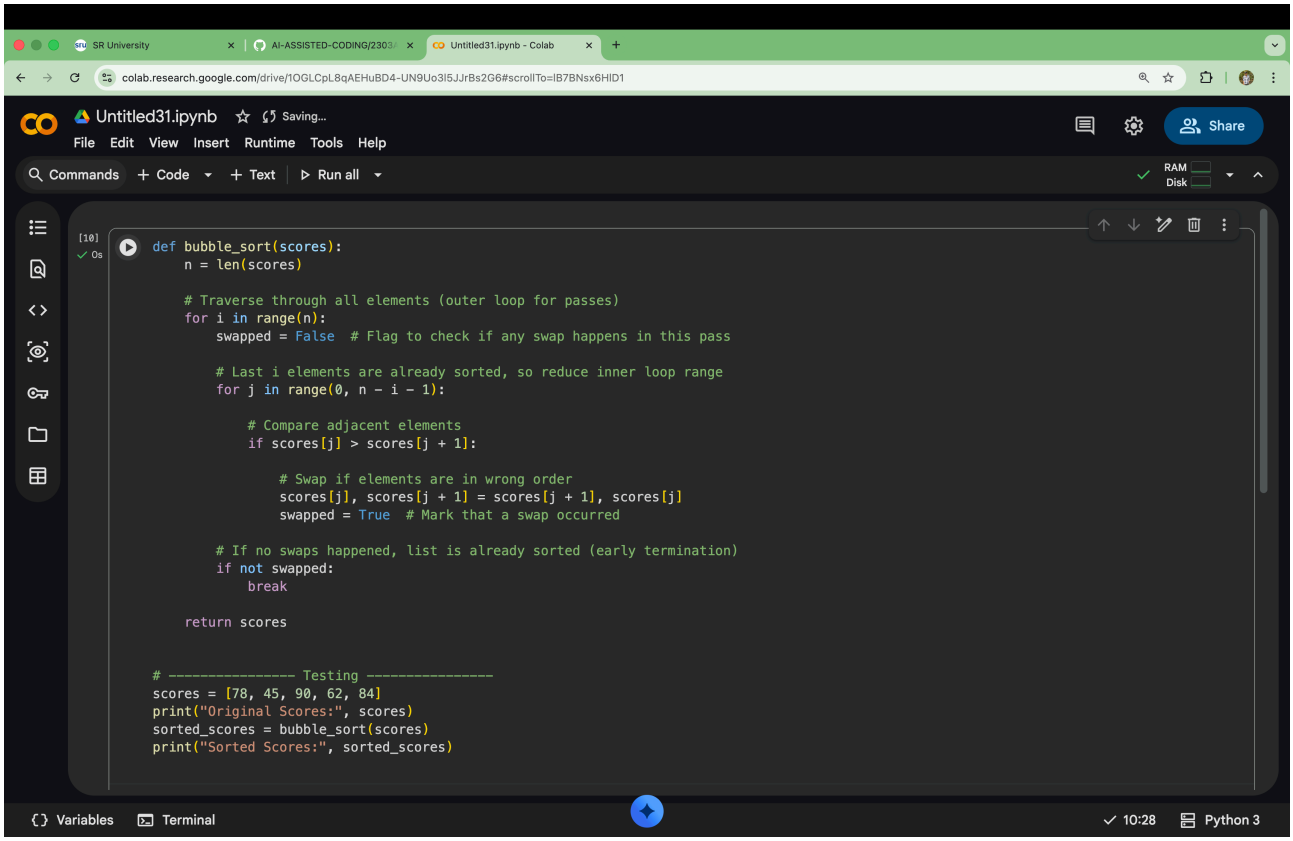
ASSIGNMENT-12

NAME:B.MANISHWAR
HALL TICKET:2303A51276
BATCH:05

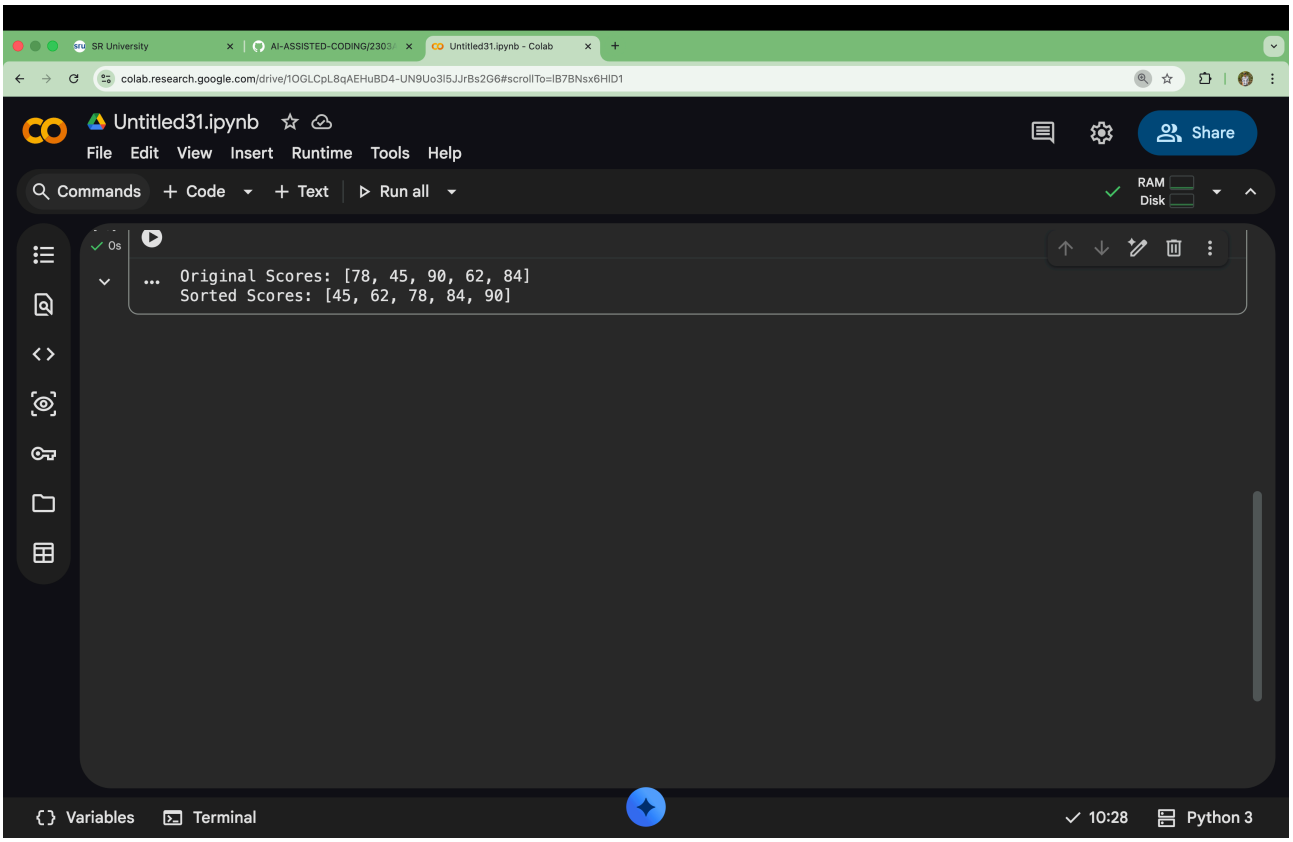
#TASK 1:

PROMPT:
Generate a Python Bubble Sort program to sort student exam scores.Add in line comments explaining comparisons, swaps, early termination condition, and include time complexity analysis with sample input/output.

