In [ ]:

```python
# Stack Implementation using a Fixed-Size Array
class Stack:
    """
    Implements a Stack using a fixed-size Python list.
    """
    def __init__(self, capacity):
        """Initializes the stack structure."""
        self.capacity = capacity
        # The stack is a list pre-allocated to the given capacity.
        self.stack = [None] * capacity
        # 'top' pointer: -1 indicates an empty stack.
        self.top = -1

    def is_empty(self):
        """Checks if the stack contains no elements."""
        return self.top == -1

    def is_full(self):
        """Checks if the stack has reached its maximum capacity."""
        return self.top == self.capacity - 1

    def push(self, item):
        """
        Adds an item to the top of the stack.
        Operation: Increment 'top', then store 'item' at stack[top].
        """
        if self.is_full():
            print("Stack Overflow, cannot push:", item)
        else:
            self.top += 1
            self.stack[self.top] = item
            print(f"'{item}' pushed into stack.")

    def pop(self):
        """
        Removes and returns the item from the top of the stack (LIFO).
        Operation: Retrieve stack[top], then decrement 'top'.
        """
        if self.is_empty():
            print("Stack Underflow, no element to pop.")
            return None
        else:
            popped_item = self.stack[self.top]
            # Optional: Set the array slot to None (good practice for memor
            self.stack[self.top] = None
            self.top -= 1
            print("Popped element:", popped_item)
            return popped_item
```

```python
    def display(self):
        """Prints the stack elements from top to bottom."""
        if self.is_empty():
            print("Stack is empty.")
        else:
            print("Stack elements (Top to Bottom):")
            for i in range(self.top, -1, -1):
                print(self.stack[i])

if __name__ == "__main__":
    try:
        capacity = int(input("Enter the size of the stack: "))
    except ValueError:
        print("Invalid input. Using default capacity of 5.")
        capacity = 5

    s = Stack(capacity)

    while True:
        print("\n--- Stack Operations Menu ---")
        print("1. Push")
        print("2. Pop")
        print("3. Display")
        print("4. Exit")

        try:
            choice = input("Enter your choice: ")
            if not choice.isdigit():
                raise ValueError
            choice = int(choice)
        except ValueError:
            print("Invalid input. Please enter a number between 1 and 4.")
            continue

        if choice == 1:
            item = input("Enter the element to push: ")
            s.push(item)
        elif choice == 2:
            s.pop()
        elif choice == 3:
            s.display()
        elif choice == 4:
            print("Exiting program.")
            break
        else:
            print("Invalid choice (1-4).")
```

Stack Implementation using Array (Fixed Capacity)

Data Structure: Array S, Integer variable TOP, Integer constant $\mathrm{CAPACITY}$

Initial State: $\mathrm{TOP} = -1$. Array S of size $\mathrm{CAPACITY}$ is allocated.

Auxiliary Procedures (Complexity $O(1)$)

Procedure IS_EMPTY()

If $\text{TOP} = -1$

Return True

Else

Return False

Procedure IS_FULL()

If $\text{TOP} = \text{CAPACITY} - 1$

Return True

Else

Return False

Core Operations (Complexity $O(1)$)

Procedure PUSH(ITEM)

If $\text{IS\_FULL}()$

Output: "Stack Overflow"

Else

$\text{TOP} \leftarrow \text{TOP} + 1$

$S[\text{TOP}] \leftarrow \text{ITEM}$

Output: ITEM pushed.

Procedure POP()

If $\text{IS\_EMPTY}()$

Output: "Stack Underflow"

Return NULL

Else

$\text{POPPED\_ITEM} \leftarrow S[\text{TOP}]$

$S[\text{TOP}] \leftarrow \text{NULL}$ (Optional: Clear array slot)

$\text{TOP} \leftarrow \text{TOP} - 1$

Output: "Popped element:" POPPED_ITEM

Return POPPED_ITEM

Utility Operation (Complexity $O(N)$)

Procedure DISPLAY()

If $\text{IS\_EMPTY}()$

Output: "Stack is empty."

Else

Output: "Stack elements (Top to Bottom):"

For i from TOP down to 0 do

Output $S[i]$

End For

Main Driver Procedure

Procedure MAIN()

Input CAPACITY.

Call $\text{Stack.Initialize}(\text{CAPACITY})$ (Sets up S and $\text{TOP} \leftarrow -1$).

Loop Forever:

Display Menu Options (Push, Pop, Display, Exit).

Input CHOICE.

If $\text{CHOICE} = 1$:

Input ITEM.

Call $\text{PUSH}(\text{ITEM})$.

Else If $\text{CHOICE} = 2$:

Call $\text{POP}()$.

Else If $\text{CHOICE} = 3$:

Call $\text{DISPLAY}()$.

Else If $\text{CHOICE} = 4$:

Output: "Exiting program."

Break Loop.

Else:

Output: "Invalid choice."

