

## ASSIGNMENT-12

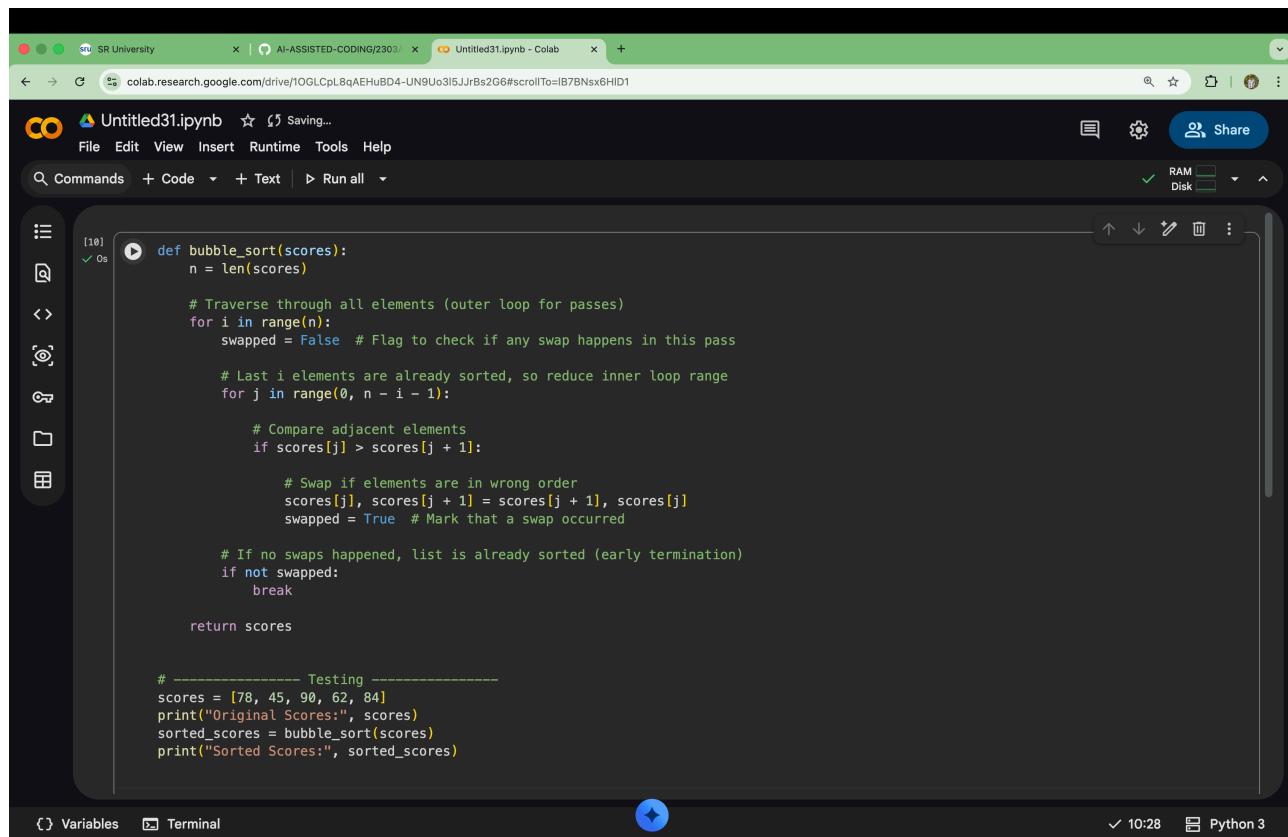
**NAME:**B.MANISHWAR  
**HALL TICKET:**2303A51276  
**BATCH:**05

### #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:



```
def bubble_sort(scores):
    n = len(scores)

    # Traverse through all elements (outer loop for passes)
    for i in range(n):
        swapped = False # Flag to check if any swap happens in this pass

        # Last i elements are already sorted, so reduce inner loop range
        for j in range(0, n - i - 1):

            # Compare adjacent elements
            if scores[j] > scores[j + 1]:

                # Swap if elements are in wrong order
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                swapped = True # Mark that a swap occurred

            # If no swaps happened, list is already sorted (early termination)
            if not swapped:
                break

    return scores

# ----- Testing -----
scores = [78, 45, 90, 62, 84]
print("Original Scores:", scores)
sorted_scores = bubble_sort(scores)
print("Sorted Scores:", sorted_scores)
```

