```python
# GraphNode Class
class GraphNode:
    # simple graph node class for adjacency list representation
    def __init__(self, name):
        self.name = name
        self.adjacency_list = []

    # enable nodes to be compared with one another
    def __lt__(self, other):
        return True

    def __le__(self, other):
        return True
```

```python
# GraphEdge Class
class GraphEdge:
    # graph edge class with from and to nodes
    def __init__(self, from_node, to_node):
        self.from_node = from_node
        self.to_node = to_node
```

```python
# Graph Class
class Graph:
    # simple graph class for adjacency list representation
    def __init__(self):
        self.num_nodes = 0
        self.num_edges = 0
        self.node_array = []
        self.node_dictionary = {}

    def add_node(self, name):
        new_node = GraphNode(name)
        self.node_array.append(new_node)
        self.node_dictionary[name] = new_node
        self.num_nodes += 1

    def print_nodes(self):
        if self.num_nodes == 0:
            print("Empty graph")
            return
        for i in range(self.num_nodes):
            print(self.node_array[i].name)

    def add_edge_by_node(self, from_node, to_node):
        from_node.adjacency_list.append(GraphEdge(from_node, to_node))
        self.num_edges += 1

    def add_edge(self, from_name, to_name):
        from_node = self.node_dictionary[from_name]
        to_node = self.node_dictionary[to_name]
        self.add_edge_by_node(from_node, to_node)

    def print_graph(self):
        if self.num_nodes == 0:
            print("Empty graph")
            return
        print("The graph structure is:")
        for i in range(self.num_nodes):
            node = self.node_array[i]
            print(f"{node.name}: {[edge.to_node.name for edge in node.adjacency_list]}")
```

```python
def print_graph(self):
        if self.num_nodes == 0:
            print("Empty graph")
            return
        print("The graph structure is:")
        for i in range(self.num_nodes):
            node = self.node_array[i]
            print(f"{node.name}: {[edge.to_node.name for edge in node.adjacency_list]}")
```

```python
# Function to create a graph from an adjacency matrix
import numpy as np

def graph_from_adjacency_matrix(names_array: list, adjacency_matrix: np.array):
    # function which takes as input a name vector and an adjacency matrix
    # and returns a Graph with adjacency list representation

    # check inputs
    n = len(names_array)
    if adjacency_matrix.shape != (n, n):
        print("Incongruent inputs")
        return None

    # initialise graph
    output_graph = Graph()

    # add nodes with names given in name vector
    for name in names_array:
        output_graph.add_node(str(name))

    # add edges based upon adjacency matrix
    for i in range(len(names_array)):
        for j in range(len(names_array)):
            if adjacency_matrix[i, j] > 0:
                output_graph.add_edge(str(names_array[i]), str(names_array[j]))

    return output_graph
```

Q1(a) Create a function called bfs_reachable_nodes which implements a variant of the breadth-first-search
algorithm. Your function should takes as input a graph object G belonging to the Graph class implemented above,
along with a string initial_node_name corresponding to the name of a node within the graph, which we shall