End If

End Loop

PRIMS

In [ ]:

```
# Prim's Algorithm Implementation (Min-Heap + Adjacency List)
# No external libraries used.

class MinHeap:
```

```
    """
    Helper class to implement a Min-Heap from scratch.
    Used to prioritize edges with the smallest weight.
    """
    def __init__(self):
        self.heap = []

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

    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def push(self, item):
        """
        Inserts an item (weight, vertex) and maintains heap property (Sift
        """
        self.heap.append(item)
        current = len(self.heap) - 1

        # Bubble up while current is smaller than parent
        while current > 0 and self.heap[current][0] < self.heap[self.parent
            self.swap(current, self.parent(current))
            current = self.parent(current)

    def pop(self):
        """
        Removes and returns the smallest item (root) and maintains heap pro
        """
        if self.is_empty():
            return None

        if len(self.heap) == 1:
            return self.heap.pop()

        root = self.heap[0]
        # Move last element to root
        self.heap[0] = self.heap.pop()
        self.heapify_down(0)
        return root

    def heapify_down(self, i):
```

```
        """Recursive function to fix the heap downwards."""
        smallest = i
        left = self.left_child(i)
        right = self.right_child(i)
        size = len(self.heap)

        # Check if left child exists and is smaller than root
        if left < size and self.heap[left][0] < self.heap[smallest][0]:
            smallest = left

        # Check if right child exists and is smaller than smallest so far
        if right < size and self.heap[right][0] < self.heap[smallest][0]:
            smallest = right

        # If smallest is not root, swap and continue
        if smallest != i:
            self.swap(i, smallest)
            self.heapify_down(smallest)


class Graph:
    """
    Implements Prim's Algorithm using Adjacency List and the custom MinHeap
    """
    def __init__(self, vertices):
        self.V = vertices
        # Adjacency list: dictionary where key=vertex, value=list of (neigh
        self.adj = {i: [] for i in range(vertices)}

    def add_edge(self, u, v, weight):
        """Adds an undirected edge between u and v."""
        self.adj[u].append((v, weight))
        self.adj[v].append((u, weight))  # Undirected Graph

    def prim_mst(self, start_node):
        """
        Executes Prim's Algorithm to find the Minimum Spanning Tree.
        """
        # Priority Queue to store (weight, vertex)
        min_heap = MinHeap()

        # Array to keep track of visited vertices
        visited = [False] * self.V

        mst_cost = 0
        edges_count = 0

        print(f"\nRunning Prim's Algorithm starting from vertex {start_node
        print("Selected Edges (Weight, Target Vertex):")

        # Initial Step: Push start_node with weight 0
```

```python
        min_heap.push((0, start_node))

        while not min_heap.is_empty():
            # Extract vertex with minimum weight
            weight, u = min_heap.pop()

            # If vertex is already included in MST, skip it
            if visited[u]:
                continue

            # Include vertex in MST
            visited[u] = True
            mst_cost += weight

            if weight != 0:
                print(f" - Added edge to vertex {u} with weight {weight}")
                edges_count += 1

            # Iterate over adjacent vertices
            for v, w in self.adj[u]:
                if not visited[v]:
                    min_heap.push((w, v))

        print("-" * 30)
        print(f"Total Minimum Spanning Tree Cost: {mst_cost}")
        print(f"Total edges in MST: {edges_count}")


# Main Driver Code
if __name__ == "__main__":
    print("--- Prim's Algorithm (Min-Heap + Adjacency List) ---")
    try:
        v_count = int(input("Enter number of vertices (0 to N-1): "))
        g = Graph(v_count)

        e_count = int(input("Enter number of edges: "))
        print("Enter edges in format: Source Destination Weight")
        for _ in range(e_count):
            u, v, w = map(int, input().split())
            g.add_edge(u, v, w)

        start = int(input("Enter start vertex: "))
        g.prim_mst(start)

    except ValueError:
        print("Invalid input! Please enter integers only.")
```

```
--- Prim's Algorithm (Min-Heap + Adjacency List) ---
Enter number of vertices (0 to N-1): 6
Enter number of edges: 4
Enter edges in format: Source Destination Weight
1 2 3
2 3 4
3 1 4
```

```
2 5 2
Enter start vertex: 2

Running Prim's Algorithm starting from vertex 2...
Selected Edges (Weight, Target Vertex):
 - Added edge to vertex 5 with weight 2
 - Added edge to vertex 1 with weight 3
 - Added edge to vertex 3 with weight 4
------------------------------
Total Minimum Spanning Tree Cost: 9
Total edges in MST: 3
```

```
Algorithm: PRIM_MST_WITH_MIN_HEAP

Data Structures
    H = []                        // Min-Heap Array (stores tuples of weight, vertex)
    adj = {i: [] for i in range(V)}   // Adjacency List
    visited = [False] * V         // Visited Array
    mst_cost = 0                  // Total Cost of MST
    edges_count = 0               // Number of edges in MST

Procedure IS_EMPTY()
    return len(H) == 0

Procedure PARENT(i)
    return (i - 1) // 2

Procedure LEFT_CHILD(i)
    return 2 * i + 1

Procedure RIGHT_CHILD(i)
    return 2 * i + 2

Procedure SWAP(i, j)
    temp = H[i]
    H[i] = H[j]
    H[j] = temp

Procedure HEAPIFY_DOWN(i)
    smallest = i
    left = LEFT_CHILD(i)
    right = RIGHT_CHILD(i)
    size = len(H)

    if left < size and H[left][0] < H[smallest][0]:
        smallest = left

    if right < size and H[right][0] < H[smallest][0]:
        smallest = right

    if smallest != i:
        SWAP(i, smallest)
        HEAPIFY_DOWN(smallest)

Procedure PUSH(item)
    H.append(item)
    current = len(H) - 1

    while current > 0 and H[current][0] < H[PARENT(current)][0]:
        SWAP(current, PARENT(current))
        current = PARENT(current)

Procedure POP()
    if IS_EMPTY():
        return None

    if len(H) == 1:
        return H.pop()

    root = H[0]
    H[0] = H.pop()
    HEAPIFY_DOWN(0)
    return root

Procedure ADD_EDGE(u, v, weight)
    adj[u].append((v, weight))
    adj[v].append((u, weight))

Procedure PRIM_MST(start_node)
```

```
    visited = [False] * V
    mst_cost = 0
    edges_count = 0

    PUSH((0, start_node))

    while not IS_EMPTY():
        weight, u = POP()

        if visited[u]:
            continue

        visited[u] = True
        mst_cost = mst_cost + weight

        if weight != 0:
            edges_count = edges_count + 1

        for (v, w) in adj[u]:
            if not visited[v]:
                PUSH((w, v))

    print("Total Minimum Spanning Tree Cost:", mst_cost)
    print("Total edges in MST:", edges_count)

Procedure MAIN()
    input V
    adj = {i: [] for i in range(V)}
    input E
    for i in range(E):
        input u, v, w
        ADD_EDGE(u, v, w)
    input start
    PRIM_MST(start)
```