#### OUTPUT:

```
Original Scores: [78, 45, 90, 62, 84]
Sorted Scores: [45, 62, 78, 84, 90]
```

## 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:

```
[12] 0s
def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
        swapped = False # Track if any swap happens

        for j in range(0, n - i - 1):
            # Compare adjacent elements
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True # Mark swap

            # Early stop if already sorted
            if not swapped:
                break
    return arr
```

Variables Terminal ✓ 10:34 Python 3

## Insertion Sort implementation:

```
[12]
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i] # Element to be inserted
        j = i - 1

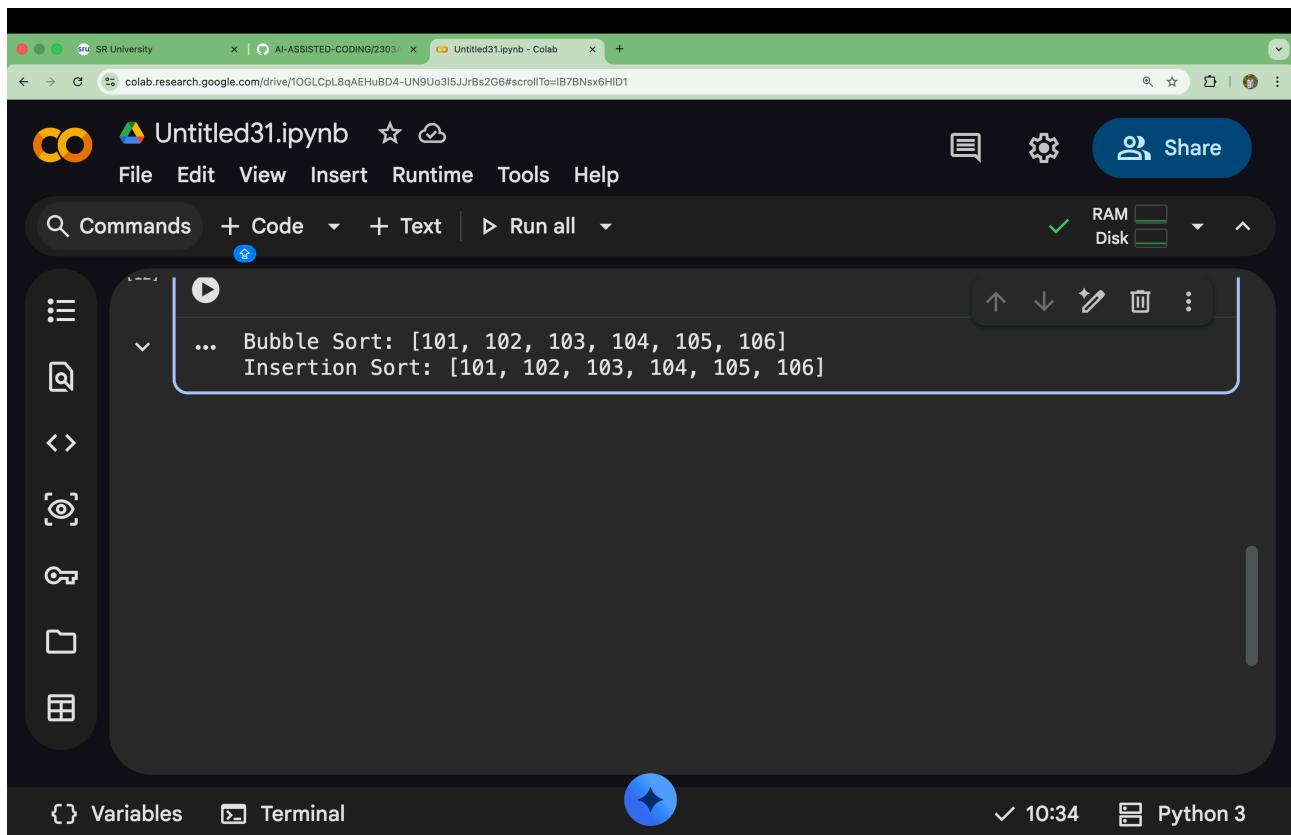
        # Shift elements greater than key
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Place key in correct position
        arr[j + 1] = key

    return arr
attendance = [101, 102, 103, 105, 104, 106]
print("Bubble Sort:", bubble_sort(attendance.copy()))
print("Insertion Sort:", insertion_sort(attendance.copy()))
```

