Experiment No.5:

Implement Greedy search algorithm for any of the following application:

□ Selection Sort

□ Minimum Spanning Tree

□ Single-Source Shortest Path Problem

□ Job Scheduling Problem

□ Prim's Minimal Spanning Tree Algorithm

□ Kruskal's Minimal Spanning Tree Algorithm

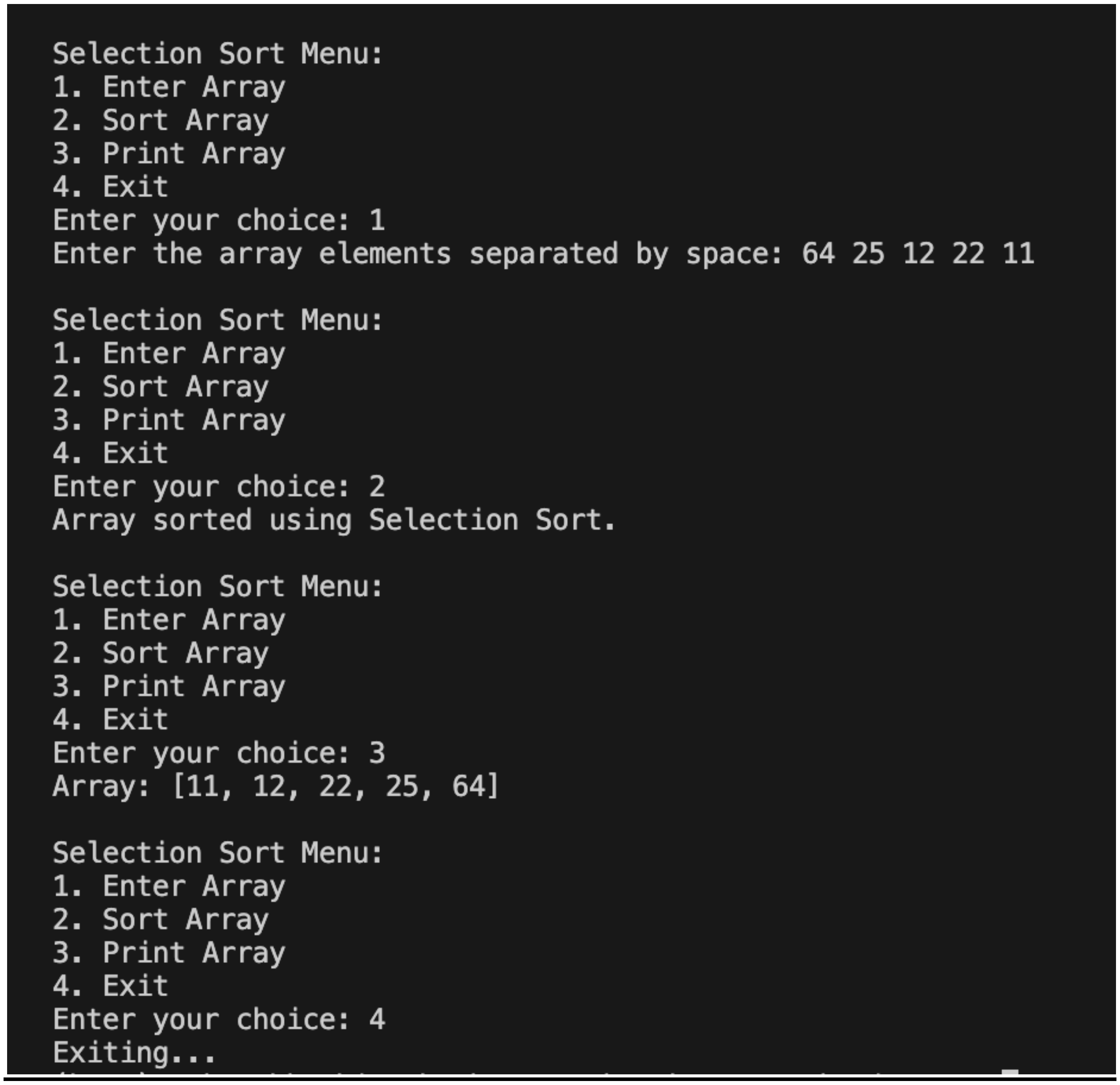
□ Dijkstra's Minimal Spanning Tree Algorithm

Code:

□ Selection Sort

```
def selection_sort(arr):
    n = len(arr)

    for i in range(n - 1):
        min_index = i

        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j

        arr[i], arr[min_index] = arr[min_index], arr[i]

def print_array(arr):
    print("Array:", arr)


def main():
    arr = []
    while True:
        print("\nSelection Sort Menu:")
        print("1. Enter Array")
        print("2. Sort Array")
        print("3. Print Array")
        print("4. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            arr = list(map(int, input("Enter the array elements separated by space: ").split()))
        elif choice == "2":
            if not arr:
                print("Please enter an array first.")
            else:
                selection_sort(arr)
                print("Array sorted using Selection Sort.")
        elif choice == "3":
            if not arr:
                print("Array is empty. Please enter an array first.")
            else:
                print_array(arr)
        elif choice == "4":
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please choose a valid option.")


if __name__ == "__main__":
    main()
```

Output:

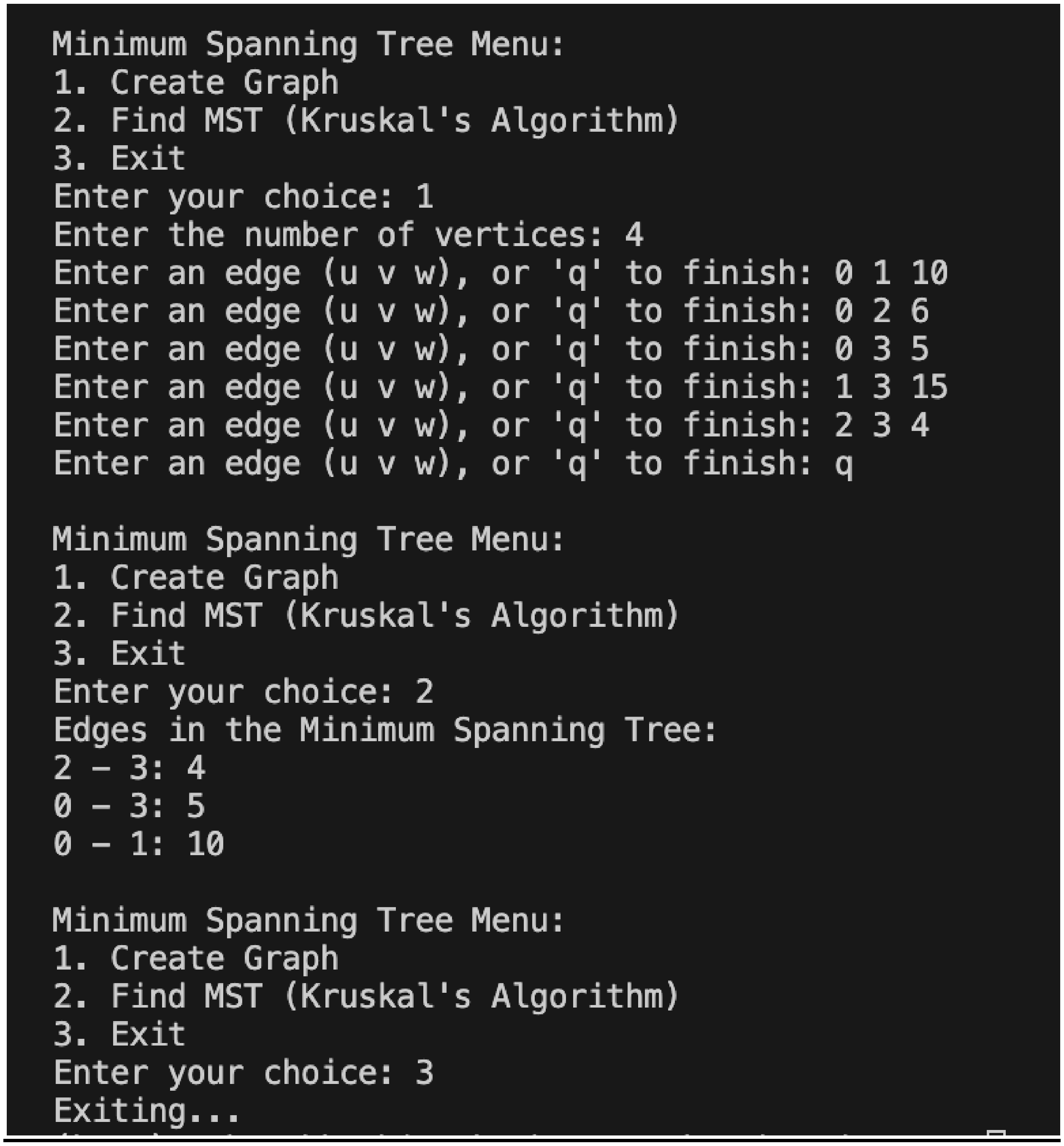


```
Selection Sort Menu:
1. Enter Array
2. Sort Array
3. Print Array
4. Exit
Enter your choice: 1
Enter the array elements separated by space: 64 25 12 22 11

Selection Sort Menu:
1. Enter Array
2. Sort Array
3. Print Array
4. Exit
Enter your choice: 2
Array sorted using Selection Sort.

Selection Sort Menu:
1. Enter Array
2. Sort Array
3. Print Array
4. Exit
Enter your choice: 3
Array: [11, 12, 22, 25, 64]

Selection Sort Menu:
1. Enter Array
2. Sort Array
3. Print Array
4. Exit
Enter your choice: 4
Exiting...
```

