

Name: I. Abhinay H.No: 2303A51811 Batch: 26

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING																																					
Program Name: B. Tech		Assignment Type: Lab	Academic Year: 2025-2026																																				
SCourse Coordinator Name		Dr. Rishabh Mittal																																					
Instructor(s) Name		<table border="1"> <tr><td>Mr. S Naresh Kumar</td><td></td></tr> <tr><td>Ms. B. Swathi</td><td></td></tr> <tr><td>Dr. Sasanko Shekhar Gantayat</td><td></td></tr> <tr><td>Mr. Md Sallauddin</td><td></td></tr> <tr><td>Dr. Mathivanan</td><td></td></tr> <tr><td>Mr. Y Srikanth</td><td></td></tr> <tr><td>Ms. N Shilpa</td><td></td></tr> <tr><td>Dr. Rishabh Mittal (Coordinator)</td><td></td></tr> <tr><td>Dr. R. Prashant Kumar</td><td></td></tr> <tr><td>Mr. Ankushavali MD</td><td></td></tr> <tr><td>Mr. B Viswanath</td><td></td></tr> <tr><td>Ms. Sujitha Reddy</td><td></td></tr> <tr><td>Ms. A. Anitha</td><td></td></tr> <tr><td>Ms. M. Madhuri</td><td></td></tr> <tr><td>Ms. Katherashala Swetha</td><td></td></tr> <tr><td>Ms. Velpula sumalatha</td><td></td></tr> <tr><td>Mr. Bingi Raju</td><td></td></tr> <tr><td>Mr. G. Kranthi</td><td></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. G. Kranthi	
		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. G. Kranthi																																							
Course Code	23CS002PC304	Course Title	AI Assisted Coding																																				
Year/Sem	III/I	Regulation	R23																																				
Date and Day of Assignment	Week 6 - Thursday	Time(s)	23CSBTB01 To 23CSBTB52																																				
Duration	2 Hours	Applicable to Batches	All Batches																																				
Assignment Number: 12.4 (Present assignment number) / 24 (Total number of assignments)																																							
Q.No.	Question	Expected Time to complete																																					
1	Lab 12 – Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms	Week 6																																					

	<p><b>Lab Objectives</b></p> <ul style="list-style-type: none"> <li>• Apply AI-assisted programming to implement and optimize sorting and searching algorithms.</li> <li>• Compare different algorithms in terms of efficiency and use cases.</li> <li>• Understand how AI tools can suggest optimized code and complexity improvements.</li> </ul> <p><b>Learning Outcome</b></p> <p>After completing this assignment, students will be able to:</p> <ul style="list-style-type: none"> <li>• Implement classical algorithms with AI assistance</li> <li>• Compare algorithm efficiency using real-world scenarios</li> <li>• Understand when optimization is necessary</li> <li>• Critically evaluate AI-generated suggestions instead of blindly accepting them</li> </ul>	
	<p><b>Task 1: Bubble Sort for Ranking Exam Scores</b></p> <p><b>Scenario</b></p> <p>You are working on a <b>college result processing system</b> where a small list of student scores needs to be sorted after every internal assessment.</p> <p><b>Task Description</b></p> <ul style="list-style-type: none"> <li>• Implement <b>Bubble Sort</b> in Python to sort a list of student scores.</li> <li>• Use an AI tool to: <ul style="list-style-type: none"> <li>○ Insert inline comments explaining key operations such as comparisons, swaps, and iteration passes</li> <li>○ Identify early-termination conditions when the list becomes sorted</li> <li>○ Provide a brief time complexity analysis</li> </ul> </li> </ul> <p><b>Expected Outcome</b></p> <ul style="list-style-type: none"> <li>• A Bubble Sort implementation with: <ul style="list-style-type: none"> <li>○ AI-generated comments explaining the logic</li> <li>○ Clear explanation of best, average, and worst-case complexity</li> <li>○ Sample input/output showing sorted scores</li> </ul> </li> </ul>	

```
C:\Users\shash> Downloads > AAC A 12.4.py > bubble_sort
1 def bubble_sort(scores):
2     n = len(scores)
3     for i in range(n - 1):
4         swapped = False
5         for j in range(n - 1 - i):
6             if scores[j] > scores[j + 1]:
7                 scores[j], scores[j + 1] = scores[j + 1], scores[j]
8                 swapped = True
9         if not swapped:
10             break
11     return scores
12
13 scores = [85, 42, 97, 63, 51, 74, 88]
14 print("Before:", scores)
15 print("After:", bubble_sort(scores))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

conda: the term conda is not recognized as the name of a command, function, script, file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.

At line:1 char:1  
+ conda activate Shashidhar  
+ ~~~~~  
+ CategoryInfo : ObjectNotFound: (conda:String) [], CommandNotFoundException  
+ FullyQualifiedErrorId : CommandNotFoundException