refer to as the initial node. The function should return a Python list containing all of the nodes (instances of the
GraphNode object) within the graph which are reachable from the initial node.
You can begin with the following outline code:

```python
from collections import deque

def bfs_reachable_nodes(input_graph: Graph, initial_node_name: str) -> list:
    # Initialization
    initial_node = None

    # Find the initial node by name and mark all nodes as not visited
    for node in input_graph.node_array:
        if node.name == initial_node_name:
            initial_node = node
            node.visited = True
        else:
            node.visited = False

    # Exit if the initial node isn't found
    if initial_node is None:
        print(f"Start node {initial_node_name} not found.")
        return []

    # Initialize a list of reachable nodes from the initial node
    list_of_reachable_nodes = [initial_node]

    # Initialize queue data structure
    node_queue = deque([initial_node])

    # Perform BFS
    while node_queue:
        current_node = node_queue.popleft()

        for edge in current_node.adjacency_list:
            neighbor = edge.to_node
            if not neighbor.visited:
                neighbor.visited = True
                list_of_reachable_nodes.append(neighbor)
                node_queue.append(neighbor)

    return list_of_reachable_nodes
```

```python
import numpy as np

# Example usage
np.random.seed(2024)  # Set random seed
num_tests = 10
num_nodes = 10

# Define node names
names_v = [chr(ord('A') + i) for i in range(num_nodes)]

# Run multiple tests
for i in range(num_tests):
    adj_mat = np.random.choice([0, 1], size=(num_nodes, num_nodes), p=[0.8, 0.2])
    G = graph_from_adjacency_matrix(names_v, adj_mat)  # Create the graph
    reachable_nodes = [node.name for node in bfs_reachable_nodes(G, "A")]
    print(f"There are {len(reachable_nodes)} reachable nodes: {', '.join(reachable_nodes)}")
```

```
There are 1 reachable nodes: A
There are 9 reachable nodes: A, B, C, D, E, H, F, G, J
There are 8 reachable nodes: A, E, F, G, I, C, B, H
There are 2 reachable nodes: A, E
There are 9 reachable nodes: A, E, J, H, D, F, B, I, C
There are 7 reachable nodes: A, E, H, I, G, J, F
There are 1 reachable nodes: A
There are 7 reachable nodes: A, F, H, J, B, G, I
There are 1 reachable nodes: A
There are 10 reachable nodes: A, B, C, F, G, H, E, J, I, D
```

Q1(b) Name an application of the breadth-first-search algorithm.

shortest path finding in unweighted graphs

Q2(a) Create a function called dfs_reachable_nodes which implements a variant of the depth-first-search
algorithm. Your function should have the same input and output structure as in the previous question: The input
is a graph object G belonging to the Graph class implemented above, along with a string initial_node_name
corresponding to the name of a node within the graph, which we shall refer to as the initial node. The function
should return a Python list containing all of the nodes (instances of the GraphNode object) within the graph

which are reachable from the initial node.

You can begin with the following outline code:

```python
def dfs_reachable_nodes(input_graph: Graph, initial_node_name: str) -> list:
    # Find initial node by name and mark all other nodes as not visited
    initial_node = None
    for node in input_graph.node_array:
        if node.name == initial_node_name:
            initial_node = node
            node.visited = True
        else:
            node.visited = False

    # Exit if the initial node isn't found
    if initial_node is None:
        print(f"Start node {initial_node_name} not found.")
        return None

    # Initialize stack and list of reachable nodes
    node_stack = [initial_node]  # Stack initialized with the start node
    list_of_reachable_nodes = []  # List to store reachable nodes

    # Perform iterative DFS
    while node_stack:
        current_node = node_stack.pop()
        if not current_node.visited:
            current_node.visited = True
            list_of_reachable_nodes.append(current_node)


        for edge in current_node.adjacency_list:
            neighbor = edge.to_node
            if not neighbor.visited:
                node_stack.append(neighbor)


    return list_of_reachable_nodes
```

```python
import numpy as np

# Test the function
np.random.seed(2024)  # Set random seed
num_tests = 10
num_nodes = 10

names_v = [chr(ord('A') + i) for i in range(num_nodes)]

for i in range(num_tests):
    adj_mat = np.random.choice([0, 1], size=(num_nodes, num_nodes), p=[0.8, 0.2])
    G = graph_from_adjacency_matrix(names_v, adj_mat)  # Assumes this function exists
    reachable_nodes = [node.name for node in dfs_reachable_nodes(G, "A")]
    print(f"There are {len(reachable_nodes)} reachable nodes: {', '.join(reachable_nodes)}")
```

```
There are 0 reachable nodes:
There are 8 reachable nodes: D, G, H, J, E, F, B, C
There are 7 reachable nodes: F, C, E, I, G, B, H
There are 1 reachable nodes: E
There are 8 reachable nodes: J, H, F, B, I, C, D, E
There are 6 reachable nodes: I, G, J, E, F, H
There are 0 reachable nodes:
There are 6 reachable nodes: J, G, I, H, B, F
There are 0 reachable nodes:
There are 9 reachable nodes: C, G, B, H, J, F, E, I, D
```