CODE:



OUTPUT:



EXPLANATION:

1) Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order.

2) After each pass, the largest unsorted element moves to its correct position.

3) The swapped flag helps detect if the list is already sorted.

4) If no swaps occur in a pass, the algorithm terminates early.

5) It is simple but inefficient for large datasets due to $O(n^2)$ time complexity.

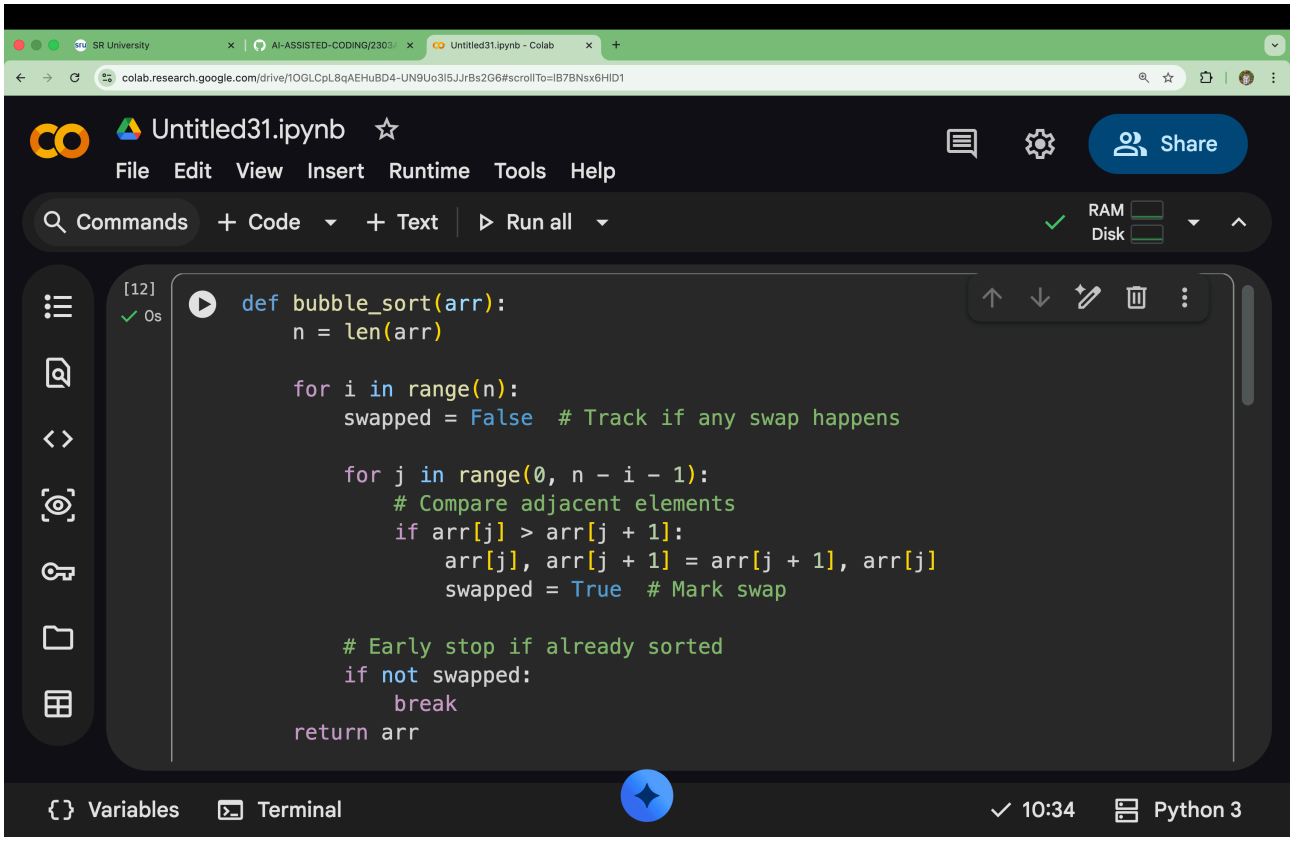
#**TASK 2:**

PROMPT:

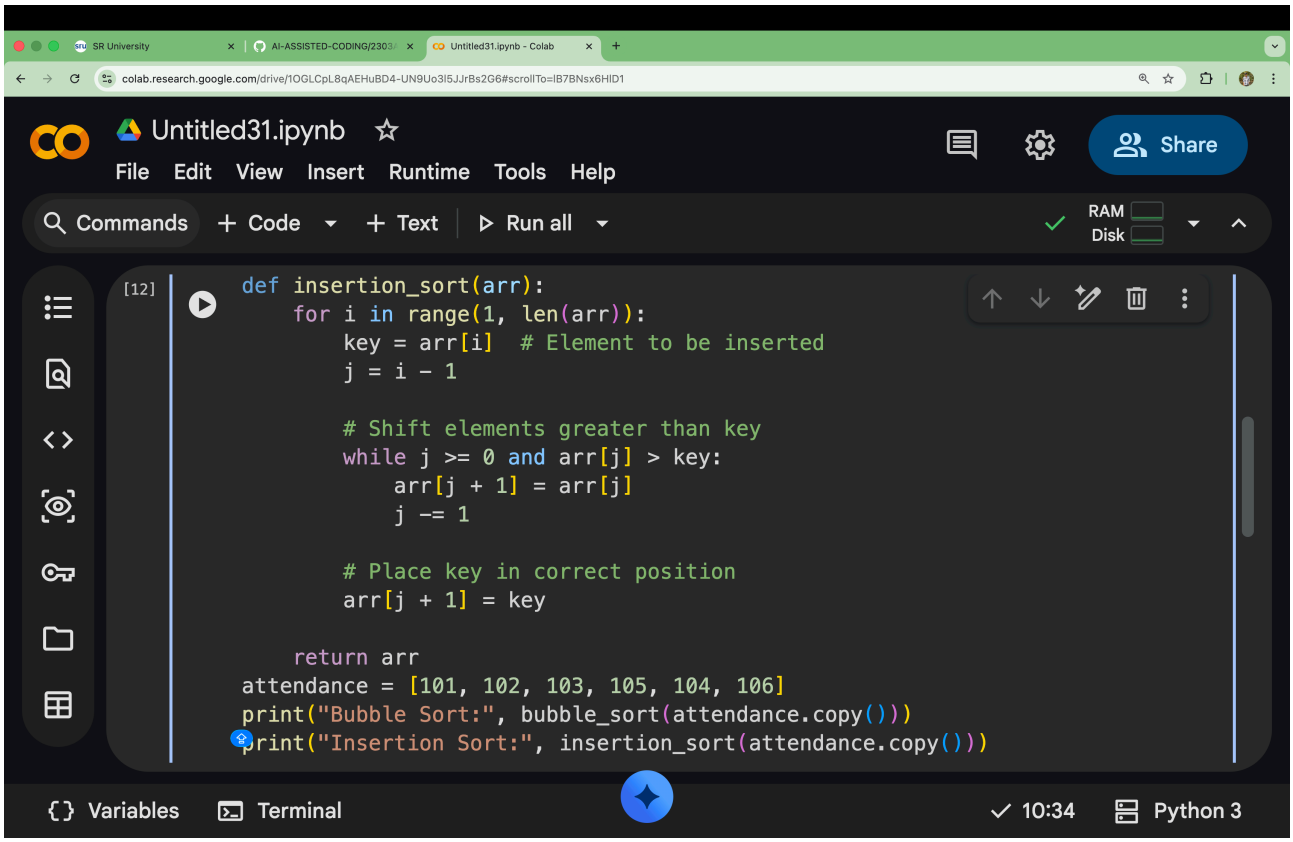
Review a nearly sorted attendance list and suggest a better sorting algorithm than Bubble Sort.Generate both Bubble Sort and Insertion Sort implementations and explain why Insertion Sort is more efficient for nearly sorted data.