PS C:\Users\shash\Downloads> & "c:\Users\shash\anaconda3\envs\Shashidhar\python.exe" "c:\Users\shash\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher" "53387" "-." "c:\Users\shash\Downloads\AAC A 12.4.py"

Before: [85, 42, 97, 63, 51, 74, 88]  
After: [42, 51, 63, 74, 85, 88, 97]

CHAT

BUBBLE SORT IMPLEMENTATION FOR S...

✓ Planned small Bubble Sort implementation in Python.

```
def bubble_sort(scores):
    n = len(scores)
    for i in range(n - 1):
        swapped = False
        for j in range(n - 1 - i):
            if scores[j] > score
            scores[j], score
            swapped = True
        if not swapped:
            break
    return scores
```

scores = [85, 42, 97, 63, 51, 74  
print("Before:", scores)  
print("After:", bubble\_sort(sco

Output:

Before: [85, 42, 97, 63, 51, 74,  
After: [42, 51, 63, 74, 85, 88, '

Time Complexity: Best  $O(n)$  (already sorted, early termination). Average/Worst  $O(n^2)$ .

AAC A 12.4.py

Describe what to build next

Claude Opus 4.6

# Task 2: Improving Sorting for Nearly Sorted Attendance Records

## Scenario

You are maintaining an **attendance system** where student roll numbers are already *almost sorted*, with only a few late updates.

## Task Description

- Start with a Bubble Sort implementation.
- Ask AI to:
  - Review the problem and suggest a more suitable sorting algorithm
  - Generate an **Insertion Sort** implementation
  - Explain why Insertion Sort performs better on nearly sorted data
- Compare execution behavior on nearly sorted input

## Expected Outcome

- Two sorting implementations:
  - Bubble Sort
  - Insertion Sort
- AI-assisted explanation highlighting efficiency differences for partially sorted datasets

The screenshot shows a VS Code editor with two Python files: `AAC A 12.4.py` and `BUBBLE SORT IMPLEMENTATION FOR S...`. The `AAC A 12.4.py` file contains the following code:

```

1 import time
2 def bubble_sort(arr):
3     n = len(arr)
4     for i in range(n - 1):
5         swapped = False
6         for j in range(n - 1 - i):
7             if arr[j] > arr[j + 1]:
8                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
9                 swapped = True
10        if not swapped:
11            break
12    return arr
13
14 def insertion_sort(arr):
15     for i in range(1, len(arr)):
16         key = arr[i]
17         j = i - 1
18         while j >= 0 and arr[j] > key:
19             arr[j + 1] = arr[j]
20             j -= 1
21         arr[j + 1] = key
22    return arr
23
24 nearly_sorted = [5, 3, 4, 2, 5, 6, 0, 7, 0, 10]