Q2(b)Write a Python called topological_sort which takes as input a graph object G belonging to the Graph
which represents directed acyclic graph. Your function should return a standard Python list corresponding to a
topological ordering of the nodes. That is, the returned list should contain every node within the graph exactly
once. Moreover, the ordering of the nodes within the returned list should be a topological ordering, so if there is
an edge e = (u, w) in the graph, then the node w should not occur earlier than the node u within the list.
Run the following test code to check your function.

```python
from collections import deque

def DFS_topological_sort(input_graph: Graph, initial_node: GraphNode, sorted_list: deque):
    # Mark the initial node as visited
    initial_node.visited = True

    # Visit all neighbors of the current node
    for edge in initial_node.adjacency_list:
        next_node = edge.to_node
        if not hasattr(next_node, "visited") or not next_node.visited:
            # Recursive call for unvisited nodes
            DFS_topological_sort(input_graph, next_node, sorted_list)

    # Insert the current node at the beginning of the sorted list
    sorted_list.appendleft(initial_node)

def topological_sort(input_graph: Graph) -> list:
    # Initialize an empty doubly-linked list to store the sorted order
    sorted_list = deque()

    # Perform DFS from each unvisited node
    for node in input_graph.node_array:
        if not hasattr(node, "visited") or not node.visited:
            DFS_topological_sort(input_graph, node, sorted_list)

    # Convert the sorted list (deque) to a standard list and return
    return list(sorted_list)
```

```python
# Test the topological sort implementation
np.random.seed(2024)  # Set random seed
for j in range(6, 30, 4):
    names_v = [chr(ord('A') + i) for i in range(j)]
    np.random.shuffle(names_v)
    adj_mat = np.triu(np.ones((j, j)), 1)  # Create a DAG (upper triangular matrix)
    G = graph_from_adjacency_matrix(names_v, adj_mat)
    np.random.shuffle(G.node_array)
    top_order = topological_sort(G)
    print(*(node.name for node in top_order), sep=', ')
```

```
E, B, D, F, C, A
G, H, J, I, B, E, F, A, C, D
J, K, I, C, A, N, B, E, M, D, F, H, G, L
D, C, M, E, L, F, Q, J, K, B, P, A, R, O, N, I, H, G
H, S, T, O, N, V, C, I, E, K, D, R, A, M, J, U, B, G, Q, F, P, L
L, M, A, I, D, Q, V, B, W, S, G, N, C, Y, R, J, T, H, K, E, X, F, O, Z, P, U
```

```python
def label_nodes_DFS(graph: Graph, node: GraphNode, visiting: set, visited: set, topological_or

    visiting.add(node)
    for edge in node.adjacency_list:
        next_node = edge.to_node
        if next_node in visiting:
            return False
        if next_node not in visited:
            if not label_nodes_DFS(graph, next_node, visiting, visited, topological_order):
                return False

    # Mark node as fully processed
    visiting.remove(node)

    # Mark node as visited
    visited.add(node)

    # Add node to the topological order
    topological_order.appendleft(node)
    return True


def topological_sort_DAG_check(graph: Graph):

    # Nodes currently in the visiting path
    visiting = set()

    # Nodes that have been fully processed
    visited = set()

    # Stores the topological order
    topological_order = deque()

    # Perform DFS from every node
    for node in graph.node_array:
        if node not in visited:
            if not label_nodes_DFS(graph, node, visiting, visited, topological_order):
                return False

    return list(topological_order)
```

```python
import numpy as np

np.random.seed(2024)  # Set random seed
for j in range(9, 30):