KRUSKAL

In [ ]:

```
# Kruskal's Algorithm Implementation (Min-Heap + Adjacency List + Union-Fin
# No external libraries used.

class MinHeap:
    """
    Helper class to implement a Min-Heap from scratch.
    Stores tuples in format: (weight, source, destination)
    """
    def __init__(self):
        self.heap = []

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

    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
```

```
    def push(self, item):
        """
        Inserts an item (weight, u, v) and maintains heap property.
        """
        self.heap.append(item)
        current = len(self.heap) - 1

        # Bubble up based on weight (item[0])
        while current > 0 and self.heap[current][0] < self.heap[self.parent
            self.swap(current, self.parent(current))
            current = self.parent(current)

    def pop(self):
        """
        Removes and returns the edge with the smallest weight.
        """
        if self.is_empty():
            return None

        if len(self.heap) == 1:
            return self.heap.pop()

        root = self.heap[0]
        self.heap[0] = self.heap.pop()
        self.heapify_down(0)
        return root

    def heapify_down(self, i):
        smallest = i
        left = self.left_child(i)
        right = self.right_child(i)
        size = len(self.heap)

        if left < size and self.heap[left][0] < self.heap[smallest][0]:
            smallest = left

        if right < size and self.heap[right][0] < self.heap[smallest][0]:
            smallest = right

        if smallest != i:
            self.swap(i, smallest)
            self.heapify_down(smallest)


class DisjointSet:
    """
    Helper class for Union-Find data structure.
    Used to detect cycles in Kruskal's algorithm.
    """
    def __init__(self, n):
```

```
        # Initially, every vertex is its own parent
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, i):
        """Finds the representative of the set containing i (with path comp
        if self.parent[i] != i:
            self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    def union(self, i, j):
        """Unions the sets containing i and j. Returns True if union was su
        root_i = self.find(i)
        root_j = self.find(j)

        if root_i != root_j:
            # Union by Rank
            if self.rank[root_i] < self.rank[root_j]:
                self.parent[root_i] = root_j
            elif self.rank[root_i] > self.rank[root_j]:
                self.parent[root_j] = root_i
            else:
                self.parent[root_j] = root_i
                self.rank[root_i] += 1
            return True # Successfully merged (no cycle)
        return False # Cycle detected


class Graph:
    """
    Implements Kruskal's Algorithm using Adjacency List, MinHeap, and Union
    """
    def __init__(self, vertices):
        self.V = vertices
        self.adj = {i: [] for i in range(vertices)}

    def add_edge(self, u, v, weight):
        """Adds an undirected edge."""
        # Note: We check bounds here to prevent KeyError
        if 0 <= u < self.V and 0 <= v < self.V:
            self.adj[u].append((v, weight))
            self.adj[v].append((u, weight))
        else:
            print(f"Error: Vertices {u} or {v} are out of bounds (0 to {sel

    def kruskal_mst(self):
        """
        Executes Kruskal's Algorithm.
        1. Push all edges to MinHeap.
        2. Pop edges and add to MST if they don't form a cycle.
        """
```

```
        min_heap = MinHeap()
        ds = DisjointSet(self.V)

        mst_cost = 0
        edges_count = 0

        # Step 1: Convert Adjacency List to Min-Heap of edges
        # We iterate through all vertices to find edges.
        # To avoid adding the same undirected edge twice (u-v and v-u), we
        for u in range(self.V):
            for v, w in self.adj[u]:
                if u < v:
                    min_heap.push((w, u, v))

        print("\nRunning Kruskal's Algorithm...")
        print("Selected Edges (Weight, Source - Destination):")

        # Step 2: Process edges from Min-Heap
        while not min_heap.is_empty() and edges_count < self.V - 1:
            weight, u, v = min_heap.pop()

            # Step 3: Check if u and v are in different sets (Cycle Detecti
            if ds.union(u, v):
                print(f" - Edge included: {u} -- {v} (Weight: {weight})")
                mst_cost += weight
                edges_count += 1