```

The terminal output shows the execution of the code:

```

PS C:\Users\shash\Downloads> & 'c:\Users\shash\anaconda3\envs\Shashi\python.exe' 'c:\Users\shash\vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher' '53307' '-.' 'c:\Users\shash\Downloads\AAC A 12.4.py'
Before: [85, 42, 97, 63, 51, 74, 88]
After: [42, 51, 63, 74, 85, 88, 97]
PS C:\Users\shash\Downloads> cd 'c:\Users\shash\Downloads'; & 'c:\Users\shash\anaconda3\envs\Shashi\python.exe' 'c:\Users\shash\vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher' '63307' '-.' 'c:\Users\shash\Downloads\AAC A 12.4.py'
Nearly Sorted Input: [1, 2, 4, 3, 5, 6, 8, 7, 9, 10]
Bubble Sort Result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Bubble Sort Time: 0.000178s
Insertion Sort Result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Insertion Sort Time: 0.000124s

```

The AI assistant sidebar on the right shows a list of actions: "Reviewed current file and requested simple Python code", "Reviewed and updated AAC A 12.4.py", and "Updated AAC A 12.4.py". It also includes a "give the code again" button and a "Describe what to build next" section.

```
Welcome AAC A 12.4.py
C: > Users > shash > Downloads > AAC A 12.4.py > ...
14 def insertion_sort(arr):
17     j = i - 1
18     while j >= 0 and arr[j] > key:
19         arr[j + 1] = arr[j]
20         j -= 1
21     arr[j + 1] = key
22     return arr
23
24 nearly_sorted = [1, 2, 4, 3, 5, 6, 8, 7, 9, 10]
25
26 print("Nearly Sorted Input:", nearly_sorted)
27
28 data1 = nearly_sorted.copy()
29 start = time.perf_counter()
30 print("Bubble Sort Result:", bubble_sort(data1))
31 print(f"Bubble Sort Time: {time.perf_counter() - start:.6f}s")
32
33 data2 = nearly_sorted.copy()
34 start = time.perf_counter()
35 print("Insertion Sort Result:", insertion_sort(data2))
36 print(f"Insertion Sort Time: {time.perf_counter() - start:.6f}s")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

● PS C:\Users\shash\Downloads> & 'c:\Users\shash\anaconda3\envs\Shashidhar\python
.18.0-win32-x64\bundle\libs\debugpy\launcher' '53307' '--' 'c:\Users\shash\Down
Before: [85, 42, 97, 63, 51, 74, 88]
After: [42, 51, 63, 74, 85, 88, 97]
● PS C:\Users\shash\Downloads> c::; cd 'c:\Users\shash\Downloads'; & 'c:\Users\sh
e\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher
Nearly Sorted Input: [1, 2, 4, 3, 5, 6, 8, 7, 9, 10]
Bubble Sort Result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Bubble Sort Time: 0.000178s
Insertion Sort Result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Insertion Sort Time: 0.000124s
● PS C:\Users\shash\Downloads> █
```

### Task 3: Searching Student Records in a Database

#### Scenario

You are developing a **student information portal** where users search for student records by roll number.

#### Task Description

- Implement:
  - **Linear Search** for unsorted student data
  - **Binary Search** for sorted student data
- Use AI to:
  - Add docstrings explaining parameters and return values
  - Explain when Binary Search is applicable
  - Highlight performance differences between the two searches

## Expected Outcome

- Two working search implementations with docstrings
- AI-generated explanation of:
  - Time complexity
  - Use cases for Linear vs Binary Search
- A short student observation comparing results on sorted vs unsorted lists

The screenshot shows a VS Code editor with a Python file named 'AAC A 12.4.py'. The code implements two search functions: `linear_search` and `binary_search`. The `linear_search` function iterates through a list until it finds the target or returns -1. The `binary_search` function uses a while loop to find the target in a sorted list, returning the index or -1. The code also includes test cases for unsorted and sorted lists, and a target value of 115.

The terminal output shows the execution of the code, displaying the unsorted and sorted lists, the target value, and the results of the searches. The `linear_search` function finds the target at index 4, while the `binary_search` function also finds the target at index 4.