    names_v = [chr(ord('A') + i) for i in range(j)]
    np.random.shuffle(names_v)
    adj_mat = np.triu(np.ones((j, j)), 1)  # Create a DAG (upper triangular matrix)
    if j % 3 == 0:
        adj_mat[-1, 0] = 1  # Add a cycle for testing
    G = graph_from_adjacency_matrix(names_v, adj_mat)
    np.random.shuffle(G.node_array)
    top_order = topological_sort_DAG_check(G)
    if top_order:
        print(f"DAG: {', '.join(node.name for node in top_order)}")
    else:
        print("Cycle found!")
```

```
Cycle found!
DAG: J, A, I, F, E, D, G, C, H, B
DAG: J, K, I, C, A, H, B, E, G, D, F
Cycle found!
DAG: J, I, D, F, A, B, E, H, G, K, M, L, C
DAG: F, I, C, K, E, L, H, M, N, A, J, B, D, G
Cycle found!
DAG: K, P, O, D, I, C, L, A, F, E, B, H, N, G, J, M
DAG: M, C, I, G, P, B, N, E, F, O, L, H, K, Q, J, A, D
Cycle found!
DAG: Q, N, O, B, H, C, I, P, D, M, G, E, A, L, K, R, F, J, S
DAG: G, N, D, C, A, B, E, I, K, S, O, P, T, M, H, J, L, R, F, Q
Cycle found!
DAG: C, D, E, S, G, M, L, V, J, A, N, U, T, O, F, K, H, Q, B, R, P, I
DAG: J, L, G, Q, D, S, K, R, W, U, V, N, B, O, I, A, F, P, T, C, M, H, E
Cycle found!
DAG: O, K, S, Y, N, R, B, W, X, I, M, G, P, L, A, T, C, E, V, J, D, Q, U, H, F
DAG: G, F, N, O, H, D, V, U, A, B, K, P, Z, C, I, W, S, R, X, Q, J, M, E, T, L,
Cycle found!
DAG: R, A, B, [, T, Q, E, N, O, C, K, Y, U, I, M, G, P, S, D, J, Z, V, L, W, H,
DAG: V, F, D, A, L, K, I, P, H, W, S, N, X, ], Z, E, B, Y, \, [, R, U, M, G, Q,
```

```python
import numpy as np

def compute_max_island_area(grid_map: np.array) -> int:
    def dfs(i, j):

        # If out of bounds or not land, return 0
        if i < 0 or j < 0 or i >= len(grid_map) or j >= len(grid_map[0]) or grid_map[i][j] ==
            return 0

        # Mark the current cell as visited
        grid_map[i][j] = 0

        # Explore all 4 directions
        return 1 + dfs(i + 1, j) + dfs(i - 1, j) + dfs(i, j + 1) + dfs(i, j - 1)

    max_island_area = 0
    for i in range(len(grid_map)):
        for j in range(len(grid_map[0])):
            if grid_map[i][j] == 1:

                # Calculate the area of the island starting from (i, j)
                max_island_area = max(max_island_area, dfs(i, j))

    return max_island_area
```

```python
from scipy import signal
import numpy as np

def random_grid_map(grid_width, land_propensity=0.5, interaction_strength=3):
    # Generate a random grid map
    grid_map = np.random.choice([0, 1],
                                size=(grid_width, grid_width),
                                p=[1-land_propensity, land_propensity])
    grid_map = (0.5 + signal.convolve2d(grid_map,
                                        np.ones((interaction_strength, interaction_strength)))
            (interaction_strength ** 2)).astype(int)
    # Ensure outer squares are all land
    grid_map[0, :] = 0
    grid_map[-1, :] = 0
    grid_map[:, 0] = 0
    grid_map[:, -1] = 0

    return grid_map

np.random.seed(2024)

grid_map = random_grid_map(6)
print("The grid map looks as follows:")
print(grid_map)

mx_area = compute_max_island_area(grid_map)
print(f"The largest island has an area of {mx_area}.")
```

```
The grid map looks as follows:
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 1 1 1 0 0 0 0]
 [0 1 1 1 0 0 0 0]
 [0 1 1 1 1 0 0 0]
 [0 0 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]]
The largest island has an area of 12.
```

```
np.random.seed(2024) # set random seed
num_lakes_list=[]
for i in range(15):
    grid_map=random_grid_map(100, land_propensity=0.45, interaction_strength=5)
    num_lakes_list.append(compute_max_island_area(grid_map))
print(num_lakes_list)
```

```
[524, 170, 275, 274, 570, 308, 305, 268, 326, 289, 391, 329, 317, 423, 287]
```

