SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE	DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

NAME:ACHINA MANASA **ENROLL NO.:2403A53043** **BATCH NO.:24BTCAICYB02**	**Assignment Type: Lab**	**Academic Year:**2025-2026

Course Coordinator Name	Venkataramana Veeramsetty
Instructor(s) Name	Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammand Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)

Course Code	24CS002PC215	**Course Title**	AI Assisted Coding
Year/Sem	II/I	**Regulation**	R24
Date and Day of Assignment	Week6 - Thursday	**Time(s)**	
Duration	2 Hours	**Applicable to Batches**	

AssignmentNumber:11.1(Present assignment number)/**24**(Total number of assignments)

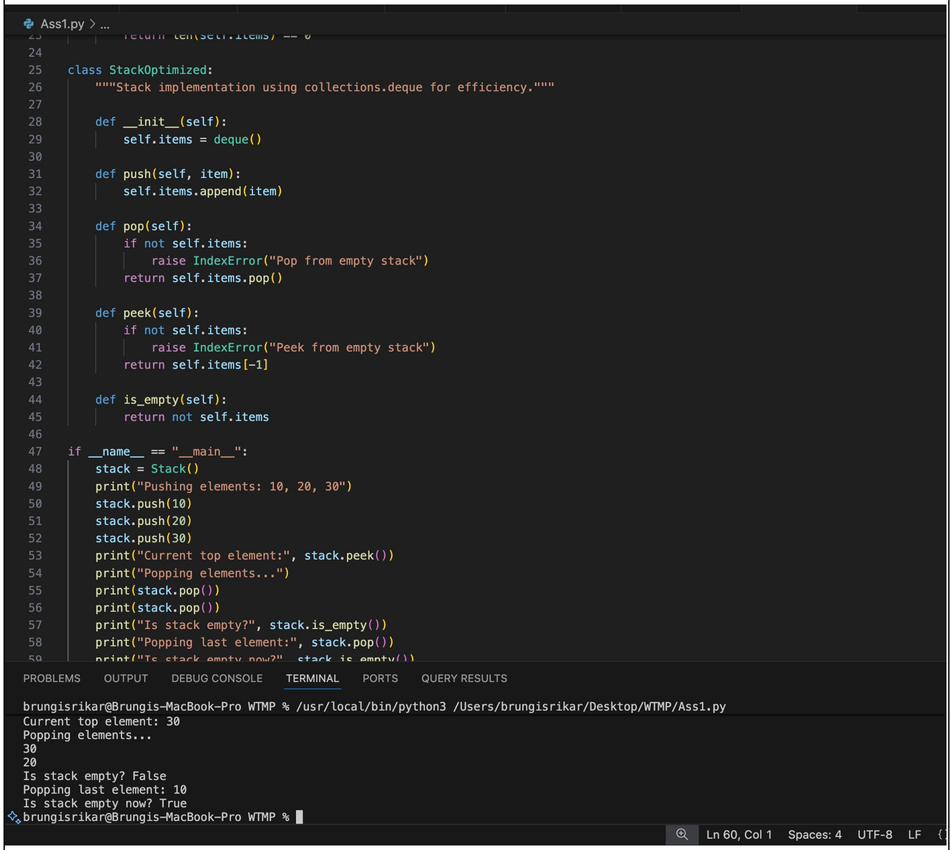
Q.No.	**Question**	***Expected Time to complete***
1	**Lab 11 – Data Structures with AI: Implementing Fundamental Structures** **Lab Objectives** • Use AI to assist in designing and implementing fundamental data structures in Python. • Learn how to prompt AI for structure creation, optimization, and documentation. • Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees,	Week6 - Thursday

Graphs, and Hash Tables.

- Enhance code quality with AI-generated comments and performance suggestions.

Task 1: Implementing a Stack (LIFO)

- **Task**: Use AI to help implement a **Stack** class in Python with the following operations: push(), pop(), peek(), and is_empty().
- **Instructions**:
  - Ask AI to generate code skeleton with docstrings.
  - Test stack operations using sample data.
  - Request AI to suggest optimizations or alternative implementations (e.g., using collections.deque).
- **Expected Output**:
  - A working Stack class with proper methods, Google-style docstrings, and inline comments for tricky parts.