The performance comparison table shows the time complexity of the two search algorithms:

	Linear Search	Binary Search
Best	$O(1)$	$O(1)$
Average	$O(n)$	$O(\log n)$
Worst	$O(n)$	$O(\log n)$
Requires sorted data	No	Yes

The table also includes a section titled 'When to use which:' which states that Linear Search works on unsorted data, while Binary Search requires sorted data and is more efficient for large datasets.

```
C:\Users\shash\Downloads> AAC A 12.4.py
18
19 unsorted = [112, 105, 121, 108, 115, 103, 119]
20 sorted_list = [103, 105, 108, 112, 115, 119, 121]
21 target = 115
22
23 print("Unsorted:", unsorted)
24 print("Sorted: ", sorted_list)
25 print("Searching for roll number:", target)
26
27 result1 = linear_search(unsorted, target)
28 print("Linear Search: Found at index", result1)
29
30 result2 = binary_search(sorted_list, target)
31 print("Binary Search: Found at index", result2)
32
33 missing = 999
34 print("\nSearching for roll number:", missing)
35 print("Linear Search:", "Not Found" if linear_search(unsorted, missing) == -1 else "Found")
36 print("Binary Search:", "Not Found" if binary_search(sorted_list, missing) == -1 else "Found")
```

PS C:\Users\shash\Downloads> cd 'c:\Users\shash\Downloads'; & 'c:\Users\shash\anaconda3\envs\Shashidhar\python.exe' 'c:\Users\shash\vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\lib\debugpy\launcher' '62305' '-.' 'c:\Users\shash\Downloads\AAC A 12.4.py'

Unsorted: [112, 105, 121, 108, 115, 103, 119]  
Sorted: [103, 105, 108, 112, 115, 119, 121]  
Searching for roll number: 115  
Linear Search: Found at index 4  
Binary Search: Found at index 4

Searching for roll number: 999  
Linear Search: Not Found  
Binary Search: Not Found

PS C:\Users\shash\Downloads>

**BUBBLE SORT IMPLEMENTATION FOR S...**

Binary Search: Found at index 4

Searching for roll number: 999  
Linear Search: Not Found  
Binary Search: Not Found

**Performance Comparison:**

	Linear Search	Binary Search
Best	$O(1)$	$O(1)$
Average	$O(n)$	$O(\log n)$
Worst	$O(n)$	$O(\log n)$
Requires sorted data	No	Yes

**When to use which:**

- **Linear Search** — works on unsorted data; suitable when data changes frequently and sorting cost is not justified.
- **Binary Search** — requires sorted data; far more efficient for large, static datasets like a pre-sorted student database.

## Task 4: Choosing Between Quick Sort and Merge Sort for Data Processing

### Scenario

You are part of a **data analytics team** that needs to sort large datasets received from different sources (random order, already sorted, and reverse sorted).

### Task Description

- Provide AI with partially written recursive functions for:
  - **Quick Sort**
  - **Merge Sort**
- Ask AI to:
  - Complete the recursive logic
  - Add meaningful docstrings
  - Explain how recursion works in each algorithm
- Test both algorithms on:
  - Random data