Q4(a) Name two examples of greedy algorithms.

1. Activity Selection Problem
   Problem: Given a set of activities with their start and finish times, select the maximum number of activities that don't overlap.
   Greedy Strategy: Always pick the activity that finishes the earliest (among the remaining activities) to maximize the remaining time for other activities.
2. Huffman Encoding
   Problem: Construct an optimal prefix code for data compression based on the frequency of characters in a string.
   Greedy Strategy: Repeatedly combine the two least frequent nodes (characters or groups of characters) into a new node, minimizing the total cost of encoding.

Q4(b) Create a function called dijkstras_min_euclidean_norm_path which takes as input an adjacency
list representation input_graph for a graph in which every edge within the graph has a non-negative edge.edge_length attribute, as well as an initial node object initial_node. Your function should modify the existing graph data structure in-place so that every node node has an additional attribute node.min_euclidean_norm_path_from_start which is equal to the minimum Euclidean norm of a path from
the initial node initial_node to the node object itself. If there is no path from initial_node to node then we have node.min_euclidean_norm_path_from_start=np.Inf. We also define the Euclidean norm of the
empty path to be 0 so that node.min_euclidean_norm_path_from_start=0.

```python
import heapq
import numpy as np

def dijkstras_min_euclidean_norm_path(input_graph: Graph, initial_node: GraphNode):
    for node in input_graph.node_array:
        node.min_euclidean_norm_path_from_start = np.inf

    # Distance to itself is 0
    initial_node.min_euclidean_norm_path_from_start = 0

    #initialise heap with initial node.
    min_dist_heap = [(0, initial_node)]

    while min_dist_heap:
        # retrieve node with minimal distance from initial node
        current_distance, current_node = heapq.heappop(min_dist_heap)

        # If the current distance is greater than the recorded distance, skip processing (alre
        if current_distance > current_node.min_euclidean_norm_path_from_start:
            continue

        for edge in current_node.adjacency_list:
            neighbor = edge.to_node
            edge_length = edge.edge_length
            new_distance = current_distance + edge_length

            if new_distance < neighbor.min_euclidean_norm_path_from_start:
                neighbor.min_euclidean_norm_path_from_start = new_distance
                # insert tuple for edges from v and an unvisited node.
                heapq.heappush(min_dist_heap, (new_distance, neighbor))
```

```python
def assign_random_edge_lengths(graph: Graph, edge_length_max: int, random_seed=0):
    np.random.seed(random_seed)  # Set random seed for reproducibility

    for node in graph.node_array:
        for edge in node.adjacency_list:
            edge.edge_length = np.random.randint(1, edge_length_max + 1)  # Random edge length
```

```python
def print_dijkstras_min_euclidean_norm_path_algorithm_output(graph: Graph, initial_node_name: s
    initial_node = graph.node_dictionary[initial_node_name]
    dijkstras_min_euclidean_norm_path(graph, initial_node)  # Run Dijkstra's algorithm

    # Print direct descendants and edge lengths
    for i in range(graph.num_nodes):