            # Else: The edge forms a cycle, so we discard it (do nothing)

        print("-" * 30)
        print(f"Total Minimum Spanning Tree Cost: {mst_cost}")
        print(f"Total edges in MST: {edges_count}")


if __name__ == "__main__":
    print("--- Kruskal's Algorithm (Min-Heap + Adjacency List) ---")
    try:
        v_count = int(input("Enter number of vertices (0 to N-1): "))
        g = Graph(v_count)

        e_count = int(input("Enter number of edges: "))
        print("Enter edges in format: Source Destination Weight")
        for _ in range(e_count):
            try:
                u, v, w = map(int, input().split())
                g.add_edge(u, v, w)
            except ValueError:
                print("Invalid edge input format.")

        g.kruskal_mst()
```

```
    except ValueError:
        print("Invalid input! Please enter integers only.")
```

```
Data structures
    H = []                                    // min-heap array storing (weight, u, v)
    adj = {i: [] for i in range(V)}          // adjacency list: vertex -> list of (neighbor, weight)
    parent = [0..V-1]                        // for DisjointSet initialization: parent[i] = i
    rank = [0] * V                           // DisjointSet rank array
    mst_cost = 0
    edges_count = 0

Procedure IS_EMPTY()
    return len(H) == 0

Procedure PARENT_INDEX(i)
    return (i - 1) // 2

Procedure LEFT_CHILD(i)
    return 2 * i + 1

Procedure RIGHT_CHILD(i)
    return 2 * i + 2

Procedure SWAP(i, j)
    temp = H[i]
    H[i] = H[j]
    H[j] = temp

Procedure HEAPIFY_DOWN(i)
    smallest = i
    left = LEFT_CHILD(i)
    right = RIGHT_CHILD(i)
    size = len(H)

    if left < size and H[left][0] < H[smallest][0]:
        smallest = left

    if right < size and H[right][0] < H[smallest][0]:
        smallest = right

    if smallest != i:
        SWAP(i, smallest)
        HEAPIFY_DOWN(smallest)

Procedure PUSH(item)   // item = (weight, u, v)
    H.append(item)
    current = len(H) - 1

    while current > 0 and H[current][0] < H[PARENT_INDEX(current)][0]:
        SWAP(current, PARENT_INDEX(current))
        current = PARENT_INDEX(current)

Procedure POP()
    if IS_EMPTY():
        return None

    if len(H) == 1:
        return H.pop()

    root = H[0]
    H[0] = H.pop()
    HEAPIFY_DOWN(0)
    return root

----------------------------------------
-- Disjoint Set (Union-Find) procedures
Procedure MAKE_SET(n)
    for i in range(n):
        parent[i] = i
        rank[i] = 0

Procedure FIND(i)
    if parent[i] != i:
        parent[i] = FIND(parent[i])   // path compression
    return parent[i]

Procedure UNION(i, j) -> boolean
    root_i = FIND(i)
```

```
    root_j = FIND(j)

    if root_i != root_j:
        if rank[root_i] < rank[root_j]:
            parent[root_i] = root_j
        elif rank[root_i] > rank[root_j]:
            parent[root_j] = root_i
        else:
            parent[root_j] = root_i
            rank[root_i] = rank[root_i] + 1
        return True   // merged, no cycle
    return False      // already same set -> would form cycle

----------------------------------------
-- Graph / Kruskal procedures

Procedure ADD_EDGE(u, v, weight)
    // undirected edge
    if 0 <= u < V and 0 <= v < V:
        adj[u].append((v, weight))
        adj[v].append((u, weight))

Procedure BUILD_HEAP_FROM_ADJ()
    // Push each undirected edge once (u < v)
    for u in range(V):
        for (v, w) in adj[u]:
            if u < v:
                PUSH((w, u, v))

Procedure KRUSKAL_MST()
    MAKE_SET(V)
    mst_cost = 0
    edges_count = 0

    BUILD_HEAP_FROM_ADJ()

    print("Running Kruskal's Algorithm...")
    print("Selected Edges (Weight, Source - Destination):")

    while not IS_EMPTY() and edges_count < V - 1:
        item = POP()
        if item is None:
            break
        weight, u, v = item

        if UNION(u, v):
            print(" - Edge included:", u, "--", v, "(Weight:", weight, ")")
            mst_cost = mst_cost + weight
            edges_count = edges_count + 1
        // else: edge forms cycle, discard

    print("-----------------------------")
    print("Total Minimum Spanning Tree Cost:", mst_cost)
    print("Total edges in MST:", edges_count)

----------------------------------------
Procedure MAIN()
    input V
    adj = {i: [] for i in range(V)}
    parent = [0..V-1]
    rank = [0] * V

    input E
    for i in range(E):
        input u, v, w
        ADD_EDGE(u, v, w)

    KRUSKAL_MST()
```