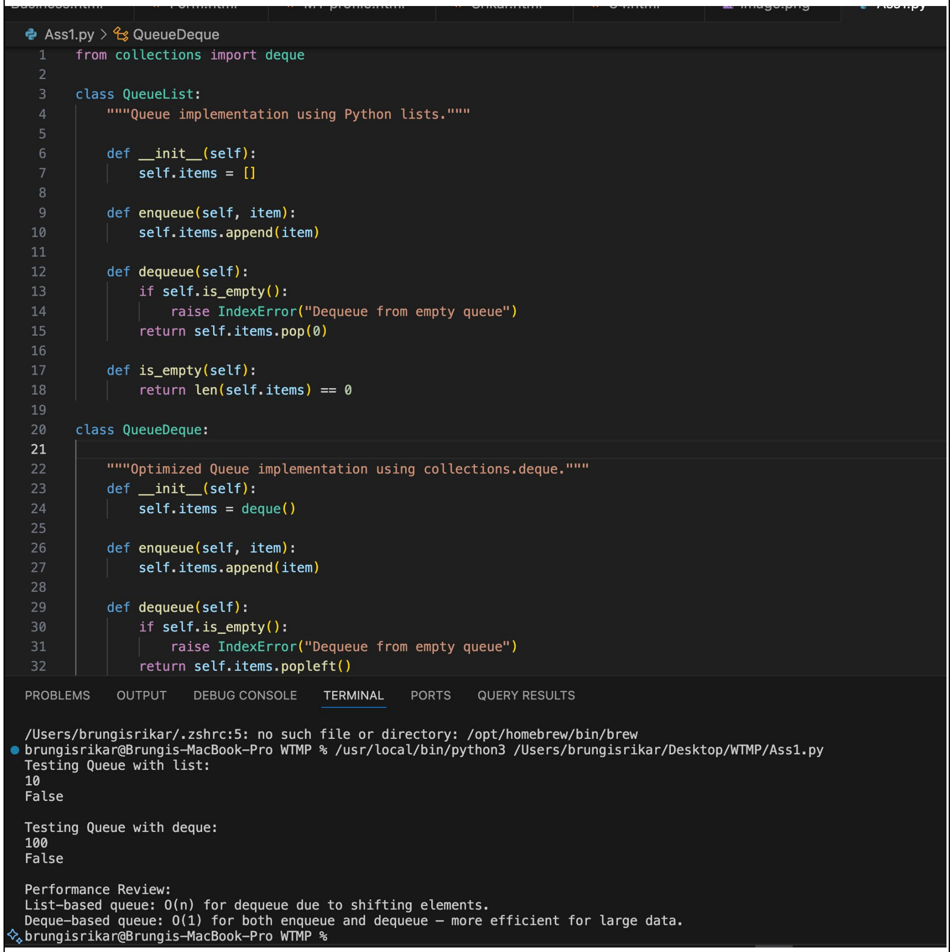
```python
        return len(self.items) == 0

class StackOptimized:
    """Stack implementation using collections.deque for efficiency."""

    def __init__(self):
        self.items = deque()

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.items:
            raise IndexError("Pop from empty stack")
        return self.items.pop()

    def peek(self):
        if not self.items:
            raise IndexError("Peek from empty stack")
        return self.items[-1]

    def is_empty(self):
        return not self.items

if __name__ == "__main__":
    stack = Stack()
    print("Pushing elements: 10, 20, 30")
    stack.push(10)
    stack.push(20)
    stack.push(30)
    print("Current top element:", stack.peek())
    print("Popping elements...")
    print(stack.pop())
    print(stack.pop())
    print("Is stack empty?", stack.is_empty())
    print("Popping last element:", stack.pop())
    print("Is stack empty now?", stack.is_empty())
```

```
brungisrikar@Brungis-MacBook-Pro WTMP % /usr/local/bin/python3 /Users/brungisrikar/Desktop/WTMP/Ass1.py
Current top element: 30
Popping elements...
30
20
Is stack empty? False
Popping last element: 10
Is stack empty now? True
brungisrikar@Brungis-MacBook-Pro WTMP %
```

Task 2: Queue Implementation with Performance Review

- **Task**: Implement a **Queue** with enqueue(), dequeue(), and is_empty() methods.
- **Instructions**:
  - First, implement using Python lists.

