


SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING																		
Program Name: B. Tech		Assignment Type: Lab	Academic Year: 2025-2026																	
Course Coordinator Name		Dr. Rishabh Mittal																		
Instructor(s) Name		<table border="1"> <tr><td>Mr. S Naresh Kumar</td></tr> <tr><td>Ms. B. Swathi</td></tr> <tr><td>Dr. Sasanko Shekhar Gantayat</td></tr> <tr><td>Mr. Md Sallauddin</td></tr> <tr><td>Dr. Mathivanan</td></tr> <tr><td>Mr. Y Srikanth</td></tr> <tr><td>Ms. N Shilpa</td></tr> <tr><td>Dr. Rishabh Mittal (Coordinator)</td></tr> <tr><td>Dr. R. Prashant Kumar</td></tr> <tr><td>Mr. Ankushavali MD</td></tr> <tr><td>Mr. B Viswanath</td></tr> <tr><td>Ms. Sujitha Reddy</td></tr> <tr><td>Ms. A. Anitha</td></tr> <tr><td>Ms. M. Madhuri</td></tr> <tr><td>Ms. Katherashala Swetha</td></tr> <tr><td>Ms. Velpula sumalatha</td></tr> <tr><td>Mr. Bingi Raju</td></tr> </table>		Mr. S Naresh Kumar	Ms. B. Swathi	Dr. Sasanko Shekhar Gantayat	Mr. Md Sallauddin	Dr. Mathivanan	Mr. Y Srikanth	Ms. N Shilpa	Dr. Rishabh Mittal (Coordinator)	Dr. R. Prashant Kumar	Mr. Ankushavali MD	Mr. B Viswanath	Ms. Sujitha Reddy	Ms. A. Anitha	Ms. M. Madhuri	Ms. Katherashala Swetha	Ms. Velpula sumalatha	Mr. Bingi Raju
Mr. S Naresh Kumar																				
Ms. B. Swathi																				
Dr. Sasanko Shekhar Gantayat																				
Mr. Md Sallauddin																				
Dr. Mathivanan																				
Mr. Y Srikanth																				
Ms. N Shilpa																				
Dr. Rishabh Mittal (Coordinator)																				
Dr. R. Prashant Kumar																				
Mr. Ankushavali MD																				
Mr. B Viswanath																				
Ms. Sujitha Reddy																				
Ms. A. Anitha																				
Ms. M. Madhuri																				
Ms. Katherashala Swetha																				
Ms. Velpula sumalatha																				
Mr. Bingi Raju																				
CourseCode	23CS002P C304	Course Title	AI Assisted Coding																	
Year/Sem	III/II	Regulation	R23																	
Date and Day of Assignment	Week1 – Wednesday	Time(s)	23CSBTB01 To 23CSBTB52																	
Duration	2 Hours	Applicable to Batches	All batches																	
Assignment Number: 1.3 (Present assignment number) / 24 (Total number of assignments)																				

Q · N o ·	Question	Exp ecte d Tim e to com

		plet e
1	<p>Lab 2: Exploring Additional AI Coding Tools beyond Copilot – Gemini (Colab) and Cursor AI</p> <p>Lab Objectives:</p> <ul style="list-style-type: none"> ❖ To explore and evaluate the functionality of Google Gemini for AI-assisted coding within Google Colab. ❖ To understand and use Cursor AI for code generation, explanation, and refactoring. ❖ To compare outputs and usability between Gemini, GitHub Copilot, and Cursor AI. ❖ To perform code optimization and documentation using AI tools. <p>Lab Outcomes (LOs): After completing this lab, students will be able to:</p> <ul style="list-style-type: none"> ❖ Generate Python code using Google Gemini in Google Colab. ❖ Analyze the effectiveness of code explanations and suggestions by Gemini. ❖ Set up and use Cursor AI for AI-powered coding assistance. ❖ Evaluate and refactor code using Cursor AI features. ❖ Compare AI tool behavior and code quality across different platforms. 	Wee k1 - Mon day
	<p>Task 1: Word Frequency from Text File</p> <ul style="list-style-type: none"> ❖ Scenario: You are analyzing log files for keyword frequency. ❖ Task: Use Gemini to generate Python code that reads a text file and counts word frequency, then explains the code. ❖ Expected Output: <ul style="list-style-type: none"> ➤ Working code ➤ Explanation ➤ Screenshot <p>Prompt Generate Python code to read a text file and count word frequency. Also explain the code step by step in simple terms.</p>	

Gemini



Generate Python code to read a text file and count word frequency. Also explain the code step by step in simple terms.