- Sorted data
- Reverse-sorted data

## Expected Outcome

- Fully functional Quick Sort and Merge Sort implementations
- AI-generated comparison covering:
  - Best, average, and worst-case complexities
  - Practical scenarios where one algorithm is preferred over the other

```

1 import time, random
2 def quick_sort(arr):
3     if len(arr) <= 1:
4         return arr
5     pivot = arr[len(arr) // 2]
6     left = [x for x in arr if x < pivot]
7     middle = [x for x in arr if x == pivot]
8     right = [x for x in arr if x > pivot]
9     return quick_sort(left) + middle + quick_sort(right)
10
11 def merge_sort(arr):
12     if len(arr) <= 1:
13         return arr
14     mid = len(arr) // 2
15     left = merge_sort(arr[:mid])
16     right = merge_sort(arr[mid:])
17     return merge(left, right)
18
19 def merge(left, right):
20     result = []
21     i = j = 0
22     while i < len(left) and j < len(right):
23         if left[i] <= right[j]:
24             result.append(left[i])
25             i += 1
26         else:
27             result.append(right[j])
28             j += 1
29     result.extend(left[i:])
30     result.extend(right[j:])
31     return result
32
33 def test_sort(name, sort_func, data):
34     start = time.perf_counter()
35     result = sort_func(data.copy())
36     elapsed = time.perf_counter() - start
37     print(f"({name}): (elapsed: {elapsed:.6f}s -> {result[:5]}...{result[-3:]})")
38
39 random_data = random.sample(range(1, 1001), 500)
40 sorted_data = list(range(1, 501))
41 reverse_data = list(range(500, 0, -1))
42 for label, data in [("Random", random_data), ("Sorted", sorted_data), ("Reverse", reverse_data)]:
43     print(f"\n{label} Data:")
44     test_sort("Quick Sort", quick_sort, data)
45     test_sort("Merge Sort", merge_sort, data)

```

Random Data:  
Quick Sort: 0.00021s -> [5, 10, 11, 15, 16]...[996, 997, 998]  
Merge Sort: 0.001611s -> [5, 10, 11, 15, 16]...[996, 997, 998]

Sorted Data:  
Quick Sort: 0.000537s -> [1, 2, 3, 4, 5]...[498, 499, 500]  
Merge Sort: 0.001203s -> [1, 2, 3, 4, 5]...[498, 499, 500]

Reverse Data:  
Quick Sort: 0.000677s -> [1, 2, 3, 4, 5]...[498, 499, 500]

```

19 def merge(left, right):
20     while i < len(left) and j < len(right):
21         if left[i] <= right[j]:
22             result.append(left[i])
23             i += 1
24         else:
25             result.append(right[j])
26             j += 1
27     result.extend(left[i:])
28     result.extend(right[j:])
29     return result
30
31 def test_sort(name, sort_func, data):
32     start = time.perf_counter()
33     result = sort_func(data.copy())
34     elapsed = time.perf_counter() - start
35     print(f"({name}): (elapsed: {elapsed:.6f}s -> {result[:5]}...{result[-3:]})")
36
37 random_data = random.sample(range(1, 1001), 500)
38 sorted_data = list(range(1, 501))
39 reverse_data = list(range(500, 0, -1))
40 for label, data in [("Random", random_data), ("Sorted", sorted_data), ("Reverse", reverse_data)]:
41     print(f"\n{label} Data:")
42     test_sort("Quick Sort", quick_sort, data)
43     test_sort("Merge Sort", merge_sort, data)

```

Random Data:  
Quick Sort: 0.00021s -> [5, 10, 11, 15, 16]...[996, 997, 998]  
Merge Sort: 0.001611s -> [5, 10, 11, 15, 16]...[996, 997, 998]

Sorted Data:  
Quick Sort: 0.000537s -> [1, 2, 3, 4, 5]...[498, 499, 500]  
Merge Sort: 0.001203s -> [1, 2, 3, 4, 5]...[498, 499, 500]

Reverse Data:  
Quick Sort: 0.000677s -> [1, 2, 3, 4, 5]...[498, 499, 500]  
Merge Sort: 0.001465s -> [1, 2, 3, 4, 5]...[498, 499, 500]



# Task 5: Optimizing a Duplicate Detection Algorithm

## Scenario

You are building a **data validation module** that must detect duplicate user IDs in a large dataset before importing it into a system.