Dijkstra's Algorithm

In [ ]:

```
# Dijkstra's Algorithm Implementation (Min-Heap + Adjacency List)
# No external libraries used.
```

```
class MinHeap:
    """
    Helper class to implement a Min-Heap from scratch.
    Stores tuples in format: (distance, vertex)
    """
    def __init__(self):
        self.heap = []

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

    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def push(self, item):
        """
        Inserts an item (distance, vertex) and maintains heap property.
        """
        self.heap.append(item)
        current = len(self.heap) - 1

        # Bubble up based on distance (item[0])
        while current > 0 and self.heap[current][0] < self.heap[self.parent
            self.swap(current, self.parent(current))
            current = self.parent(current)

    def pop(self):
        """
        Removes and returns the node with the smallest distance.
        """
        if self.is_empty():
            return None

        if len(self.heap) == 1:
            return self.heap.pop()

        root = self.heap[0]
        self.heap[0] = self.heap.pop()
        self.heapify_down(0)
        return root

    def heapify_down(self, i):
```

```
        smallest = i
        left = self.left_child(i)
        right = self.right_child(i)
        size = len(self.heap)

        if left < size and self.heap[left][0] < self.heap[smallest][0]:
            smallest = left

        if right < size and self.heap[right][0] < self.heap[smallest][0]:
            smallest = right

        if smallest != i:
            self.swap(i, smallest)
            self.heapify_down(smallest)


class Graph:
    def __init__(self, vertices):
        self.V = vertices
        # Adjacency list: {vertex: [(neighbor, weight), ...]}
        self.adj = {i: [] for i in range(vertices)}

    def add_edge(self, u, v, weight):
        """Adds an undirected edge with bounds checking."""
        if 0 <= u < self.V and 0 <= v < self.V:
            self.adj[u].append((v, weight))
            self.adj[v].append((u, weight)) # For Undirected Graph
        else:
            print(f"Error: Ignored edge {u}-{v}. Vertices must be between 0

    def dijkstra(self, start_node):
        """
        Calculates shortest paths from start_node to all other nodes.
        """
        # Initialize distances to Infinity
        # float('inf') acts as a number larger than any other number
        distances = [float('inf')] * self.V
        distances[start_node] = 0

        min_heap = MinHeap()
        # Push start node: (distance, vertex)
        min_heap.push((0, start_node))

        print(f"\nRunning Dijkstra's Algorithm from Source: {start_node}")

        while not min_heap.is_empty():
            # Extract vertex with smallest distance so far
            current_dist, u = min_heap.pop()

            # Optimization: If the popped distance is greater than the
            # already known shortest distance, we skip processing (stale en
```

```python
            if current_dist > distances[u]:
                continue

            # Explore neighbors
            for v, weight in self.adj[u]:
                # Relaxation Step
                if distances[u] + weight < distances[v]:
                    distances[v] = distances[u] + weight
                    min_heap.push((distances[v], v))

        # Output Results
        print("-" * 40)
        print(f"{'Vertex':<10} | {'Shortest Distance from Source'}")
        print("-" * 40)
        for i in range(self.V):
            dist_display = distances[i] if distances[i] != float('inf') els
            print(f"{i:<10} | {dist_display}")


# --- Driver Code ---
if __name__ == "__main__":
    print("--- Dijkstra's Algorithm (Min-Heap + Adjacency List) ---")
    try:
        v_count = int(input("Enter number of vertices (0 to N-1): "))
        g = Graph(v_count)

        e_count = int(input("Enter number of edges: "))
        print("Enter edges in format: Source Destination Weight")
        for _ in range(e_count):
            try:
                u, v, w = map(int, input().split())
                g.add_edge(u, v, w)
            except ValueError:
                print("Invalid edge input. Please enter 3 integers.")
                continue

        start = int(input("Enter start vertex (Source): "))

        if 0 <= start < v_count:
            g.dijkstra(start)
        else:
            print(f"Error: Start vertex must be between 0 and {v_count-1}."

    except ValueError:
        print("Invalid input! Please enter integers only.")
```

```
Data structures
    H = []                                 // min-heap array storing (distance, vertex)
    adj = {i: [] for i in range(V)}       // adjacency list: vertex -> list of (neighbor, weight)
    distances = [infinity] * V            // shortest known distances from source
    INF = infinity

Procedure IS_EMPTY()
    return len(H) == 0
```

```
Procedure PARENT(i)
    return (i - 1) // 2

Procedure LEFT_CHILD(i)
    return 2 * i + 1

Procedure RIGHT_CHILD(i)
    return 2 * i + 2

Procedure SWAP(i, j)
    temp = H[i]
    H[i] = H[j]
    H[j] = temp

Procedure HEAPIFY_DOWN(i)
    smallest = i
    left = LEFT_CHILD(i)
    right = RIGHT_CHILD(i)
    size = len(H)

    if left < size and H[left][0] < H[smallest][0]:
        smallest = left

    if right < size and H[right][0] < H[smallest][0]:
        smallest = right

    if smallest != i:
        SWAP(i, smallest)
        HEAPIFY_DOWN(smallest)

Procedure PUSH(item)   // item = (distance, vertex)
    H.append(item)
    current = len(H) - 1

    while current > 0 and H[current][0] < H[PARENT(current)][0]:
        SWAP(current, PARENT(current))
        current = PARENT(current)

Procedure POP() -> (distance, vertex) or NULL
    if IS_EMPTY():
        return NULL

    if len(H) == 1:
        return H.pop()

    root = H[0]
    H[0] = H.pop()
    HEAPIFY_DOWN(0)
    return root

Procedure ADD_EDGE(u, v, weight)
    if 0 <= u < V and 0 <= v < V:
        adj[u].append((v, weight))
        adj[v].append((u, weight))   // undirected graph
    // else: ignore invalid edge

Procedure DIJKSTRA(start_node)
    // Initialize distances
    for i in range(V):
        distances[i] = INF

    distances[start_node] = 0

    // Priority queue (min-heap) seeded with source
    PUSH((0, start_node))

    while not IS_EMPTY():
        item = POP()
        if item is NULL:
            break
        current_dist, u = item

        // Skip stale heap entries: we already found a better path to u
        if current_dist > distances[u]:
            continue

        // Relax neighbors
        for (v, weight) in adj[u]:
            // If going through u gives a shorter path to v, update
```

```
            if distances[u] + weight < distances[v]:
                distances[v] = distances[u] + weight
                PUSH((distances[v], v))

    // Output results
    print("------------------------------")
    print("Vertex     | Shortest Distance from Source")
    print("------------------------------")
    for i in range(V):
        if distances[i] == INF:
            print(i, "|", "Unreachable")
        else:
            print(i, "|", distances[i])

Procedure MAIN()
    input V
    adj = {i: [] for i in range(V)}
    distances = [INF] * V

    input E
    for _ in range(E):
        input u, v, w
        ADD_EDGE(u, v, w)

    input start
    if 0 <= start < V:
        DIJKSTRA(start)
    else:
        print("Error: Start vertex out of range")
```