Working...

```

import re
from collections import Counter

def count_word_frequency(filename):
    """
    Reads a text file, counts the frequency of each word,
    and returns a Counter object.
    """
    all_words = []
    with open(filename, 'r', encoding='utf-8') as file:
        text = file.read()
        # Convert to lowercase and remove punctuation
        cleaned_text = re.sub(r'[^a-zA-Z ]+', '', text).lower()
        # Split the text into words
        words = cleaned_text.split()
        all_words.extend(words)
    except FileNotFoundError:
        print(f"Error: The file '{filename}' was not found.")
        return Counter()
    except Exception as e:
        print(f"An error occurred: {e}")
        return Counter()

# Count word frequencies
word_counts = Counter(all_words)
return word_counts

# Example usage:
# Specify the path to your text file
file_path = 'sample.txt'

# Call the function to count word frequencies
frequencies = count_word_frequency(file_path)

# Display the top 10 most common words
print(f"Word frequencies in '{file_path}':")
for word, count in frequencies.most_common(10):
    print(f"{word}: {count}")
else:
    print("No words found or an error occurred.")

# Word frequencies in 'sample.txt':
# sample: 3
# text: 3
# this: 2
# is: 2
# contains: 1
# to: 1
# demonstrate: 1

```

Generate Python code to read a text file and count word frequency. Also explain the code step by step in simple terms.

Clearly I'll generate the Python code to read a text file and count word frequencies, along with a step-by-step explanation. The code has successfully processed the file 'sample.txt' and created with the sample content. Then, the word frequency counting script ran, and here are the top 10 most common words and their counts from 'sample.txt':

- sample: 3
- text: 3
- this: 2
- is: 2
- contains: 1
- to: 1
- to: 1
- demonstrate: 1

Is there anything else you'd like to do with this file or analysis?

We have successfully generated Python code to:

- Create a sample text file ('sample.txt').
- Read the content of the file.
- Process the text by cleaning it (converting to lowercase, removing punctuation).
- Count the frequency of each word using 'collections.Counter'.
- Display the top 10 most common words and their counts.

All steps were explained in detail. Is there anything else you would like to do with this word frequency analysis, or perhaps a different task?

What can I help you build?

Gemini 1.5 Flash

Description

In this task, Google Gemini is used within Google Colab to generate Python code that reads a text file and calculates the frequency of each word. This task simulates real-world log file analysis where keyword frequency is important. Gemini not only generates correct Python code but also provides a clear explanation of each step, helping users understand how file handling and text processing work in Python.

❖ **Scenario:**
You are automating basic file operations.

❖ **Task:**
Use Cursor AI to generate a program that:

- Creates a text file
- Writes sample text
- Reads and displays the content

❖ **Expected Output:**

- Functional code
- Cursor AI screenshots

Prompt

Write a Python program that:

1. Creates a text file
2. Writes sample text into it
3. Reads the file
4. Displays the content

The screenshot shows the Cursor AI IDE interface. The main editor displays a Python script named `itay.py` with the following code:

```
1 # Python program to create, write, read, and display a text file
2
3 # Step 1: Create and write to a text file
4 filename = "sample.txt"
5 sample_text = """Hello, this is a sample text file!
6 This program demonstrates file operations in Python.
7 It creates a file, writes content to it, reads it back, and displays it.
8 Python file handling is simple and powerful."""
9
10 print("Step 1: Creating file and writing content...")
11 with open(filename, "w") as file:
12     file.write(sample_text)
13
14 print(f"/ File '{filename}' created and content written successfully!\n")
15
16 # Step 2: Read the file
17 print("Step 2: Reading the file...")
18 with open(filename, "r") as file:
19     content = file.read()
20
21 print(f"/ File read successfully!\n")
22
23 # Step 3: Display the content
24 print("Step 3: Displaying file content:")
25 print("-" * 50)
26 print(content)
27 print("\nProgram completed successfully!")
```

The right sidebar shows the "View Plans" panel with a plan titled "Write a Python program that: 1. Creates a text file 2. Writes sample text into it". The plan includes a description of the program's purpose and a list of steps: 1. Create a text file, 2. Write sample text, 3. Read the file, 4. Display the content. The plan also includes a code snippet for the program.