## Task Description

- Write a **naive duplicate detection algorithm** using nested loops.
- Use AI to:
  - Analyze the time complexity
  - Suggest an optimized approach using sets or dictionaries
  - Rewrite the algorithm with improved efficiency
- Compare execution behavior conceptually for large input sizes

## Expected Outcome

- Two versions of the algorithm:
  - Brute-force ( $O(n^2)$ )
  - Optimized ( $O(n)$ )
- AI-assisted explanation showing how and why performance improved

The screenshot shows a code editor with a Python script for duplicate detection. The script defines two functions: `find_duplicates_brute` and `find_duplicates_optimized`. The brute-force function uses nested loops to check for duplicates, while the optimized function uses a set to track seen user IDs. The script also includes a list of user IDs and a performance comparison section.

```
1 import time, random
2 def find_duplicates_brute(user_ids):
3     duplicates = []
4     for i in range(len(user_ids)):
5         for j in range(i + 1, len(user_ids)):
6             if user_ids[i] == user_ids[j] and user_ids[i] not in duplicates:
7                 duplicates.append(user_ids[i])
8     return duplicates
9
10 def find_duplicates_optimized(user_ids):
11     seen = set()
12     duplicates = set()
13     for uid in user_ids:
14         if uid in seen:
15             duplicates.add(uid)
16         else:
17             seen.add(uid)
18     return list(duplicates)
19
20 user_ids = [101, 203, 305, 101, 407, 201, 509, 305, 611, 407]
21 print("User IDs:", user_ids)
22
23 start = time.perf_counter()
24 results = find_duplicates_brute(user_ids)
25 end = time.perf_counter()
26 print(f"Brute Force Time: {end - start} seconds")
27
28 start = time.perf_counter()
29 results = find_duplicates_optimized(user_ids)
30 end = time.perf_counter()
31 print(f"Optimized Time: {end - start} seconds")
```

The execution results show the following output:

```
User IDs: [101, 203, 305, 101, 407, 201, 509, 305, 611, 407]
Brute Force O(n^2): [101, 203, 305, 407] Time: 0.0000365
Optimized O(n): [305, 203, 101, 407] Time: 0.0000225
```

The AI assistant provides a detailed explanation of the performance improvement, suggesting the use of sets or dictionaries for the optimized approach. It also includes a comparison of execution behavior conceptually for large input sizes.

```
Welcome AAC A 12.4.py
C: > Users > shash > Downloads > AAC A 12.4.py > ...

19
20 user_ids = [101, 203, 305, 101, 407, 203, 509, 305, 611, 407]
21 print("User IDs:", user_ids)
22
23 start = time.perf_counter()
24 result1 = find_duplicates_brute(user_ids)
25 t1 = time.perf_counter() - start
26 print(f"\nBrute Force O(n^2): {result1} Time: {t1:.6f}s")
27
28 start = time.perf_counter()
29 result2 = find_duplicates_optimized(user_ids)
30 t2 = time.perf_counter() - start
31 print(f"Optimized O(n): {result2} Time: {t2:.6f}s")
32 large = random.choices(range(1, 5001), k=10000)
33 start = time.perf_counter()
34 find_duplicates_brute(large)
35 t1 = time.perf_counter() - start
36 start = time.perf_counter()
37 find_duplicates_optimized(large)
38 t2 = time.perf_counter() - start
39 print(f"\nLarge dataset (10000 IDs):")
40 print(f"Brute Force: {t1:.4f}s")
41 print(f"Optimized: {t2:.4f}s")

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\shash\Downloads> c:\, cd c:\Users\shash\Downloads, & c:\Users\shash\anac
e\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '5636
● PS C:\Users\shash\Downloads> c:: cd 'c:\Users\shash\Downloads'; & 'c:\Users\shash\anac
e\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '6420
User IDs: [101, 203, 305, 101, 407, 203, 509, 305, 611, 407]

Brute Force O(n^2): [101, 203, 305, 407] Time: 0.000036s
Optimized O(n): [305, 203, 101, 407] Time: 0.000022s

Large dataset (10000 IDs):
Brute Force: 3.2178s
Optimized: 0.0009s
PS C:\Users\shash\Downloads> 
```

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