Variables Terminal ✓ 10:34 Python 3

## OUTPUT:



```
... Bubble Sort: [101, 102, 103, 104, 105, 106]
Insertion Sort: [101, 102, 103, 104, 105, 106]
```

## 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

The screenshot shows a Google Colab notebook titled "Untitled31.ipynb". The code cell [14] contains the following Python code:

```
def linear_search(records, target):
    """
    Performs Linear Search on student records.

    Parameters:
    records (list): List of student roll numbers (unsorted or sorted).
    target (int): Roll number to search.

    Returns:
    int: Index of the target if found, otherwise -1.
    """
    for i in range(len(records)):
        if records[i] == target:
            return i
    return -1
```

The code is annotated with docstrings explaining the purpose, parameters, and returns of the function. The Colab interface includes a sidebar with various icons, a toolbar at the top, and a status bar at the bottom indicating the time as 10:46 and the Python version as 3.

## Binary Search for sorted student data:

The screenshot shows a Google Colab notebook titled "Untitled31.ipynb". The code cell [15] contains the following Python code:

```
def binary_search(records, target):
    """
    Performs Binary Search on sorted student records.

    Parameters:
    records (list): Sorted list of student roll numbers.
    target (int): Roll number to search.

    Returns:
    int: Index of the target if found, otherwise -1.
    """
    left = 0
    right = len(records) - 1

    while left <= right:
        mid = (left + right) // 2

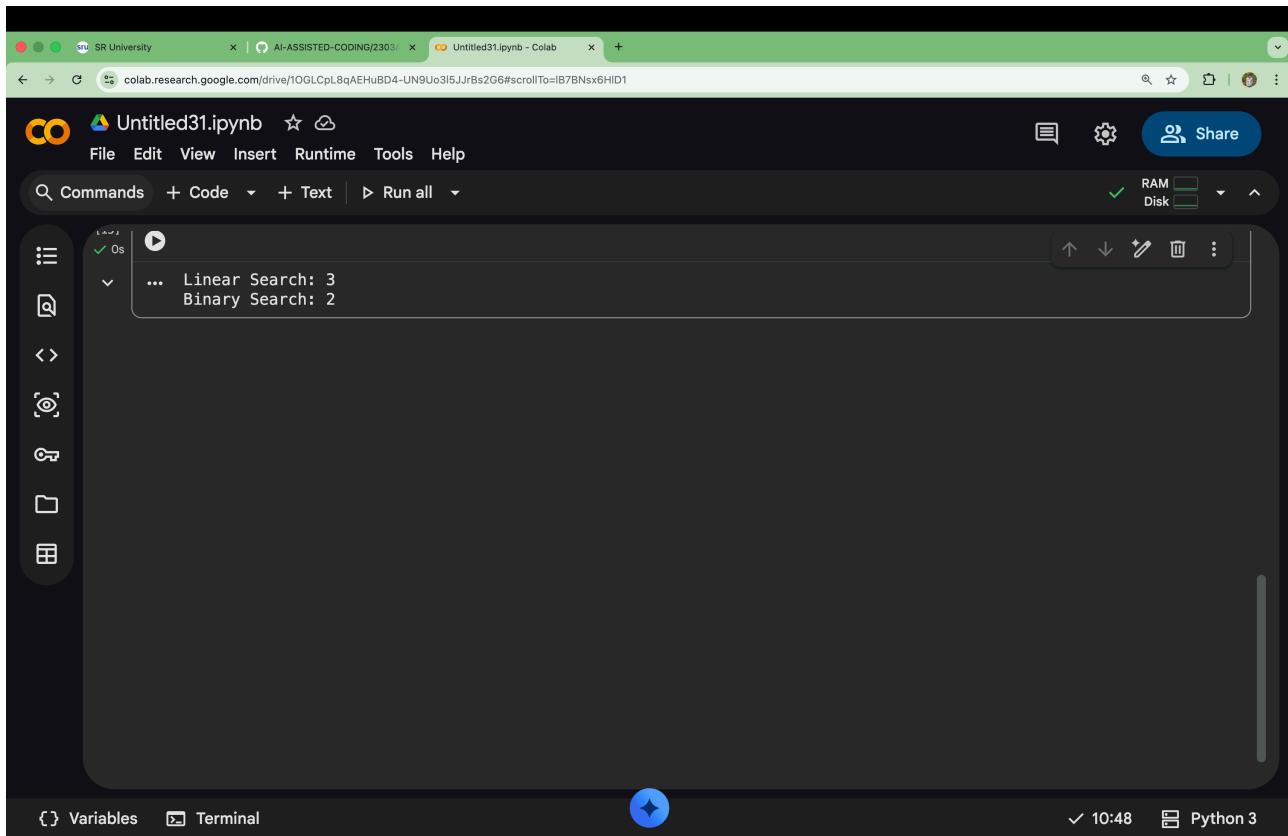
        if records[mid] == target:
            return mid
        elif records[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1

unsorted_records = [105, 101, 109, 103, 102]
sorted_records = sorted(unsorted_records)
print("Linear Search:", linear_search(unsorted_records, 103))
print("Binary Search:", binary_search(sorted_records, 103))
```