Explanation

- File opened in write mode
- Sample text written
- File reopened in read mode
- Content printed

Description

This task demonstrates the use of Cursor AI to automate basic file operations in Python. Cursor AI is used to generate a program that creates a text file, writes sample content into it, reads the stored data, and displays the output. This task highlights how Cursor AI assists in quickly generating functional code and is useful for understanding file handling concepts in Python.

Task 3: CSV Data Analysis

- ❖ **Scenario:**
You are processing structured data from a CSV file.
- ❖ **Task:**
Use Gemini in Colab to read a CSV file and calculate mean, min, and max.
- ❖ **Expected Output:**
 - Correct output
 - Screenshot

Prompt

Write Python code to read a CSV file and calculate mean, minimum, and maximum values.

The screenshot displays a Jupyter Notebook interface with a dark theme. The left sidebar shows a file explorer with 'sample_data' and 'sample.txt'. The main area contains a Python script that reads a CSV file, calculates mean, min, and max for numerical columns, and prints the results. The output shows the first 5 rows of the dataset and the calculated statistics.

```
# Import necessary libraries
import pandas as pd

# Define the path to your CSV file
csv_file_path = "sample_data.csv"

# Read the CSV file into a pandas DataFrame
df = pd.read_csv(csv_file_path)

# Display the first few rows of the DataFrame to understand its structure
print("Here's a view of the dataset:")
display(df.head())

# Select only numerical columns for calculations
numerical_df = df.select_dtypes(include="number")

# Calculate mean, min, and max for numerical columns
mean_values = numerical_df.mean()
min_values = numerical_df.min()
max_values = numerical_df.max()

# Print the results
print("Calculated Statistics (Mean, Min, Max):")
print(mean_values)
print(min_values)
print(max_values)

# Print the numerical columns found in the CSV to calculate statistics.
print("Numerical columns found in the CSV to calculate statistics:")
print(numerical_df.columns)
```

Successfully loaded data from 'sample_data.csv'.

First 5 rows of the dataset:

Duration	Pulse	Resective	Calories
0	100	120	400.1
1	100	120	470.0
2	100	120	500.0
3	100	120	500.0
4	100	120	500.0

Calculated Statistics (Mean, Min, Max):

Duration	Pulse	Resective	Calories
Mean	100.0	120.0	470.0
Min	100.0	120.0	400.1
Max	100.0	120.0	500.0

```
Duration    63.846154
Pulse       107.461538
Maxpulse    134.047337
Calories    375.790244

dtype: float64

Minimum values:
0
Duration    15.0
Pulse       80.0
Maxpulse    100.0
Calories    50.3

dtype: float64

Maximum values:
0
Duration    300.0
Pulse       159.0
Maxpulse    184.0
Calories    1860.4

dtype: float64
```

Code Explanation:

1. `import pandas as pd` : Imports the pandas library, which is essential for working with DataFrames (table-like data structures).
2. `csv_file_path = '/content/data.csv'` : Defines the path to your CSV file. Make sure this path is correct.
3. `try...except` block: This block is used for robust error handling:
 - `df = pd.read_csv(csv_file_path)` : Attempts to read the CSV file into a pandas DataFrame named `df`.
 - `display(df.head())` : Shows the first 5 rows of the loaded DataFrame. This helps verify that the data was loaded correctly and gives an idea of its structure.
 - `numerical_df = df.select_dtypes(include=['number'])` : Filters the DataFrame to include only columns with numerical data types (integers and floats). Statistical calculations only make sense for numerical columns.
 - `mean_values = numerical_df.mean()` : Calculates the mean (average) for each numerical column.
 - `min_values = numerical_df.min()` : Calculates the minimum value for each numerical column.
 - `max_values = numerical_df.max()` : Calculates the maximum value for each numerical column.
 - `display(...)` : Prints the calculated mean, minimum, and maximum values for each respective numerical column.
 - **Error Handling:** Catches `FileNotFoundError` if the CSV doesn't exist, `EmptyDataError` if the file is empty, and a general `Exception` for any other issues, providing informative messages.