   - Then, ask AI to review performance and suggest a more efficient implementation (using collections.deque).
- **Expected Output**:
   - Two versions of a queue: one with lists and one optimized with deque, plus an AI-generated performance comparison.

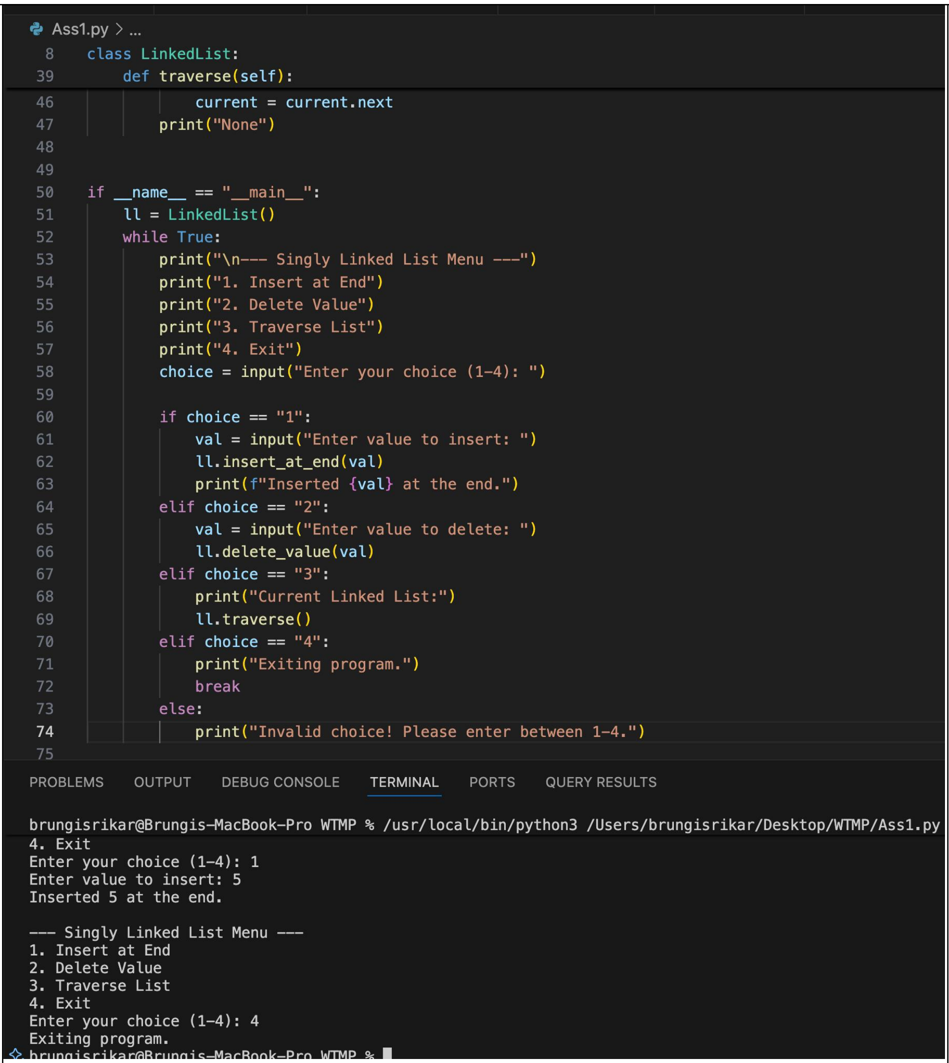


```python
from collections import deque

class QueueList:
    """Queue implementation using Python lists."""

    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)

    def is_empty(self):
        return len(self.items) == 0

class QueueDeque:

    """Optimized Queue implementation using collections.deque."""
    def __init__(self):
        self.items = deque()

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.popleft()
```

```
/Users/brungisrikar/.zshrc:5: no such file or directory: /opt/homebrew/bin/brew
brungisrikar@Brungis-MacBook-Pro WTMP % /usr/local/bin/python3 /Users/brungisrikar/Desktop/WTMP/Ass1.py
Testing Queue with list:
10
False

Testing Queue with deque:
100
False

Performance Review:
List-based queue: O(n) for dequeue due to shifting elements.
Deque-based queue: O(1) for both enqueue and dequeue – more efficient for large data.
brungisrikar@Brungis-MacBook-Pro WTMP %
```

Task 3: Singly Linked List with Traversal

- **Task**: Implement a **Singly Linked List** with operations: insert_at_end(), delete_value(), and traverse().
- **Instructions**:
   - Start with a simple class-based implementation (Node, LinkedList).
   - Use AI to generate inline comments explaining pointer updates (which are non-trivial).
   - Ask AI to suggest test cases to validate all operations.
- **Expected Output**:
   - A functional linked list implementation with clear comments explaining the logic of insertions and deletions.