BFS & DFS

In [ ]:

```
# BFS and DFS Implementation (Queue and Stack)
# No external libraries used.

class Stack:
    """
    LIFO Data Structure for DFS.
    """
    def __init__(self):
        self.items = []

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

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

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return None


class Queue:
    """
    FIFO Data Structure for BFS.
    """
    def __init__(self):
        self.items = []
```

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

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

    def dequeue(self):
        if not self.is_empty():
            # Removes the first element (index 0)
            return self.items.pop(0)
        return None


class Graph:
    def __init__(self, vertices):
        self.V = vertices
        # Adjacency list: {vertex: [neighbors]}
        self.adj = {i: [] for i in range(vertices)}

    def add_edge(self, u, v):
        """Adds an undirected edge with bounds checking."""
        if 0 <= u < self.V and 0 <= v < self.V:
            self.adj[u].append(v)
            self.adj[v].append(u)
        else:
            print(f"Error: Vertices {u} or {v} out of bounds.")

    def bfs(self, start_node):
        """
        Breadth-First Search using a Queue.
        """
        visited = [False] * self.V
        queue = Queue()

        # Mark source as visited and enqueue
        visited[start_node] = True
        queue.enqueue(start_node)

        print(f"BFS Traversal starting from {start_node}: ", end="")

        while not queue.is_empty():
            u = queue.dequeue()
            print(u, end=" ")

            # Get all adjacent vertices
            for v in self.adj[u]:
                if not visited[v]:
                    visited[v] = True
                    queue.enqueue(v)
        print() # Newline
```

```
    def dfs(self, start_node):
        """
        Iterative Depth-First Search using a Stack.
        """
        visited = [False] * self.V
        stack = Stack()

        # Push start node
        stack.push(start_node)

        print(f"DFS Traversal starting from {start_node}: ", end="")

        while not stack.is_empty():
            u = stack.pop()

            # In Iterative DFS, we check visited AFTER popping
            if not visited[u]:
                visited[u] = True
                print(u, end=" ")

                # Push all adjacent vertices
                # We reverse the list to visit neighbors in standard order
                # because a stack reverses the order again.
                for v in reversed(self.adj[u]):
                    if not visited[v]:
                        stack.push(v)
        print() # Newline


# --- Driver Code ---
if __name__ == "__main__":
    print("--- Graph Traversals (BFS & DFS) ---")
    try:
        v_count = int(input("Enter number of vertices (0 to N-1): "))
        g = Graph(v_count)

        e_count = int(input("Enter number of edges: "))
        print("Enter edges (u v):")
        for _ in range(e_count):
            try:
                u, v = map(int, input().split())
                g.add_edge(u, v)
            except ValueError:
                print("Invalid edge input.")

        start = int(input("Enter start vertex: "))

        if 0 <= start < v_count:
            g.bfs(start)
            g.dfs(start)
```

```
        else:
            print("Start vertex out of bounds.")

    except ValueError:
        print("Invalid input! Please enter integers only.")
```

```
Data structures
    stack_items = []                       // for Stack (LIFO)
    queue_items = []                       // for Queue (FIFO)
    adj = {i: [] for i in range(V)}        // adjacency list: vertex -> list of neighbors
    visited = [False] * V

----------------------------------------
-- Stack (LIFO) procedures
Procedure STACK_IS_EMPTY()
    return len(stack_items) == 0

Procedure STACK_PUSH(item)
    stack_items.append(item)

Procedure STACK_POP()
    if not STACK_IS_EMPTY():
        return stack_items.pop()
    return NULL

----------------------------------------
-- Queue (FIFO) procedures
Procedure QUEUE_IS_EMPTY()
    return len(queue_items) == 0

Procedure ENQUEUE(item)
    queue_items.append(item)

Procedure DEQUEUE()
    if not QUEUE_IS_EMPTY():
        return queue_items.pop(0)
    return NULL

----------------------------------------
-- Graph procedures

Procedure ADD_EDGE(u, v)
    // Undirected edge with bounds checking
    if 0 <= u < V and 0 <= v < V:
        adj[u].append(v)
        adj[v].append(u)
    // else: ignore or report out-of-bounds

Procedure BFS(start_node)
    // Breadth-First Search using Queue
    visited = [False] * V
    queue_items = []

    visited[start_node] = True
    ENQUEUE(start_node)

    print("BFS Traversal starting from", start_node, ":")

    while not QUEUE_IS_EMPTY():
        u = DEQUEUE()
        if u is NULL:
            break
        print(u, end=" ")

        for v in adj[u]:
            if not visited[v]:
                visited[v] = True
                ENQUEUE(v)
    print()  // newline after traversal