Explanation

- Pandas used for CSV handling
- `mean()`, `min()`, `max()` functions applied
- Displays statistical values

Description

In this task, Google Gemini is used in Google Colab to process structured data from a CSV file. The generated Python code reads the CSV file and calculates statistical values such as mean, minimum, and maximum. This task demonstrates how AI tools simplify data analysis tasks and help users apply Python libraries like Pandas for real-world data processing.

Task 4: Sorting Lists – Manual vs Built-in

❖ **Scenario:**

You are reviewing algorithm choices for efficiency.

❖ **Task:**

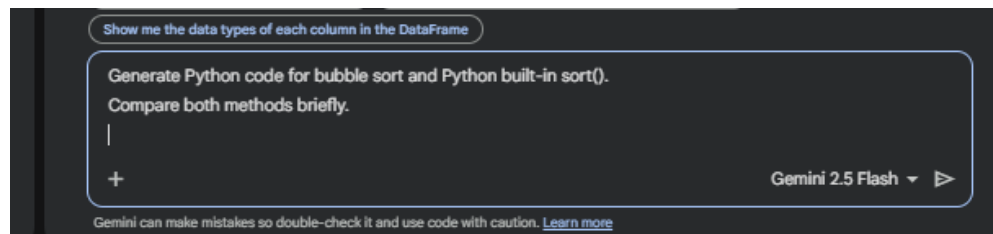
Use **Gemini** to generate:

- Bubble sort
- Python's built-in sort()
- Compare both implementations.

❖ **Expected Output:**

- Two versions of code
- Short comparison

Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.