        node = graph.node_array[i]
        print("{node}: {direct_descendants}".format(
            node=node.name,
            direct_descendants=[(edge.to_node.name, edge.edge_length) for edge in node.adjacenc
        ))

    # Print minimum norm path results
    for i in range(graph.num_nodes):
        node = graph.node_array[i]
        print("The minimum norm path from node {init_node} to node {node} is {norm}".format(
            node=node.name,
            norm=node.min_euclidean_norm_path_from_start,
            init_node=initial_node_name
        ))
```

```python
import numpy as np

np.random.seed(2024)  # Set random seed
num_nodes = 20

# Create adjacency matrix
adj_mat = np.random.choice([0, 1], size=(num_nodes, num_nodes), p=[0.7, 0.3])
names_v = [chr(ord('A') + i) for i in range(num_nodes)]

# Create graph from adjacency matrix
G = graph_from_adjacency_matrix(names_v, adj_mat)

# Assign random edge lengths
assign_random_edge_lengths(G, 10)

# Run and print Dijkstra's algorithm output
print_dijkstras_min_euclidean_norm_path_algorithm_output(G, "A")
```

```
A: [('G', 6), ('L', 1), ('N', 4), ('T', 4)]
B: [('C', 8), ('F', 10), ('G', 4), ('L', 6), ('O', 3), ('Q', 5), ('R', 8)]
C: [('D', 7), ('E', 9), ('H', 9), ('J', 2), ('N', 7), ('Q', 8), ('R', 8), ('T', 
D: [('C', 2), ('D', 6), ('I', 10), ('L', 9), ('S', 10), ('T', 5)]
E: [('C', 4), ('I', 1), ('M', 4), ('P', 6), ('Q', 1)]
F: [('B', 3), ('C', 4), ('D', 9), ('E', 2), ('L', 4), ('M', 4), ('O', 4), ('P', 
G: [('A', 1), ('B', 2), ('C', 10), ('D', 10), ('H', 1), ('L', 5), ('P', 8), ('Q'
H: [('A', 3), ('C', 8), ('F', 3), ('L', 1)]
I: [('B', 1), ('F', 5), ('H', 6), ('M', 6), ('N', 7), ('P', 9), ('R', 5), ('T', 
J: [('A', 5), ('B', 10), ('D', 9), ('E', 2), ('J', 2), ('L', 8), ('O', 10), ('P'
K: [('B', 7), ('E', 8), ('F', 3), ('L', 1), ('M', 4), ('P', 6), ('Q', 10), ('R', 
L: [('B', 7), ('C', 5), ('F', 5), ('K', 4), ('L', 5), ('O', 5), ('Q', 9), ('T', 
M: [('G', 4), ('H', 8), ('I', 6), ('M', 6)]
N: [('B', 1), ('I', 2), ('T', 6)]
O: [('G', 10), ('J', 4), ('P', 1), ('R', 6), ('S', 1), ('T', 2)]
P: [('E', 3), ('H', 5), ('R', 3), ('T', 1)]
Q: [('D', 4), ('E', 3), ('F', 1), ('J', 8), ('K', 6)]
R: [('A', 10), ('L', 1), ('N', 3), ('R', 8)]
S: [('B', 3), ('C', 10), ('G', 3), ('I', 4), ('O', 4), ('P', 3), ('S', 4)]
T: [('H', 5), ('N', 2), ('R', 3), ('T', 10)]
The minimum norm path from node A to node A is 0
The minimum norm path from node A to node B is 5
The minimum norm path from node A to node C is 6
The minimum norm path from node A to node D is 13
The minimum norm path from node A to node E is 8
The minimum norm path from node A to node F is 6
The minimum norm path from node A to node G is 6
The minimum norm path from node A to node H is 7
The minimum norm path from node A to node I is 6
The minimum norm path from node A to node J is 8
The minimum norm path from node A to node K is 5
The minimum norm path from node A to node L is 1
The minimum norm path from node A to node M is 9
The minimum norm path from node A to node N is 4
The minimum norm path from node A to node O is 6
The minimum norm path from node A to node P is 7
The minimum norm path from node A to node Q is 9
The minimum norm path from node A to node R is 7
The minimum norm path from node A to node S is 7
The minimum norm path from node A to node T is 4
```