🔲 Minimum Spanning Tree

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):
        x_root = self.find(parent, x)
        y_root = self.find(parent, y)

        if rank[x_root] < rank[y_root]:
            parent[x_root] = y_root
        elif rank[x_root] > rank[y_root]:
            parent[y_root] = x_root
        else:
            parent[y_root] = x_root
            rank[x_root] += 1

    def kruskal_mst(self):
        result = []
        i = 0
        e = 0

        self.graph = sorted(self.graph, key=lambda item: item[2])

        parent = []
        rank = []

        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        while e < self.V - 1:
            u, v, w = self.graph[i]
            i += 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            if x != y:
                e += 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)

        return result

def print_mst(mst):
    print("Edges in the Minimum Spanning Tree:")
    for u, v, w in mst:
        print(f"{u} - {v}: {w}")


def main():
    while True:
        print("\nMinimum Spanning Tree Menu:")
        print("1. Create Graph")
        print("2. Find MST (Kruskal's Algorithm)")
        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            vertices = int(input("Enter the number of vertices: "))
            g = Graph(vertices)

            while True:
                edge_info = input("Enter an edge (u v w), or 'q' to finish: ")
                if edge_info == 'q':
                    break
                u, v, w = map(int, edge_info.split())
                g.add_edge(u, v, w)

        elif choice == "2":
            if 'g' in locals():
                mst = g.kruskal_mst()
                print_mst(mst)
            else:
                print("Please create a graph first.")
        elif choice == "3":
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please choose a valid option.")

if __name__ == "__main__":
    main()
```

Output:

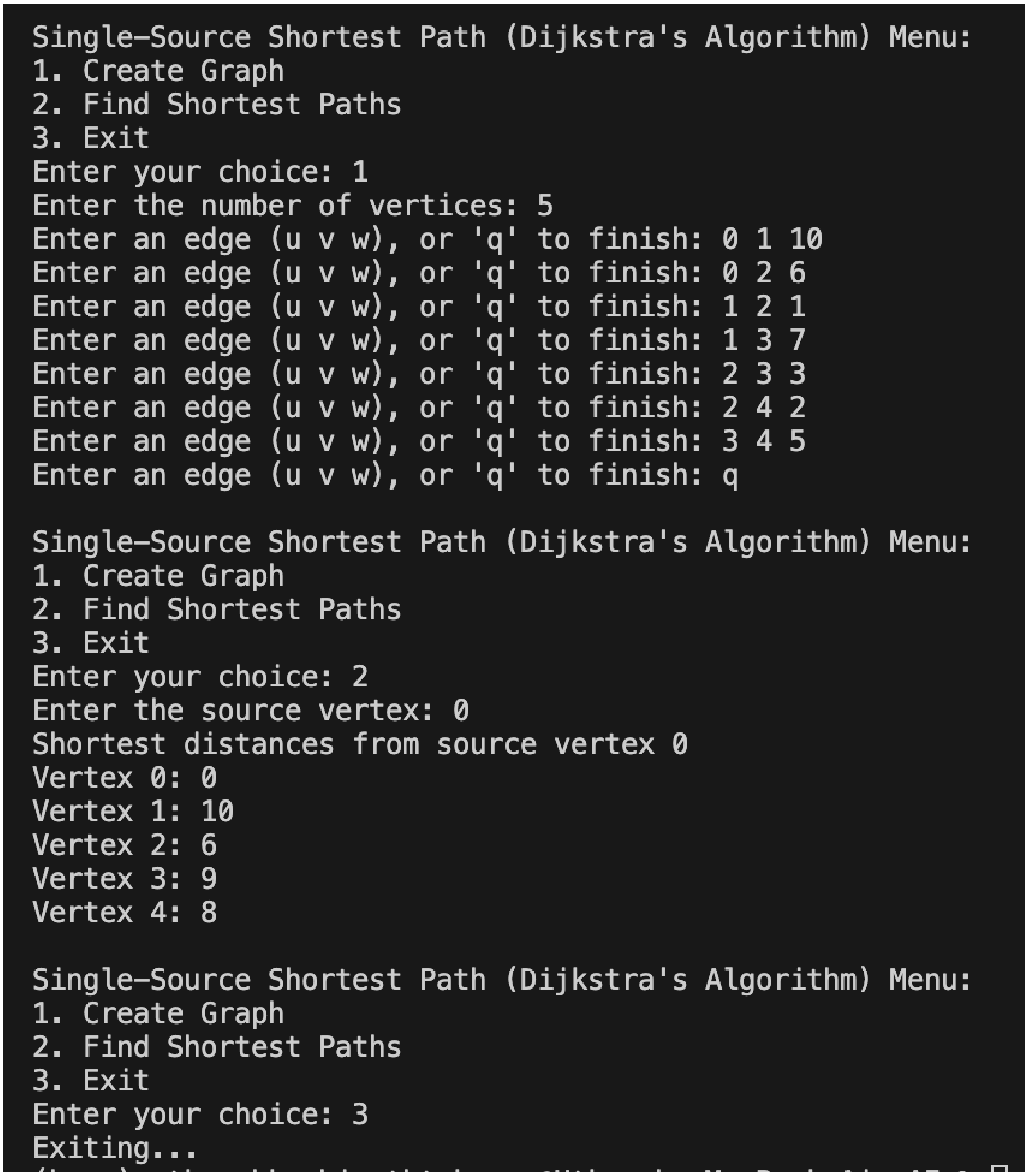