CODE:

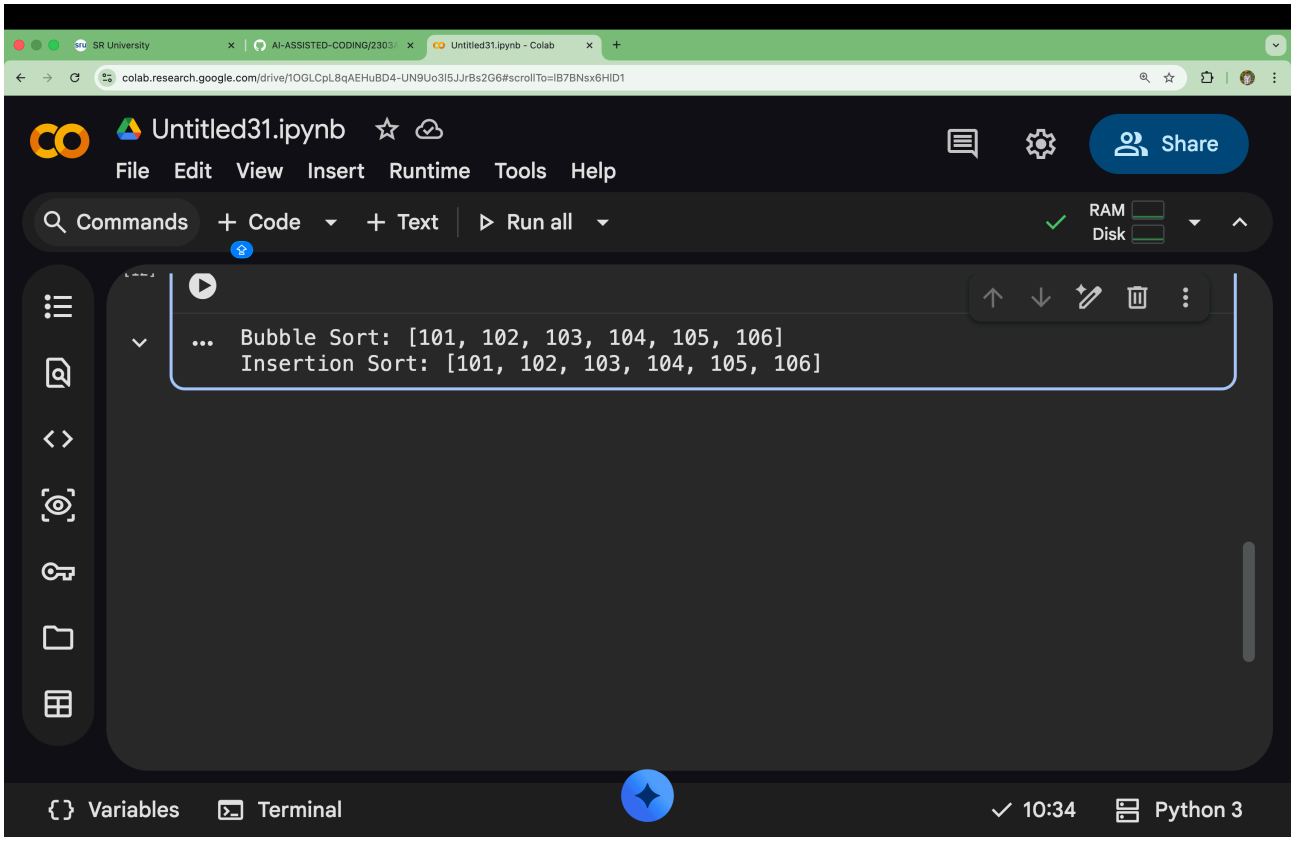
Bubble Sort implementation:



Insertion Sort implementation:



OUTPUT:

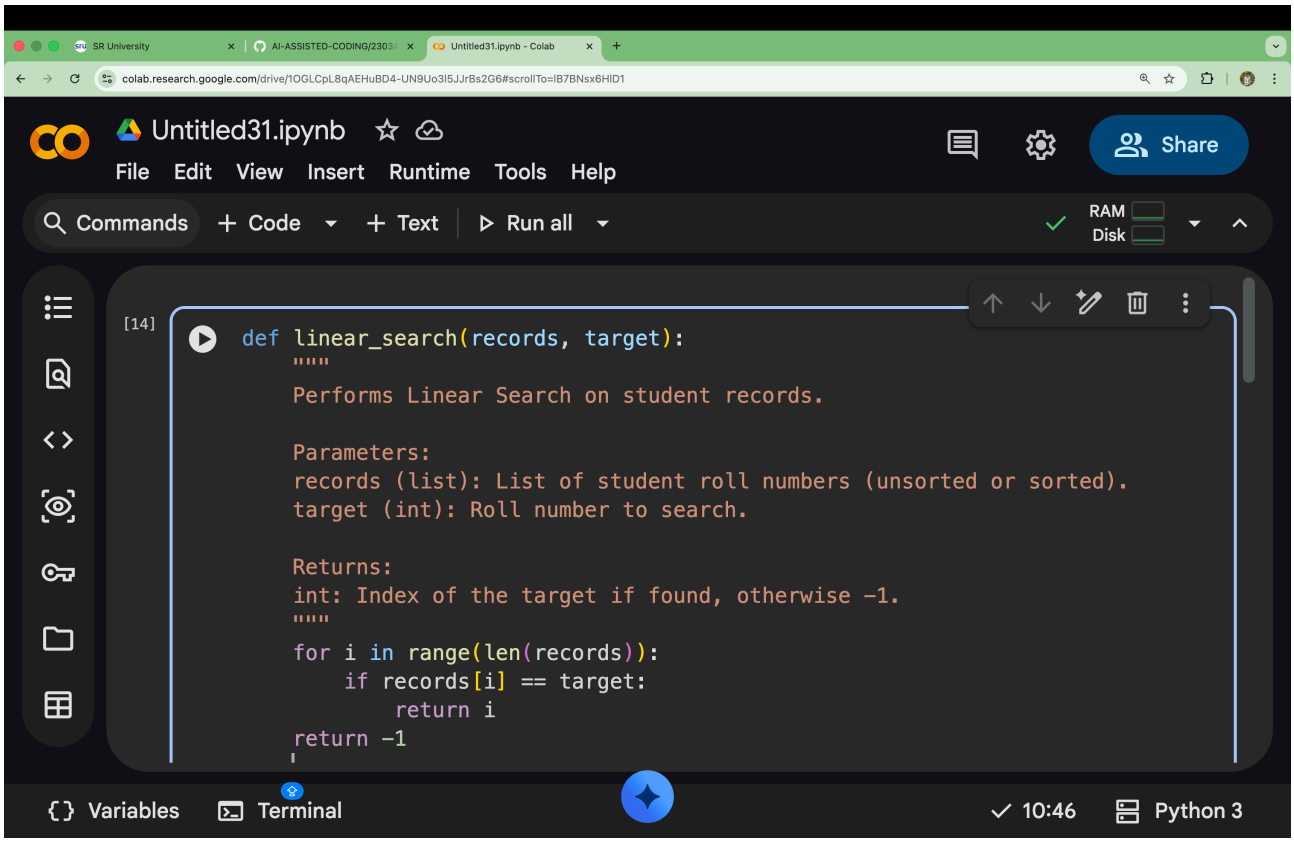


EXPLANATION:
1) Insertion Sort shifts only misplaced elements instead of repeatedly comparing all pairs.
2) If the list is nearly sorted, very few shifts are required.
3) Its best-case time complexity is O(n) when the list is already sorted.
4) Bubble Sort still performs multiple unnecessary comparisons even if the list is almost sorted.
5) Therefore, Insertion Sort executes faster for partially sorted datasets like attendance records.