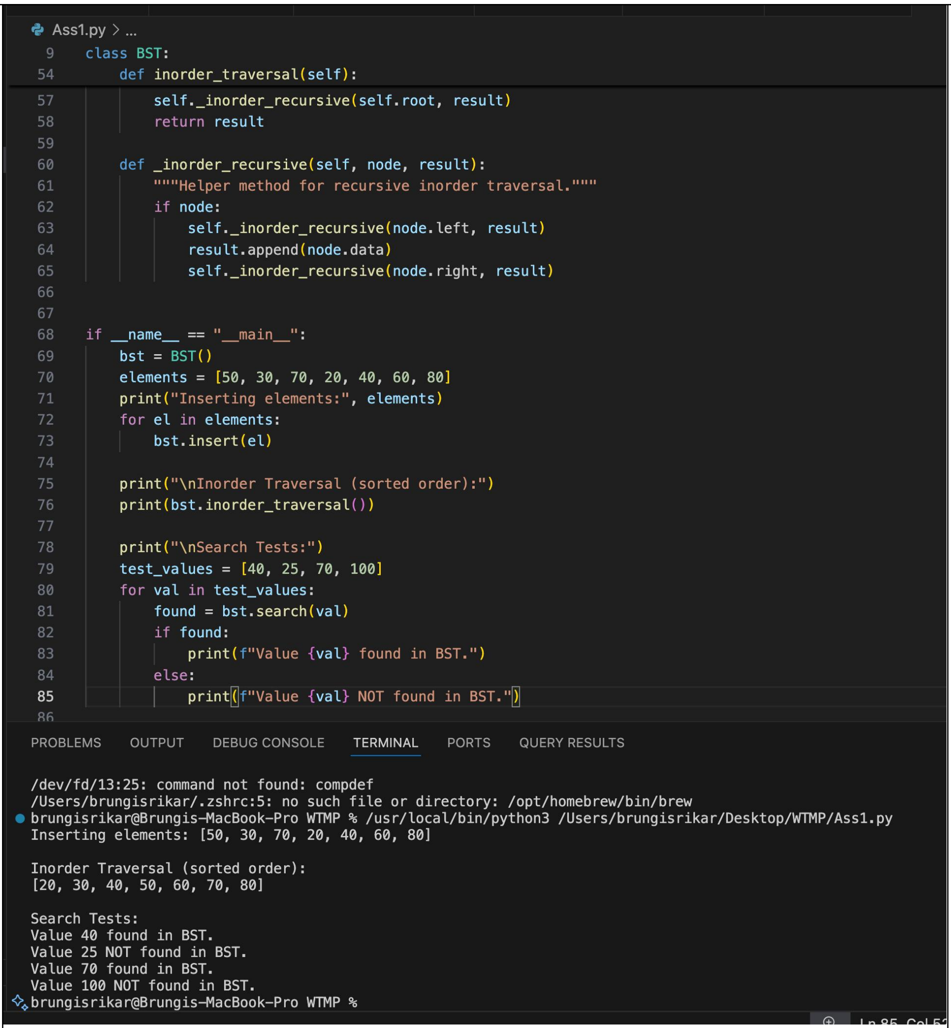


```python
class LinkedList:
    def traverse(self):
            current = current.next
        print("None")


if __name__ == "__main__":
    ll = LinkedList()
    while True:
        print("\n--- Singly Linked List Menu ---")
        print("1. Insert at End")
        print("2. Delete Value")
        print("3. Traverse List")
        print("4. Exit")
        choice = input("Enter your choice (1-4): ")

        if choice == "1":
            val = input("Enter value to insert: ")
            ll.insert_at_end(val)
            print(f"Inserted {val} at the end.")
        elif choice == "2":
            val = input("Enter value to delete: ")
            ll.delete_value(val)
        elif choice == "3":
            print("Current Linked List:")
            ll.traverse()
        elif choice == "4":
            print("Exiting program.")
            break
        else:
            print("Invalid choice! Please enter between 1-4.")