Prompt

**Generate Python code for bubble sort and Python built-in sort().
Compare both methods briefly.**

	<div><div>First, let's implement the Bubble Sort algorithm. It's a simple comparison-based sorting algorithm.</div><div><pre>def bubble_sort(arr): n = len(arr) # Traverse through all array elements for i in range(n - 1): # Last i elements are already in place for j in range(0, n - i - 1): # Traverse the array from 0 to n-i-1 # Swap if the element found is greater # than the next element if arr[j] > arr[j + 1]: arr[j], arr[j + 1] = arr[j + 1], arr[j] return arr # Test Bubble Sort my_list = [64, 34, 25, 12, 22, 11, 90] print(f"Original list: {my_list}") sorted_list_bubble = bubble_sort(list(my_list)) # Use a copy to avoid modifying original print(f"Sorted using Bubble Sort: {sorted_list_bubble}")</pre></div><div>Next, let's use Python's built-in <code>sort()</code> method (for lists) or <code>sorted()</code> function (for any iterable). These are highly optimized and generally preferred in practice.</div><div><pre># Test Python's built-in sort() my_list_2 = [64, 34, 25, 12, 22, 11, 90] print(f"Original list: {my_list_2}") my_list_2.sort() # Sorts the list in-place print(f"Sorted using built-in sort() method: {my_list_2}") # Test Python's built-in sorted() function my_tuple = (64, 34, 25, 12, 22, 11, 90) print(f"Original tuple: {my_tuple}") sorted_tuple = sorted(my_tuple) # Returns a new sorted list from an iterable print(f"Sorted using built-in sorted() function: {sorted_tuple}")</pre></div><div><div><div>◆ Gemini</div><div>Comparison of Bubble Sort vs. Python's Built-in Sort</div><table><thead><tr><th>Feature</th><th>Bubble Sort</th><th>Python's Built-in <code>sort()</code> / <code>sorted()</code></th></tr></thead><tbody><tr><td>Algorithm</td><td>Simple comparison-based, repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.</td><td>Timsort (a hybrid stable sorting algorithm, derived from merge sort and insertion sort). Highly optimized.</td></tr><tr><td>Time Complexity</td><td>Worst/Average: $O(n^2)$ (very inefficient for large datasets). Best: $O(n)$ (if already sorted).</td><td>Worst/Average/Best: $O(n \log n)$ (very efficient for large datasets).</td></tr><tr><td>Space Complexity</td><td>$O(1)$ (in-place sort).</td><td>$O(n)$ in worst case (for Timsort, due to temporary storage).</td></tr><tr><td>Stability</td><td>Stable (maintains the relative order of equal elements).</td><td>Stable.</td></tr><tr><td>Ease of Use</td><td>Requires manual implementation.</td><td>Very easy to use; built directly into Python.</td></tr><tr><td>When to Use</td><td>Primarily for educational purposes to understand basic sorting concepts. Not recommended for production.</td><td>Almost always preferred for general-purpose sorting in Python due to its efficiency and reliability.</td></tr></tbody></table><p>In summary: While Bubble Sort is easy to understand, Python's built-in <code>sort()</code> method (and <code>sorted()</code> function) are significantly more efficient, robust, and should be used for practical applications.</p></div><div><div>◆ Sure!</div><div>I'll generate the Python code for both bubble sort and the built-in <code>sort()</code> function, and then provide a brief comparison of these two sorting methods.</div><div>Great! Both the Bubble Sort implementation and the demonstrations of Python's built-in <code>sort()</code> method and <code>sorted()</code> function have executed successfully.</div><div>Bubble Sort:</div><ul style="list-style-type: none">• Original list: [64, 34, 25, 12, 22, 11, 90]• Sorted using Bubble Sort: [11, 12, 22, 25, 34, 64, 90]<div>Python's Built-in Sort:</div><ul style="list-style-type: none">• Original list (<code>my_list_2</code>): [64, 34, 25, 12, 22, 11, 90]• Sorted using <code>sort()</code> method: [11, 12, 22, 25, 34, 64, 90]• Original tuple (<code>my_tuple</code>): (64, 34, 25, 12, 22, 11, 90)• Sorted using <code>sorted()</code> function: [11, 12, 22, 25, 34, 64, 90]<p>As you can see, both methods correctly sorted the provided lists/tuples. Now, let's briefly compare them:</p><div><div>Comparison of Bubble Sort vs. Python's Built-in Sort</div><table><thead><tr><th>Feature</th><th>Bubble Sort</th><th>Python's Built-in <code>sort()</code> / <code>sorted()</code></th></tr></thead><tbody><tr><td>Algorithm</td><td>Simple comparison-based, repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.</td><td>Timsort (a hybrid stable sorting algorithm, derived from merge sort and insertion sort). Highly optimized.</td></tr><tr><td>Time Complexity</td><td>Worst/Average: $O(n^2)$ (very inefficient for large datasets). Best: $O(n)$ (if already sorted).</td><td>Worst/Average/Best: $O(n \log n)$ (very efficient for large datasets).</td></tr><tr><td>Space Complexity</td><td>$O(1)$ (in-place sort).</td><td>$O(n)$ in worst case (for Timsort, due to temporary storage).</td></tr><tr><td>Stability</td><td>Stable (maintains the relative order of equal elements).</td><td>Stable.</td></tr><tr><td>Ease of Use</td><td>Requires manual implementation.</td><td>Very easy to use; built directly into Python.</td></tr><tr><td>When to Use</td><td>Primarily for educational purposes to understand basic sorting concepts. Not recommended for production.</td><td>Almost always preferred for general-purpose sorting in Python due to its efficiency and reliability.</td></tr></tbody></table><p>In summary: While Bubble Sort is easy to understand, Python's built-in <code>sort()</code> method (and <code>sorted()</code> function) are significantly more efficient, robust, and should be used for practical applications.</p><p>We've successfully generated Python code for both the Bubble Sort algorithm and Python's highly optimized built-in <code>sort()</code> method and <code>sorted()</code> function. We also executed them to demonstrate their functionality, and provided a detailed comparison highlighting their differences in terms of algorithm, time/space complexity, stability, and ease of use.