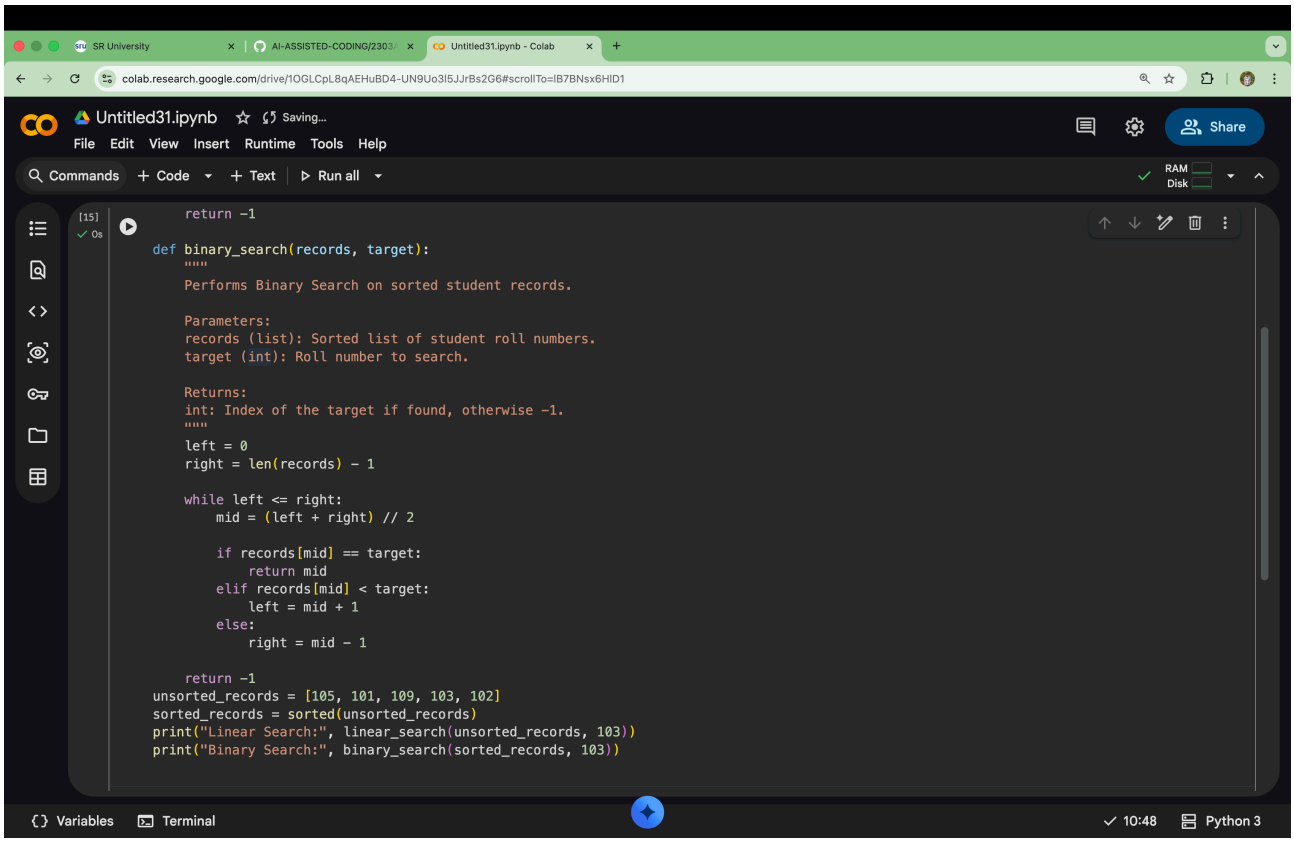
#**TASK 3:**

PROMPT:
Generate Python implementations of Linear Search and Binary Search for student roll numbers.Add proper docstrings, explain time complexity, and compare their performance on sorted and unsorted data.

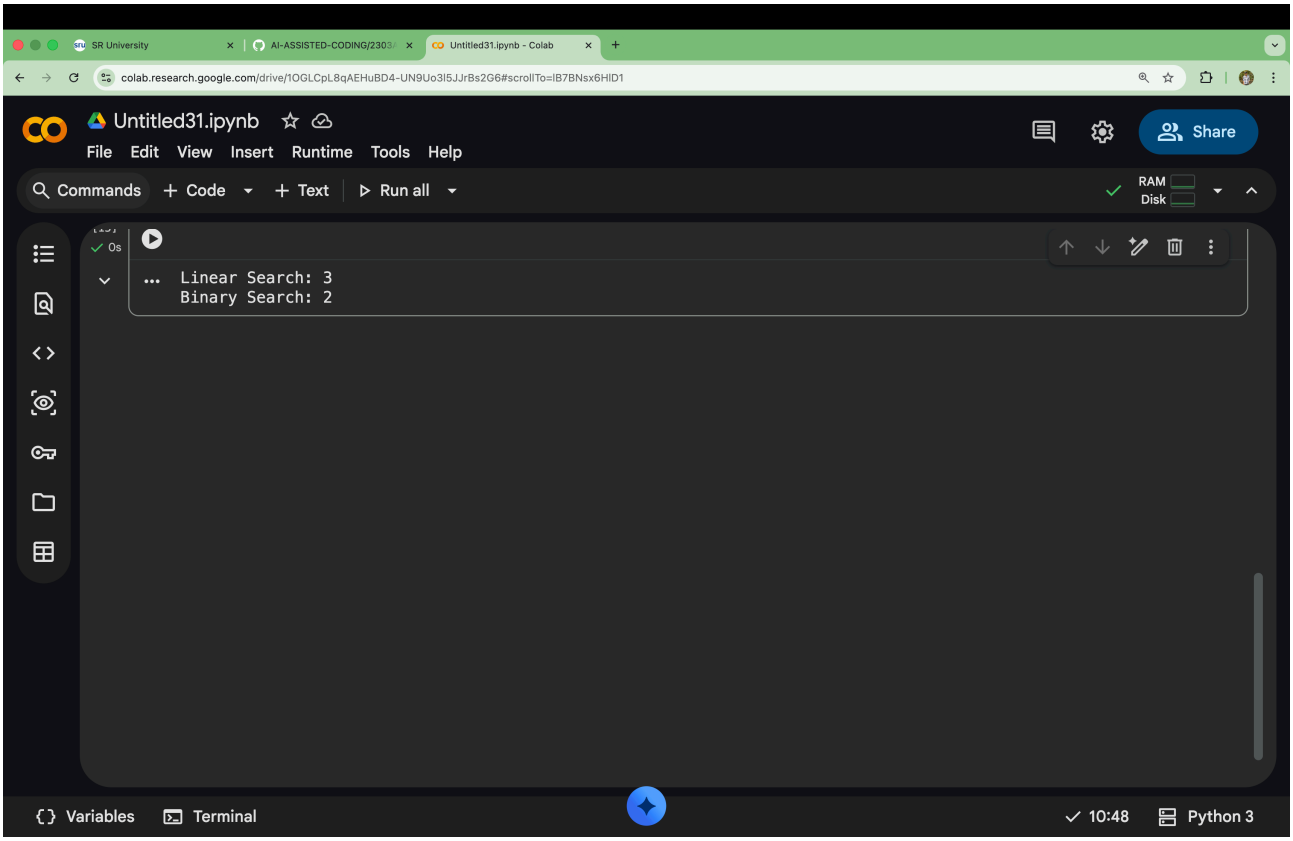
CODE:
Linear Search for unsorted student data