```

```
brungisrikar@Brungis-MacBook-Pro WTMP % /usr/local/bin/python3 /Users/brungisrikar/Desktop/WTMP/Ass1.py
4. Exit
Enter your choice (1-4): 1
Enter value to insert: 5
Inserted 5 at the end.

--- Singly Linked List Menu ---
1. Insert at End
2. Delete Value
3. Traverse List
4. Exit
Enter your choice (1-4): 4
Exiting program.
brungisrikar@Brungis-MacBook-Pro WTMP % 
```

Task 4: Binary Search Tree (BST)

- **Task**: Implement a **Binary Search Tree** with methods for insert(), search(), and inorder_traversal().
- **Instructions**:
  - Provide AI with a partially written Node and BST class.
  - Ask AI to complete missing methods and add docstrings.
  - Test with a list of integers and compare outputs of search() for present vs absent elements.
- **Expected Output**:
  - A BST class with clean implementation, meaningful docstrings, and correct traversal output.

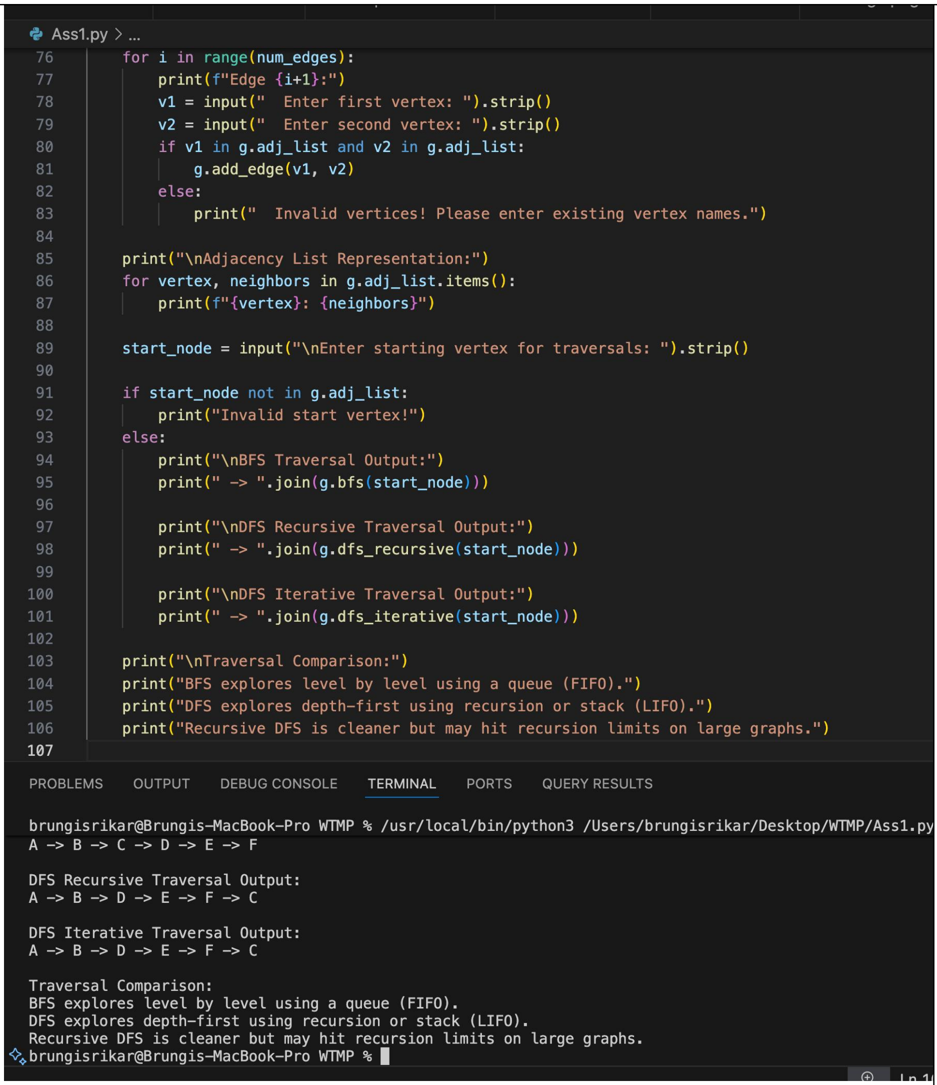


```python
class BST:
    def inorder_traversal(self):
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        """Helper method for recursive inorder traversal."""
        if node:
            self._inorder_recursive(node.left, result)
            result.append(node.data)
            self._inorder_recursive(node.right, result)


if __name__ == "__main__":
    bst = BST()
    elements = [50, 30, 70, 20, 40, 60, 80]
    print("Inserting elements:", elements)
    for el in elements:
        bst.insert(el)

    print("\nInorder Traversal (sorted order):")
    print(bst.inorder_traversal())

    print("\nSearch Tests:")
    test_values = [40, 25, 70, 100]
    for val in test_values:
        found = bst.search(val)
        if found:
            print(f"Value {val} found in BST.")
        else:
            print(f"Value {val} NOT found in BST.")
```

```
/dev/fd/13:25: command not found: compdef
/Users/brungisrikar/.zshrc:5: no such file or directory: /opt/homebrew/bin/brew
brungisrikar@Brungis-MacBook-Pro WTMP % /usr/local/bin/python3 /Users/brungisrikar/Desktop/WTMP/Ass1.py
Inserting elements: [50, 30, 70, 20, 40, 60, 80]

Inorder Traversal (sorted order):
[20, 30, 40, 50, 60, 70, 80]

Search Tests:
Value 40 found in BST.
Value 25 NOT found in BST.
Value 70 found in BST.
Value 100 NOT found in BST.
brungisrikar@Brungis-MacBook-Pro WTMP %
```