Procedure DFS(start_node)
    // Iterative Depth-First Search using Stack
    visited = [False] * V
    stack_items = []

    STACK_PUSH(start_node)
```

```
    print("DFS Traversal starting from", start_node, ":")

    while not STACK_IS_EMPTY():
        u = STACK_POP()
        if u is NULL:
            break

        // Check visited after popping (iterative DFS pattern)
        if not visited[u]:
            visited[u] = True
            print(u, end=" ")

            // Push neighbors in reverse order so they are visited in original order
            for v in reversed(adj[u]):
                if not visited[v]:
                    STACK_PUSH(v)
    print()  // newline after traversal

----------------------------------------
Procedure MAIN()
    input V
    adj = {i: [] for i in range(V)}

    input E
    for i in range(E):
        input u, v
        ADD_EDGE(u, v)

    input start
    if 0 <= start < V:
        BFS(start)
        DFS(start)
    else:
        print("Start vertex out of bounds")
```

FLOYD WARSHALL

In [ ]:

```
# Floyd-Warshall Algorithm Implementation (Adjacency Matrix)
# No external libraries used.

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        # Initialize matrix with Infinity
        # float('inf') represents no direct path
        self.matrix = [[float('inf')] * vertices for _ in range(vertices)]

        # Distance from a node to itself is always 0
        for i in range(vertices):
            self.matrix[i][i] = 0

    def add_edge(self, u, v, weight):
        """
        Adds a directed edge.
        For undirected, you would add matrix[v][u] = weight as well.
        """
        if 0 <= u < self.V and 0 <= v < self.V:
            self.matrix[u][v] = weight
        else:
            print(f"Error: Vertices {u} or {v} out of bounds.")
```

```
    def floyd_warshall(self):
        """
        Computes shortest paths between all pairs of vertices.
        """
        # Create a copy of the matrix to store solution (dist)
        # We use a list comprehension to deep copy the rows
        dist = [row[:] for row in self.matrix]

        print("\nRunning Floyd-Warshall Algorithm...")

        # Core Logic: 3 Nested Loops
        # k = intermediate vertex
        # i = source vertex
        # j = destination vertex
        for k in range(self.V):
            for i in range(self.V):
                for j in range(self.V):

                    # If vertex k is on the shortest path from i to j,
                    # then update the value of dist[i][j]

                    # Check if paths through k exist (avoid inf + something
                    if dist[i][k] != float('inf') and dist[k][j] != float('
                        if dist[i][k] + dist[k][j] < dist[i][j]:
                            dist[i][j] = dist[i][k] + dist[k][j]

        self.display_solution(dist)

    def display_solution(self, dist):
        """
        Prints the final shortest distance matrix.
        """
        print("-" * 40)
        print("Shortest distances between every pair of vertices:")
        print("-" * 40)

        # Header
        print("      ", end="")
        for i in range(self.V):
            print(f"{i:4}", end="")
        print("\n")

        for i in range(self.V):
            print(f"{i:<4}| ", end="")
            for j in range(self.V):
                if dist[i][j] == float('inf'):
                    print(" INF", end="")
                else:
                    print(f"{dist[i][j]:4}", end="")
            print() # Newline for next row
```

```python
# --- Driver Code ---
if __name__ == "__main__":
    print("--- Floyd-Warshall Algorithm (Adjacency Matrix) ---")
    try:
        v_count = int(input("Enter number of vertices (0 to N-1): "))
        g = Graph(v_count)

        e_count = int(input("Enter number of edges: "))
        print("Enter edges in format: Source Destination Weight")
        print("(Note: This algorithm handles Directed Edges)")

        for _ in range(e_count):
            try:
                u, v, w = map(int, input().split())
                g.add_edge(u, v, w)
            except ValueError:
                print("Invalid input. Skipping edge.")

        g.floyd_warshall()

    except ValueError:
        print("Invalid input! Please enter integers only.")
```

```
Data structures
    V = number of vertices
    INF = infinity
    matrix = [[INF for j in range(V)] for i in range(V)]   // adjacency matrix
    for i in range(V):
        matrix[i][i] = 0                                    // distance to self = 0

Procedure ADD_EDGE(u, v, weight)
    // Directed edge from u to v
    if 0 <= u < V and 0 <= v < V:
        matrix[u][v] = weight
    // else: ignore or report out-of-bounds

Procedure COPY_MATRIX(src) -> dst
    // Deep copy rows
    dst = [row[:] for row in src]
    return dst

Procedure DISPLAY_SOLUTION(dist)
    print("------------------------------")
    print("Shortest distances between every pair of vertices:")
    print("------------------------------")
    // Header
    print("      ", end="")
    for i in range(V):
        print(f"{i:4}", end="")
    print("\n")
    for i in range(V):
        print(f"{i:<4}| ", end="")
        for j in range(V):
            if dist[i][j] == INF:
                print(" INF", end="")
            else:
                print(f"{dist[i][j]:4}", end="")
        print()    // newline for next row

Procedure FLOYD_WARSHALL()
    // dist holds current shortest distances
    dist = COPY_MATRIX(matrix)

    // k = intermediate vertex, i = source, j = destination
    for k in range(V):
        for i in range(V):
            for j in range(V):
```

```
                // Only update if paths i->k and k->j exist
                if dist[i][k] != INF and dist[k][j] != INF:
                    if dist[i][k] + dist[k][j] < dist[i][j]:
                        dist[i][j] = dist[i][k] + dist[k][j]

    DISPLAY_SOLUTION(dist)

Procedure MAIN()
    input V
    INF = infinity
    matrix = [[INF for j in range(V)] for i in range(V)]
    for i in range(V):
        matrix[i][i] = 0

    input E
    for _ in range(E):
        input u, v, w
        ADD_EDGE(u, v, w)

    FLOYD_WARSHALL()
```

BELLMAN FORD

In [ ]:

```
# Bellman-Ford Algorithm Implementation (Adjacency List)
# No external libraries used.