Binary Search for sorted student data:



OUTPUT:



Time Complexity Explanation:

- **Linear Search**

Best Case → O(1)
Average Case → O(n)
Worst Case → O(n)

- **Binary Search**

Best Case → O(1)
Average Case → O(log n)
Worst Case → O(log n)

EXPLANATION:

When searching in an unsorted list, Linear Search is required because elements are not arranged in order. In a sorted list, Binary Search performs much faster since it reduces the search space by half each time. For large student databases, Binary Search is more efficient. However, sorting is necessary before applying Binary Search.

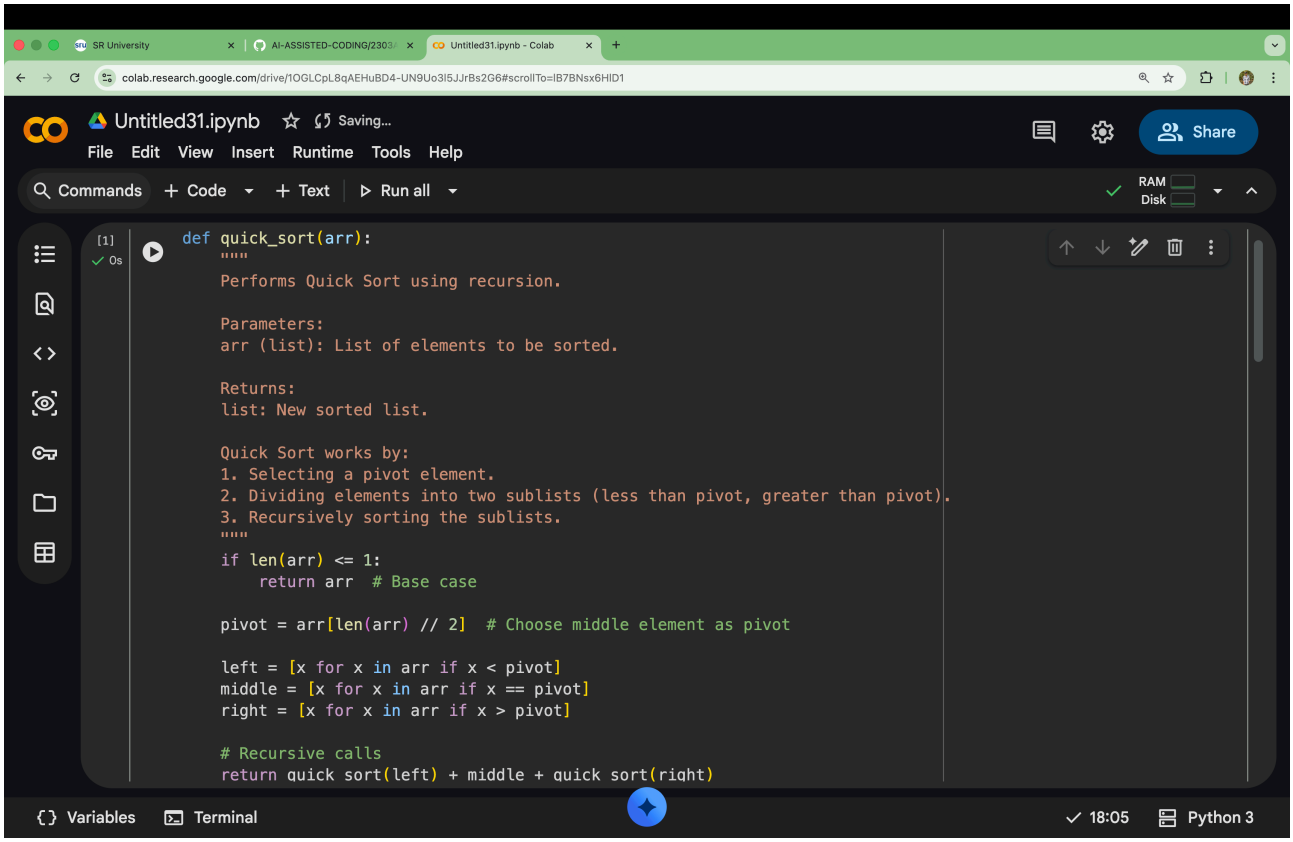
#TASK 4:

PROMPT:

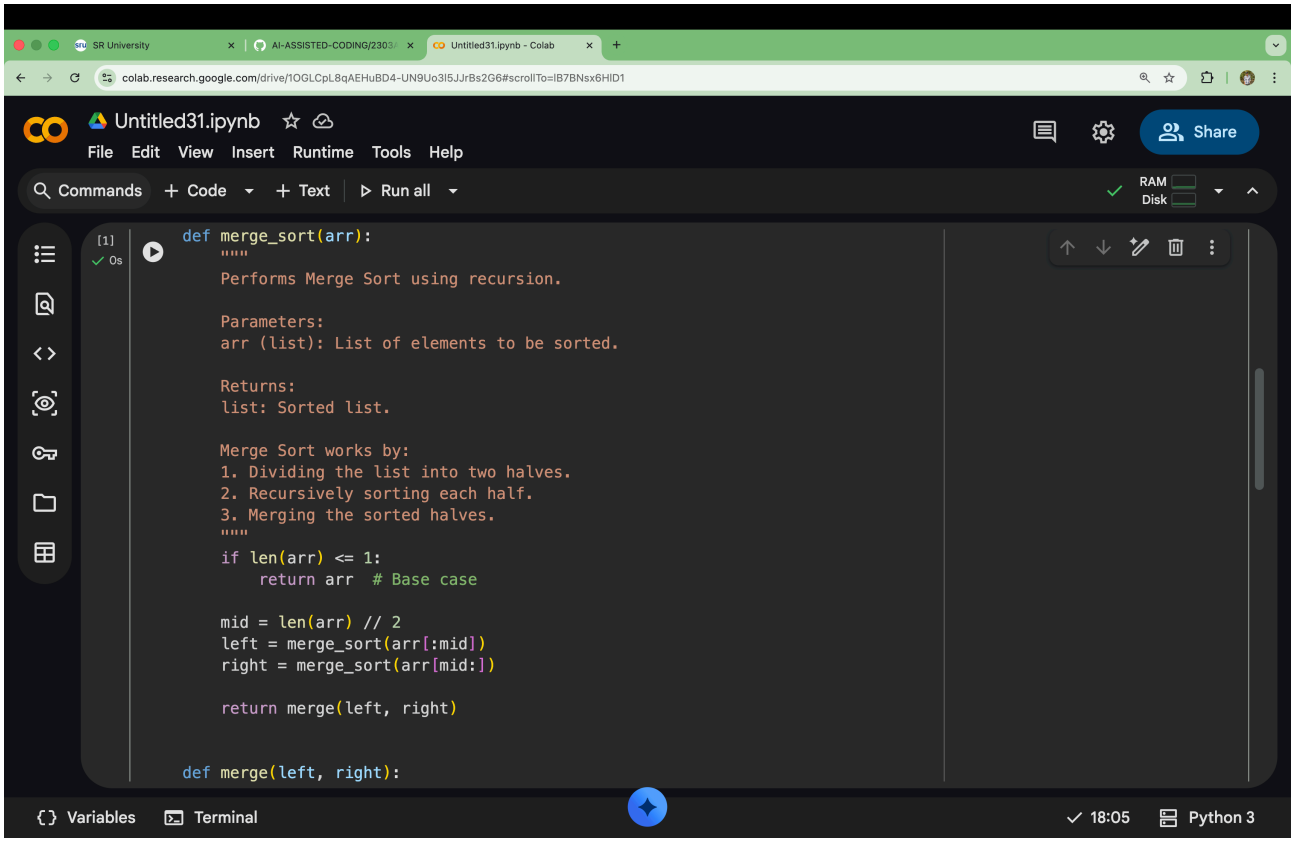
Complete recursive implementations of Quick Sort and Merge Sort in Python with proper docstrings.Test them on random, sorted, and reverse-sorted datasets and compare their time complexities and practical use cases.