Task 5: Graph Representation and BFS/DFS Traversal

- **Task**: Implement a **Graph** using an adjacency list, with traversal methods BFS() and DFS().
- **Instructions**:
  - Start with an adjacency list dictionary.
  - Ask AI to generate BFS and DFS implementations with inline comments.
  - Compare recursive vs iterative DFS if suggested by AI.
- **Expected Output**:
  - A graph implementation with BFS and DFS traversal methods, with AI-generated comments explaining traversal steps.



```python
    for i in range(num_edges):
        print(f"Edge {i+1}:")
        v1 = input("  Enter first vertex: ").strip()
        v2 = input("  Enter second vertex: ").strip()
        if v1 in g.adj_list and v2 in g.adj_list:
            g.add_edge(v1, v2)
        else:
            print("  Invalid vertices! Please enter existing vertex names.")

    print("\nAdjacency List Representation:")
    for vertex, neighbors in g.adj_list.items():
        print(f"{vertex}: {neighbors}")

    start_node = input("\nEnter starting vertex for traversals: ").strip()

    if start_node not in g.adj_list:
        print("Invalid start vertex!")
    else:
        print("\nBFS Traversal Output:")
        print(" -> ".join(g.bfs(start_node)))

        print("\nDFS Recursive Traversal Output:")
        print(" -> ".join(g.dfs_recursive(start_node)))

        print("\nDFS Iterative Traversal Output:")
        print(" -> ".join(g.dfs_iterative(start_node)))

    print("\nTraversal Comparison:")
    print("BFS explores level by level using a queue (FIFO).")
    print("DFS explores depth-first using recursion or stack (LIFO).")
    print("Recursive DFS is cleaner but may hit recursion limits on large graphs.")

```

```
brungisrikar@Brungis-MacBook-Pro WTMP % /usr/local/bin/python3 /Users/brungisrikar/Desktop/WTMP/Ass1.py
A -> B -> C -> D -> E -> F

DFS Recursive Traversal Output:
A -> B -> D -> E -> F -> C

DFS Iterative Traversal Output:
A -> B -> D -> E -> F -> C

Traversal Comparison:
BFS explores level by level using a queue (FIFO).
DFS explores depth-first using recursion or stack (LIFO).
Recursive DFS is cleaner but may hit recursion limits on large graphs.
brungisrikar@Brungis-MacBook-Pro WTMP %
```