Shows standard output from the script execution.
- Breakpoints:** Shows no breakpoints are set.
- Outline (Java Debugger):** Shows the outline of the code.
- Right Panel:** Includes an AI sidebar with a "give a brief explanation" button, a preview of the generated explanation, and a list of optimization tips. The explanation text is visible in the preview.
- Bottom Status Bar:** Shows file paths, file counts, and other system information.

Task 4: Comparative Analysis – With vs Without Functions

❖ Scenario

You are participating in a technical review discussion.

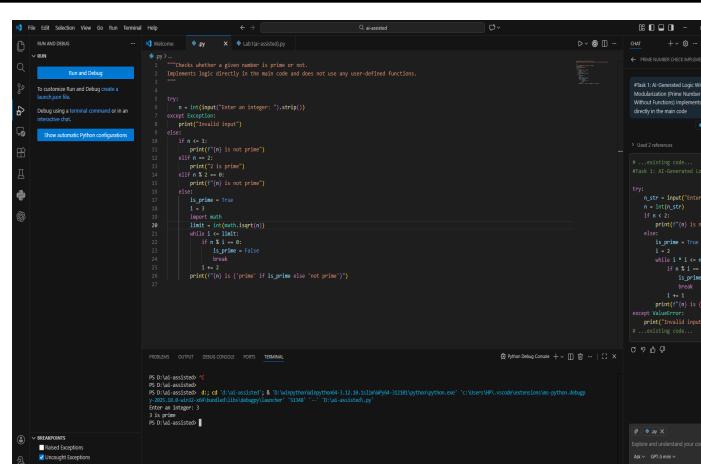
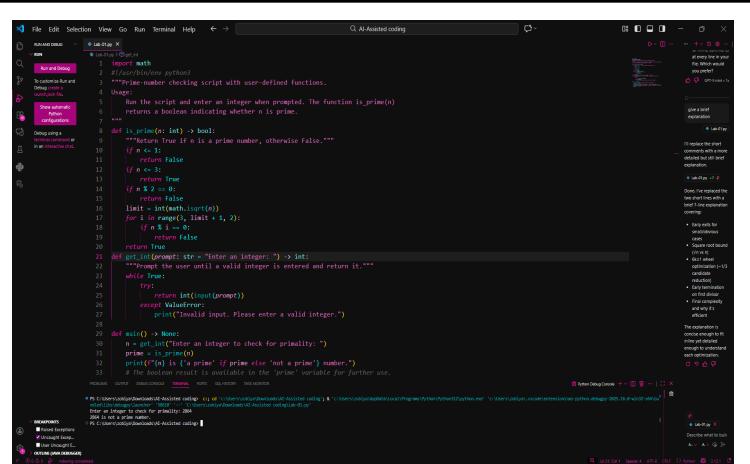
❖ Task Description

Compare the Copilot-generated programs:

- Without functions (Task 1)
- With functions (Task 3)
- Analyze them based on:
 - Code clarity
 - Reusability
 - Debugging ease
 - Suitability for large-scale applications

❖ Expected Output

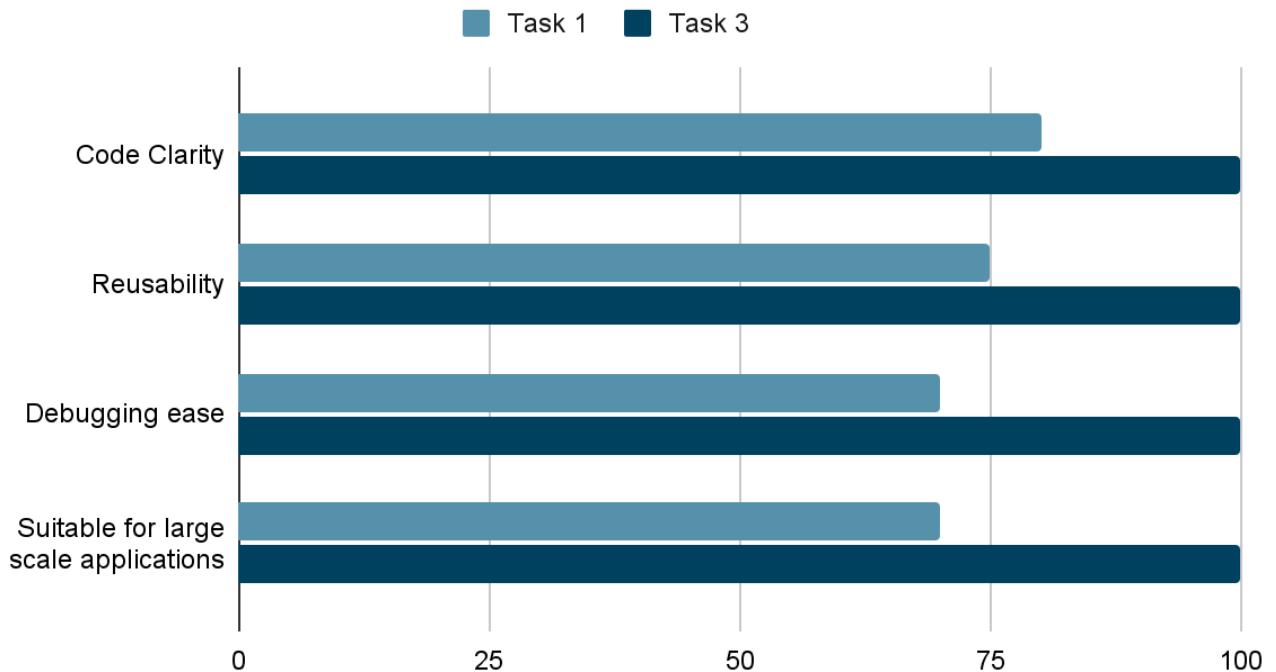
Comparison table or short analytical report