The code defines a binary search function for sorted lists and demonstrates it by comparing it with a linear search on an unsorted list. The Colab interface is identical to the previous screenshot, showing the same sidebar, toolbar, and status bar.

## OUTPUT:



Untitled31.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

RAM Disk

Linear Search: 3  
Binary Search: 2

Variables Terminal

10:48 Python 3

## 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.

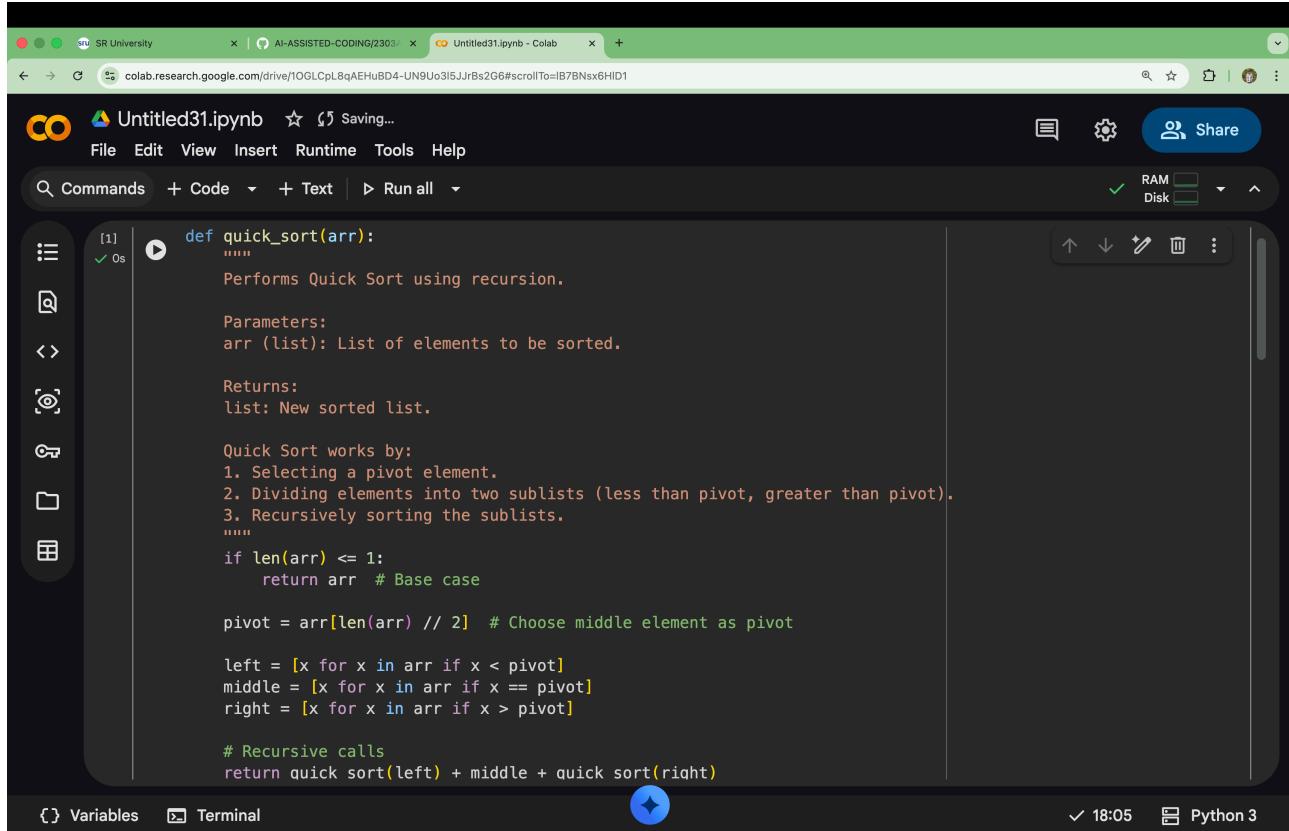
## #TASK 4:

## 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:



The screenshot shows a Google Colab notebook titled "Untitled31.ipynb". The code cell contains the following implementation of Quick Sort:

```
[1]: def quick_sort(arr):
    """
    Performs Quick Sort using recursion.

    Parameters:
    arr (list): List of elements to be sorted.

    Returns:
    list: New sorted list.

    Quick Sort works by:
    1. Selecting a pivot element.
    2. Dividing elements into two sublists (less than pivot, greater than pivot).
    3. Recursively sorting the sublists.
    """
    if len(arr) <= 1:
        return arr # Base case

    pivot = arr[len(arr) // 2] # Choose middle element as pivot

    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    # Recursive calls
    return quick_sort(left) + middle + quick_sort(right)
```

The notebook interface includes a sidebar with various icons, a toolbar with "File", "Edit", "View", etc., and a status bar at the bottom indicating "18:05" and "Python 3".

### Merge Sort:

```
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):
```

```
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))
```

## OUTPUT:

The screenshot shows a Google Colab notebook titled "Untitled31.ipynb". The code cell contains the following output:

```
... Quick Sort (Random): [1, 5, 12, 23, 34, 78]
Merge Sort (Random): [1, 5, 12, 23, 34, 78]
Quick Sort (Sorted): [1, 5, 12, 23, 34, 78]
Merge Sort (Sorted): [1, 5, 12, 23, 34, 78]
Quick Sort (Reverse): [1, 5, 12, 23, 34, 78]
Merge Sort (Reverse): [1, 5, 12, 23, 34, 78]
```

The notebook interface includes a sidebar with various icons, a toolbar at the top, and a status bar at the bottom indicating "18:05" and "Python 3".

## 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 ( $O(n^2)$ ):

The screenshot shows a Google Colab notebook titled "Untitled31.ipynb". The code cell [2] contains the following Python function:

```
[2]: def find_duplicates_bruteforce(user_ids):
    """
    Detects duplicates using nested loops.

    Parameters:
    user_ids (list): List of user IDs.

    Returns:
    list: List of duplicate user IDs.
    """
    duplicates = []

    for i in range(len(user_ids)):
        for j in range(i + 1, len(user_ids)):
            if user_ids[i] == user_ids[j] and user_ids[i] not in duplicates:
                duplicates.append(user_ids[i])

    return duplicates
```

The notebook interface includes a sidebar with various icons, a toolbar with "File", "Edit", "View", etc., and a status bar at the bottom showing "Variables", "Terminal", "18:12", and "Python 3".

## Optimized ( $O(n)$ ):

The screenshot shows a Google Colab notebook titled "Untitled31.ipynb". The code cell [2] contains the following Python function:

```
[2]: def find_duplicates_optimized(user_ids):
    """
    Detects duplicates using a set for efficient lookup.

    Parameters:
    user_ids (list): List of user IDs.

    Returns:
    list: List of duplicate user IDs.
    """
    seen = set()
    duplicates = set()

    for user_id in user_ids:
        if user_id in seen:
            duplicates.add(user_id)
        else:
            seen.add(user_id)

    return list(duplicates)
data = [101, 202, 303, 101, 404, 202, 505]
print("Brute Force:", find_duplicates_bruteforce(data))
print("Optimized:", find_duplicates_optimized(data))
```

The notebook interface includes a sidebar with various icons, a toolbar with "File", "Edit", "View", etc., and a status bar at the bottom showing "Variables", "Terminal", "18:12", and "Python 3".

## OUTPUT:

The screenshot shows a Google Colab notebook titled "Untitled31.ipynb". The code cell contains the following output:

```
Brute Force: [101, 202]
Optimized: [202, 101]
```

The Colab interface includes a sidebar with file management icons, a toolbar with "File", "Edit", "View", etc., and a status bar at the bottom indicating "18:15" and "Python 3".

## 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) This makes the solution scalable for large data validation systems.