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        # Adjacency list: {vertex: [(neighbor, weight), ...]}
        # Bellman-Ford works best with Directed Graphs usually
        self.adj = {i: [] for i in range(vertices)}

    def add_edge(self, u, v, weight):
        """
        Adds a directed edge from u to v.
        """
        if 0 <= u < self.V and 0 <= v < self.V:
            self.adj[u].append((v, weight))
        else:
            print(f"Error: Vertices {u} or {v} out of bounds.")

    def bellman_ford(self, src):
        """
        Calculates shortest paths from src to all other vertices.
        Detects negative weight cycles.
        """
        # Step 1: Initialize distances from src to all other vertices as IN
        dist = [float('inf')] * self.V
        dist[src] = 0

        print(f"\nRunning Bellman-Ford Algorithm from Source: {src}")

        # Step 2: Relax all edges |V| - 1 times
        # A simple shortest path from src to any other vertex can have at-m
        for i in range(self.V - 1):
```

```python
            # Iterate over all vertices and their edges
            for u in range(self.V):
                for v, w in self.adj[u]:

                    # Relaxation Step
                    if dist[u] != float('inf') and dist[u] + w < dist[v]:
                        dist[v] = dist[u] + w

        # Step 3: Check for Negative Weight Cycles
        # The above step guarantees shortest distances if graph doesn't con
        # negative weight cycle. If we get a shorter path, then there is a
        for u in range(self.V):
            for v, w in self.adj[u]:
                if dist[u] != float('inf') and dist[u] + w < dist[v]:
                    print("Error: Graph contains a Negative Weight Cycle")
                    return # Stop execution

        # Step 4: Print the calculated distances
        print("-" * 40)
        print(f"{'Vertex':<10} | {'Distance from Source'}")
        print("-" * 40)
        for i in range(self.V):
            if dist[i] == float('inf'):
                print(f"{i:<10} | Unreachable")
            else:
                print(f"{i:<10} | {dist[i]}")


# --- Driver Code ---
if __name__ == "__main__":
    print("--- Bellman-Ford Algorithm ---")
    try:
        v_count = int(input("Enter number of vertices (0 to N-1): "))
        g = Graph(v_count)

        e_count = int(input("Enter number of edges: "))
        print("Enter edges in format: Source Destination Weight")
        print("(Note: Negative weights are allowed)")

        for _ in range(e_count):
            try:
                u, v, w = map(int, input().split())
                g.add_edge(u, v, w)
            except ValueError:
                print("Invalid edge input.")

        start = int(input("Enter source vertex: "))

        if 0 <= start < v_count:
            g.bellman_ford(start)
        else:
```

```
            print(f"Error: Source vertex must be between 0 and {v_count-1}"

    except ValueError:
        print("Invalid input! Please enter integers only.")
```

```
Data structures
    V = number of vertices
    INF = infinity
    adj = {i: [] for i in range(V)}      // adjacency list: vertex -> list of (neighbor, weight)
    dist = [INF] * V                     // distances from source

Procedure ADD_EDGE(u, v, weight)
    // directed edge u -> v
    if 0 <= u < V and 0 <= v < V:
        adj[u].append((v, weight))
    // else: ignore or report out-of-bounds

Procedure INITIALIZE(src)
    for i in range(V):
        dist[i] = INF
    dist[src] = 0

Procedure RELAX_ALL_EDGES()
    // Relax every edge once (one pass)
    for u in range(V):
        for (v, w) in adj[u]:
            if dist[u] != INF and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

Procedure DETECT_NEGATIVE_CYCLE() -> boolean
    // If any edge can still be relaxed, a negative cycle exists
    for u in range(V):
        for (v, w) in adj[u]:
            if dist[u] != INF and dist[u] + w < dist[v]:
                return True
    return False

Procedure BELLMAN_FORD(src)
    INITIALIZE(src)

    // Step 2: Relax all edges |V| - 1 times
    for i in range(V - 1):
        RELAX_ALL_EDGES()

    // Step 3: Check for negative weight cycles
    if DETECT_NEGATIVE_CYCLE():
        print("Error: Graph contains a Negative Weight Cycle")
        return   // stop execution

    // Step 4: Print final distances
    print("-----------------------------")
    print("Vertex     | Distance from Source")
    print("-----------------------------")
    for i in range(V):
        if dist[i] == INF:
            print(i, "|", "Unreachable")
        else:
            print(i, "|", dist[i])

Procedure MAIN()
    input V
    adj = {i: [] for i in range(V)}
    dist = [INF] * V

    input E
    for _ in range(E):
        input u, v, w
        ADD_EDGE(u, v, w)

    input src
    if 0 <= src < V:
        BELLMAN_FORD(src)
    else:
        print("Error: Source vertex out of range")
```