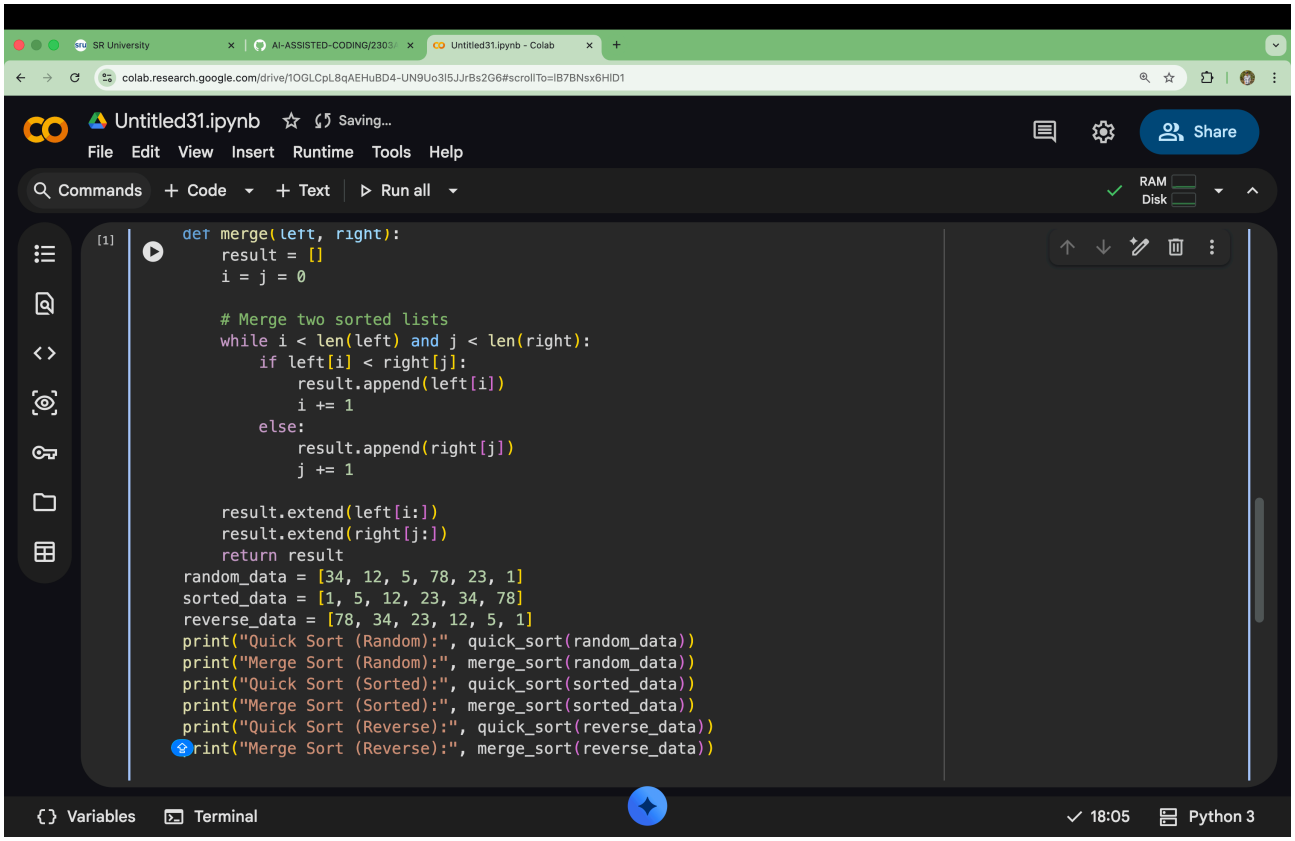
CODE:

Quick Sort:

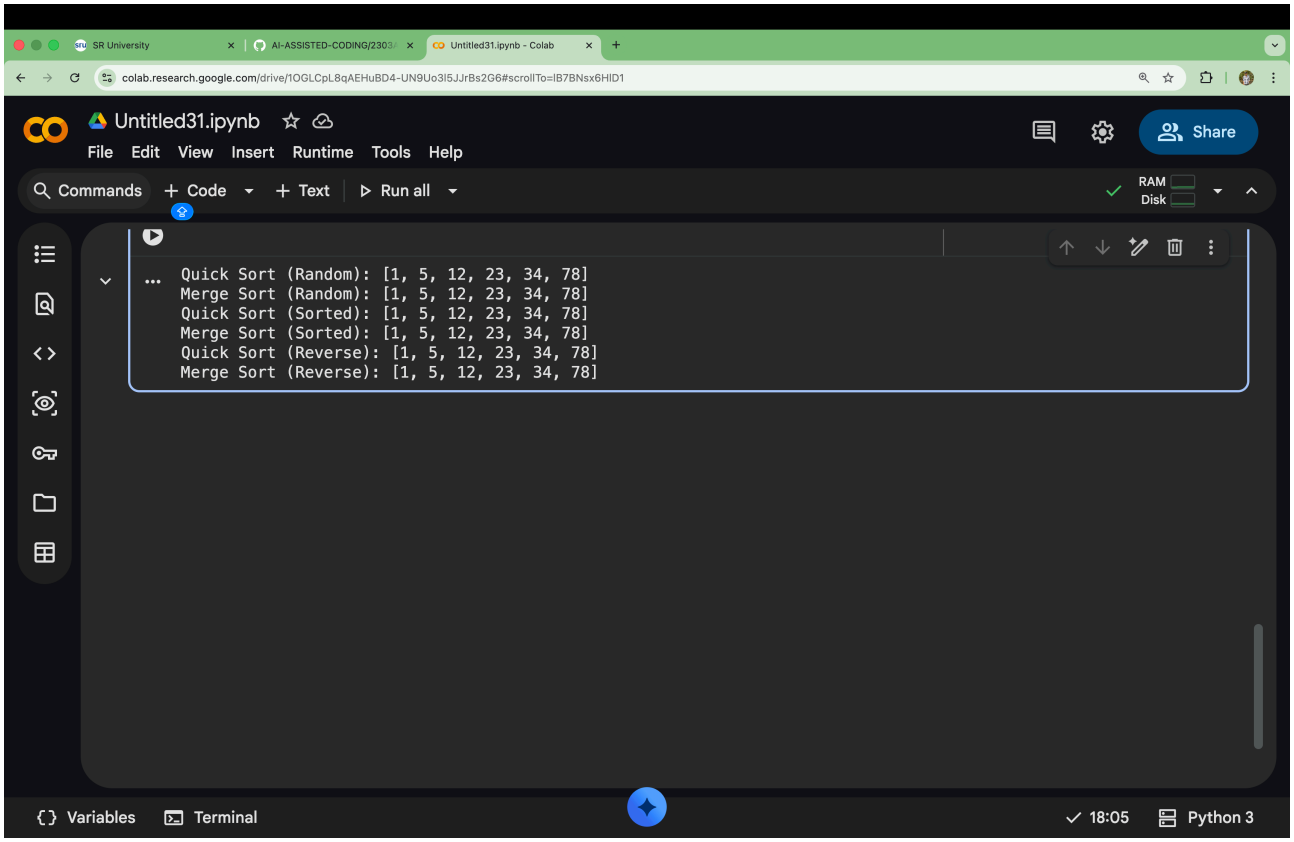


Merge Sort:


SR University
AI-ASSISTED-CODING/2303
Untitled31.ipynb - Colab
colab.research.google.com/drive/1OGLCpL8qAEHuBD4-UN9Uo3l5JJrBs2G6#scrollTo=lB7BNsx6HlD1
Untitled31.ipynb
File Edit View Insert Runtime Tools Help
Share
Commands + Code + Text Run all
RAM
Disk
[1]
0s
def merge_sort(arr):
"""
Performs Merge Sort using recursion.
Parameters:
arr (list): List of elements to be sorted.
Returns:
list: Sorted list.
Merge Sort works by:
1. Dividing the list into two halves.
2. Recursively sorting each half.
3. Merging the sorted halves.
"""
if len(arr) <= 1:
return arr # Base case
mid = len(arr) // 2
left = merge_sort(arr[:mid])
right = merge_sort(arr[mid:])
return merge(left, right)
def merge(left, right):
Variables
Terminal
18:05
Python 3


SR University
AI-ASSISTED-CODING/2303
Untitled31.ipynb - Colab
colab.research.google.com/drive/1OGLCpL8qAEHuBD4-UN9Uo3l5JJrBs2G6#scrollTo=lB7BNsx6HlD1
Untitled31.ipynb
Saving...
File Edit View Insert Runtime Tools Help
Share
Commands + Code + Text Run all
RAM
Disk
[1]
def merge(left, right):
result = []
i = j = 0
# Merge two sorted lists
while i < len(left) and j < len(right):
if left[i] < right[j]:
result.append(left[i])
i += 1
else:
result.append(right[j])
j += 1
result.extend(left[i:])
result.extend(right[j:])
return result
random_data = [34, 12, 5, 78, 23, 1]
sorted_data = [1, 5, 12, 23, 34, 78]
reverse_data = [78, 34, 23, 12, 5, 1]
print("Quick Sort (Random):", quick_sort(random_data))
print("Merge Sort (Random):", merge_sort(random_data))
print("Quick Sort (Sorted):", quick_sort(sorted_data))
print("Merge Sort (Sorted):", merge_sort(sorted_data))
print("Quick Sort (Reverse):", quick_sort(reverse_data))
print("Merge Sort (Reverse):", merge_sort(reverse_data))
Variables
Terminal
18:05
Python 3

OUTPUT:



EXPLANATION:

- Quick Sort is faster in practice and uses less memory, but worst case is $O(n^2)$.
- Merge Sort guarantees O(n log n) in all cases but requires extra memory.
- For random large datasets → Quick Sort is usually preferred.
- For guaranteed performance and stable sorting → Merge Sort is better.
- For reverse-sorted data → Merge Sort performs more consistently.

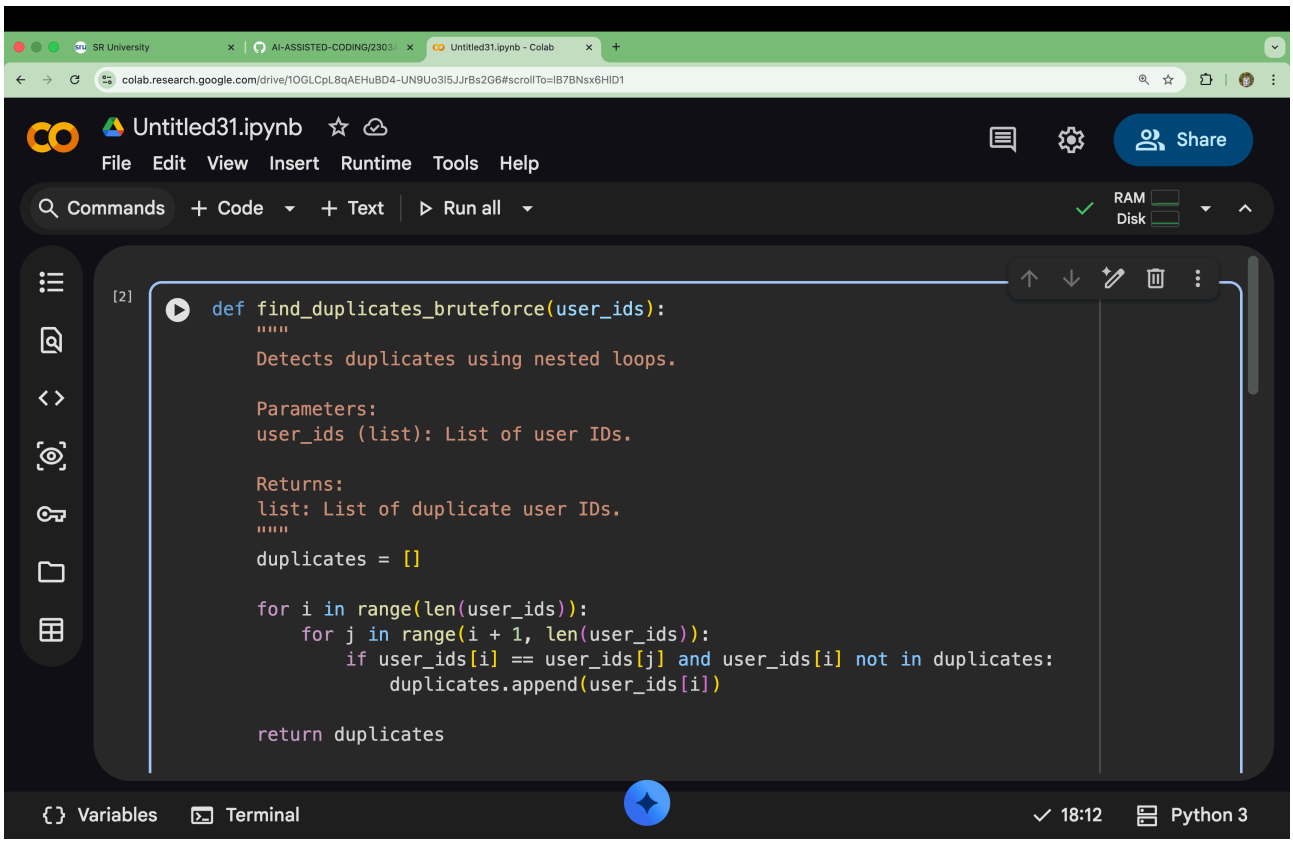
#TASK 5:

PROMPT:

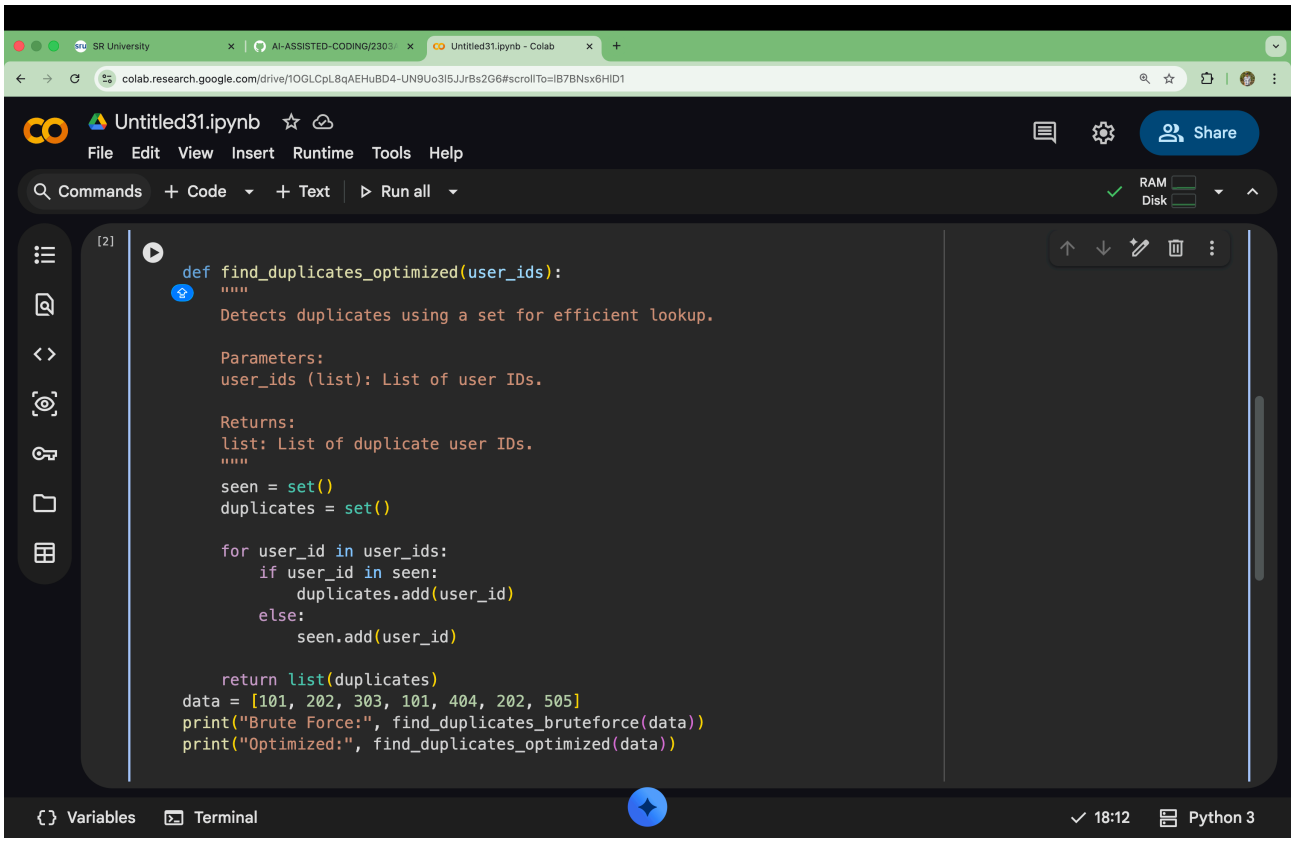
Generate a naive duplicate detection algorithm using nested loops and analyze its time complexity.Suggest and implement an optimized version using sets or dictionaries and compare performance for large datasets.

CODE:

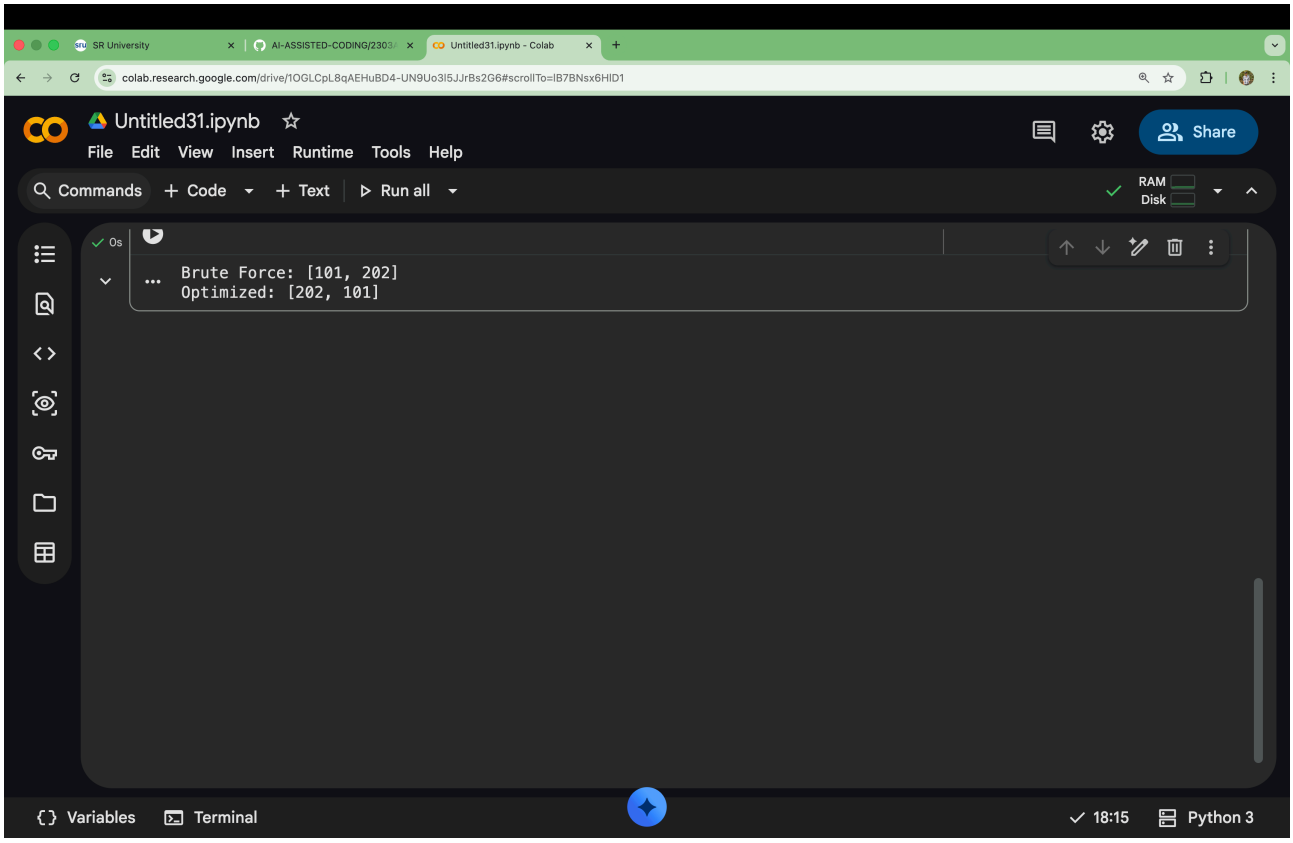
Brute-force ($O(n^2)$):



Optimized (O(n)):



OUTPUT:



EXPLANATION:

1) Nested loops cause repeated comparisons in brute-force.
2) Sets use hashing for constant-time lookups.
3) Each element is processed only once in optimized version.
4) Time complexity reduced from $O(n^2)$ to $O(n)$.
5) his makes the solution scalable for large data validation systems.