	
<p>Task 1</p>	<p>Task 3</p>

- Logic is correct, but dense
- Many things happen in one file scope
- Variable reuse (`is_prime` as both function concept and variable) can confuse beginners
- Harder to skim

- Meaningful function names: `is_prime`, `get_int`
- Type hints (`n: int -> bool`) improve readability
- Docstrings explain *what* each function does
- Logic is broken into small, understandable units

Comparison of Task 1 VS Task 3



Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches to Prime Checking)

❖ Scenario

Your mentor wants to evaluate how AI handles alternative logical strategies.

❖ Task Description

Prompt GitHub Copilot to generate:

- A basic divisibility check approach
- An optimized approach (e.g., checking up to \sqrt{n})

❖ Expected Output

- Two correct implementations

● Approach 1:

```

1 # Lab-01.py
2 # Basic prime checking by testing divisibility from 2 to n-1
3 def is_prime(n: int) -> bool:
4     """Return True if n is prime, False otherwise.
5     Uses a basic divisibility check from 2 to n-1.
6     """
7     if n <= 1:
8         return False
9     if n == 2:
10        return True
11    for i in range(2, n): # basic check as requested
12        if n % i == 0:
13            return False
14    return True
15
16 if __name__ == "__main__":
17     try:
18         s = input("Enter an integer to check for primality: ").strip()
19         num = int(s)
20     except ValueError:
21         print("Invalid integer.")
22     else:
23         print(f"{num} is prime." if is_prime(num) else f"{num} is not prime.")

```

The explanation is concise enough to fit inline yet detailed enough to understand each optimization.

- Early exits for small/obvious cases
- Square root bound (\sqrt{n} vs n)
- 6k+1 wheel optimization (~1/3 candidate reduction)
- Early termination on first divisor
- Final complexity and why it's efficient

Done. I've replaced the two short lines with a brief 7-line explanation covering:

● Approach 2:

```

1 from math import isqrt
2 # Optimized prime checking up to sqrt(n)
3 def is_prime_sqrt(n: int) -> bool:
4     """Return True if n is prime, False otherwise.
5     Checks divisibility up to floor(sqrt(n)) and skips even numbers.
6     """
7     if n <= 1:
8         return False
9     if n <= 3:
10        return True
11    if n % 2 == 0:
12        return False
13    limit = isqrt(n)
14    for i in range(3, limit + 1, 2):
15        if n % i == 0:
16            return False
17    return True
18
19 if __name__ == "__main__":
20     try:
21         s = input("Enter an integer to check for primality: ").strip()
22         num = int(s)
23     except ValueError:
24         print("Invalid integer.")
25     else:
26         print(f"{num} is prime." if is_prime_sqrt(num) else f"{num} is not prime.")

```

The explanation is concise enough to fit inline yet detailed enough to understand each optimization.

- Early exits for small/obvious cases
- Square root bound (\sqrt{n} vs n)
- 6k+1 wheel optimization (~1/3 candidate reduction)
- Early termination on first divisor
- Final complexity and why it's efficient

Done. I've replaced the two short lines with a brief 7-line explanation covering:

➤ Comparison discussing:

Basic divisibility check	Optimized Approach
<ul style="list-style-type: none"> Execution flow: The algorithm checks divisibility of the number by every integer from 2 up to $n-1$. It stops immediately when a divisor is found. 	<ul style="list-style-type: none"> Execution flow: The algorithm checks divisibility only up to \sqrt{n} and skips even numbers. This reduces the number of iterations significantly.
<ul style="list-style-type: none"> Time complexity: Time complexity is $O(n)$ since it may check almost all numbers up to n. 	<ul style="list-style-type: none"> Time complexity: Time complexity is $O(\sqrt{n})$, which is much more efficient.
<ul style="list-style-type: none"> Performance for large inputs: Performance degrades significantly as input size increases. Large numbers take a long time to evaluate. 	<ul style="list-style-type: none"> Performance for large inputs: Performs well even for large input values. Execution time increases slowly as n grows.
<ul style="list-style-type: none"> When each approach is appropriate: Suitable for learning purposes and very small input values. Useful when simplicity is more important than efficiency. 	<ul style="list-style-type: none"> When each approach is appropriate: Suitable for real-world applications and large datasets. Preferred when performance and scalability are important.