Q4(c) Justify the claim that once the algorithm implemented by the dijkstras_min_euclidean_norm_path
function has terminated, for every node in the graph the attribute

node.min_euclidean_norm_path_from_start
will be equal to the minimum Euclidean norm of a path from the initial node initial_node to the node object
itself.

At the beginning of the algorithm, the min_euclidean_norm_path_from_start for all nodes is set to infinity (np.inf), except for the initial node, which is set to 0.
This ensures that the initial node is processed first with a correct initial value of 0 since the path from a node to itself is always of length 0.

The algorithm uses a min-heap (priority queue) to always process the node with the smallest min_euclidean_norm_path_from_start value next.
At any point, the smallest known distance to any unprocessed node is extracted and finalized (because the node's shortest path is guaranteed to have been found at that point).

For each edge (u, v) with weight w, the algorithm checks if going through u to reach v yields a smaller path than the currently known path to v.
If so, the distance to v is updated, and v is added back to the heap for further processing.
This ensures that all possible paths to a node are considered, and only the shortest one is retained.

Once a node is processed (popped from the heap), its shortest path is finalized, and its min_euclidean_norm_path_from_start is never updated again.
The monotonicity property ensures that no shorter path can later emerge for a node already processed, which aligns with the correctness of Dijkstra's algorithm.

Q5(b) Write a function called min_graph_cost which takes as input an $n \times n$ symmetric NumPy matrix edge_cost_matrix with strictly positive entries and returns a number corresponding to the minimum graph
cost graph_cost($G'$
) where $G' = (V, E'$
) is a connected graph.

```python
import heapq
import numpy as np

def min_graph_cost(edge_cost_matrix: np.ndarray) -> int:
    n = edge_cost_matrix.shape[0]
    visited = [False] * n
    min_heap = []
    total_cost = 0

    # Start from the first node (node 0)
    visited[0] = True
    for j in range(1, n):
        if edge_cost_matrix[0, j] > 0:
            heapq.heappush(min_heap, (edge_cost_matrix[0, j], j))

    # Process until all nodes are connected
    while min_heap:
        cost, next_node = heapq.heappop(min_heap)
        if visited[next_node]:
            continue


        visited[next_node] = True
        total_cost += cost

        for j in range(n):
            if not visited[j] and edge_cost_matrix[next_node, j] > 0:
                heapq.heappush(min_heap, (edge_cost_matrix[next_node, j], j))

    return total_cost
```

```python
np.random.seed(2024)  # Set random seed

for n in range(3, 31, 3):
    # Generate a random symmetric edge cost matrix
    edge_cost_matrix = np.random.randint(low=1, high=25, size=(n, n))
    edge_cost_matrix = (edge_cost_matrix + edge_cost_matrix.T) // 2  # Make it symmetric
    np.fill_diagonal(edge_cost_matrix, 0)  # No self-loops

    # Compute the minimum graph cost
    min_cost = min_graph_cost(edge_cost_matrix)
    print(f"The minimum cost for this {n}x{n} edge cost matrix was {min_cost}.")
```

```
The minimum cost for this 3x3 edge cost matrix was 4.
The minimum cost for this 6x6 edge cost matrix was 45.
The minimum cost for this 9x9 edge cost matrix was 51.
The minimum cost for this 12x12 edge cost matrix was 61.
The minimum cost for this 15x15 edge cost matrix was 57.
The minimum cost for this 18x18 edge cost matrix was 68.
The minimum cost for this 21x21 edge cost matrix was 90.
The minimum cost for this 24x24 edge cost matrix was 76.
The minimum cost for this 27x27 edge cost matrix was 94.
The minimum cost for this 30x30 edge cost matrix was 120.
```

Q5(c) Justify the claim that the algorithm implemented by min_graph_cost is guaranteed to return the minimal
cost of a connected undirected graph for the given input edge_cost_matrix.

The min_graph_cost function implements Prim’s algorithm, which is a well-known and proven algorithm for computing the Minimum Spanning Tree (MST) of a graph. The MST connects all the vertices of the graph with the minimal sum of edge weights, ensuring that the graph remains connected and loop-free.

The algorithm starts from an arbitrary node and initializes its cost as 0 (or selects it as the starting point in the MST).
All other nodes are initialized with infinite cost to ensure that the first edge added to the MST has the lowest weight.

For each visited node, the algorithm examines its unvisited neighbors and updates the minimal edge costs in the heap. This ensures that only the minimum-cost edge connecting the MST to the remaining nodes is considered at each step.

Prim’s algorithm is greedy, selecting the smallest edge weight that connects an unvisited vertex at each step. This is guaranteed to result in the MST because:
At every step, adding the smallest edge ensures that the partial solution (current set of edges in the MST) is optimal.
This property is proven mathematically in the cut property of MSTs

The algorithm uses a priority queue (min-heap) to efficiently select the smallest edge at each step, ensuring the minimal cost edge is always processed next.
This guarantees the algorithm’s time complexity of
$O$
(
$n$
2
log

), where
$n$
n is the number of nodes in the graph.

The algorithm stops only when all nodes are visited. Since the algorithm processes edges in increasing order of weight and adds a node to the MST only when it is connected by the smallest edge, the resulting graph is guaranteed to be connected.