</p><p>Is there anything else I can assist you with regarding sorting, or perhaps a different programming task?</p><div><div>👍</div><div>🗨</div></div></div></div></div></div>	Feature	Bubble Sort	Python's Built-in <code>sort()</code> / <code>sorted()</code>	Algorithm	Simple comparison-based, repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.	Timsort (a hybrid stable sorting algorithm, derived from merge sort and insertion sort). Highly optimized.	Time Complexity	Worst/Average: $O(n^2)$ (very inefficient for large datasets). Best: $O(n)$ (if already sorted).	Worst/Average/Best: $O(n \log n)$ (very efficient for large datasets).	Space Complexity	$O(1)$ (in-place sort).	$O(n)$ in worst case (for Timsort, due to temporary storage).	Stability	Stable (maintains the relative order of equal elements).	Stable.	Ease of Use	Requires manual implementation.	Very easy to use; built directly into Python.	When to Use	Primarily for educational purposes to understand basic sorting concepts. Not recommended for production.	Almost always preferred for general-purpose sorting in Python due to its efficiency and reliability.	Feature	Bubble Sort	Python's Built-in <code>sort()</code> / <code>sorted()</code>	Algorithm	Simple comparison-based, repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.	Timsort (a hybrid stable sorting algorithm, derived from merge sort and insertion sort). Highly optimized.	Time Complexity	Worst/Average: $O(n^2)$ (very inefficient for large datasets). Best: $O(n)$ (if already sorted).	Worst/Average/Best: $O(n \log n)$ (very efficient for large datasets).	Space Complexity	$O(1)$ (in-place sort).	$O(n)$ in worst case (for Timsort, due to temporary storage).	Stability	Stable (maintains the relative order of equal elements).	Stable.	Ease of Use	Requires manual implementation.	Very easy to use; built directly into Python.	When to Use	Primarily for educational purposes to understand basic sorting concepts. Not recommended for production.	Almost always preferred for general-purpose sorting in Python due to its efficiency and reliability.	
Feature	Bubble Sort	Python's Built-in <code>sort()</code> / <code>sorted()</code>																																										
Algorithm	Simple comparison-based, repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.	Timsort (a hybrid stable sorting algorithm, derived from merge sort and insertion sort). Highly optimized.																																										
Time Complexity	Worst/Average: $O(n^2)$ (very inefficient for large datasets). Best: $O(n)$ (if already sorted).	Worst/Average/Best: $O(n \log n)$ (very efficient for large datasets).																																										
Space Complexity	$O(1)$ (in-place sort).	$O(n)$ in worst case (for Timsort, due to temporary storage).																																										
Stability	Stable (maintains the relative order of equal elements).	Stable.																																										
Ease of Use	Requires manual implementation.	Very easy to use; built directly into Python.																																										
When to Use	Primarily for educational purposes to understand basic sorting concepts. Not recommended for production.	Almost always preferred for general-purpose sorting in Python due to its efficiency and reliability.																																										
Feature	Bubble Sort	Python's Built-in <code>sort()</code> / <code>sorted()</code>																																										
Algorithm	Simple comparison-based, repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.	Timsort (a hybrid stable sorting algorithm, derived from merge sort and insertion sort). Highly optimized.																																										
Time Complexity	Worst/Average: $O(n^2)$ (very inefficient for large datasets). Best: $O(n)$ (if already sorted).	Worst/Average/Best: $O(n \log n)$ (very efficient for large datasets).																																										
Space Complexity	$O(1)$ (in-place sort).	$O(n)$ in worst case (for Timsort, due to temporary storage).																																										
Stability	Stable (maintains the relative order of equal elements).	Stable.																																										
Ease of Use	Requires manual implementation.	Very easy to use; built directly into Python.																																										
When to Use	Primarily for educational purposes to understand basic sorting concepts. Not recommended for production.	Almost always preferred for general-purpose sorting in Python due to its efficiency and reliability.																																										
	<div><div>Final Comparison Section</div><table><thead><tr><th>Tool</th><th>Strengths</th><th>Weakness</th></tr></thead><tbody><tr><td>Gemini</td><td>Good explanations, works well in Colab</td><td>Needs internet</td></tr></tbody></table></div>	Tool	Strengths	Weakness	Gemini	Good explanations, works well in Colab	Needs internet																																					
Tool	Strengths	Weakness																																										
Gemini	Good explanations, works well in Colab	Needs internet																																										

	Cursor AI GitHub Copilot	Fast code generation & refactoring Inline suggestions	Editor based Limited explanation	
<p>Description</p> <p>This task compares two sorting approaches generated by Google Gemini: a manual sorting algorithm (Bubble Sort) and Python’s built-in sort() method. The purpose of this task is to analyze the efficiency and readability of different sorting techniques. Gemini helps in generating both implementations and provides a brief comparison, showing why built-in methods are more efficient for practical applications.</p>				