```
Minimum Spanning Tree Menu:
1. Create Graph
2. Find MST (Kruskal's Algorithm)
3. Exit
Enter your choice: 1
Enter the number of vertices: 4
Enter an edge (u v w), or 'q' to finish: 0 1 10
Enter an edge (u v w), or 'q' to finish: 0 2 6
Enter an edge (u v w), or 'q' to finish: 0 3 5
Enter an edge (u v w), or 'q' to finish: 1 3 15
Enter an edge (u v w), or 'q' to finish: 2 3 4
Enter an edge (u v w), or 'q' to finish: q

Minimum Spanning Tree Menu:
1. Create Graph
2. Find MST (Kruskal's Algorithm)
3. Exit
Enter your choice: 2
Edges in the Minimum Spanning Tree:
2 - 3: 4
0 - 3: 5
0 - 1: 10

Minimum Spanning Tree Menu:
1. Create Graph
2. Find MST (Kruskal's Algorithm)
3. Exit
Enter your choice: 3
Exiting...
```

□ Single-Source Shortest Path Problem

```
import heapq

import sys


class Graph:

    def __init__(self, vertices):

        self.V = vertices

        self.graph = [[] for _ in range(vertices)]


    def add_edge(self, u, v, w):

        self.graph[u].append((v, w))


    def dijkstra(self, src):

        distances = [float('inf')] * self.V

        distances[src] = 0


        min_heap = [(0, src)]


        while min_heap:

            dist_u, u = heapq.heappop(min_heap)


            if dist_u > distances[u]:

                continue


            for v, weight in self.graph[u]:

                if distances[u] + weight < distances[v]:

                    distances[v] = distances[u] + weight

                    heapq.heappush(min_heap, (distances[v], v))
        return distances


def print_shortest_paths(distances, src):
    print("Shortest distances from source vertex", src)
    for i, dist in enumerate(distances):
        print(f"Vertex {i}: {dist}")


def main():
    while True:
        print("\nSingle-Source Shortest Path (Dijkstra's Algorithm) Menu:")
        print("1. Create Graph")
        print("2. Find Shortest Paths")
        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            vertices = int(input("Enter the number of vertices: "))
            g = Graph(vertices)

            while True:
                edge_info = input("Enter an edge (u v w), or 'q' to finish: ")
                if edge_info == 'q':
                    break
                u, v, w = map(int, edge_info.split())
                g.add_edge(u, v, w)
        elif choice == "2":
            if 'g' in locals():
                src_vertex = int(input("Enter the source vertex: "))
                distances = g.dijkstra(src_vertex)
                print_shortest_paths(distances, src_vertex)
            else:
                print("Please create a graph first.")
        elif choice == "3":
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please choose a valid option.")


if __name__ == "__main__":
    main()
```

Output:

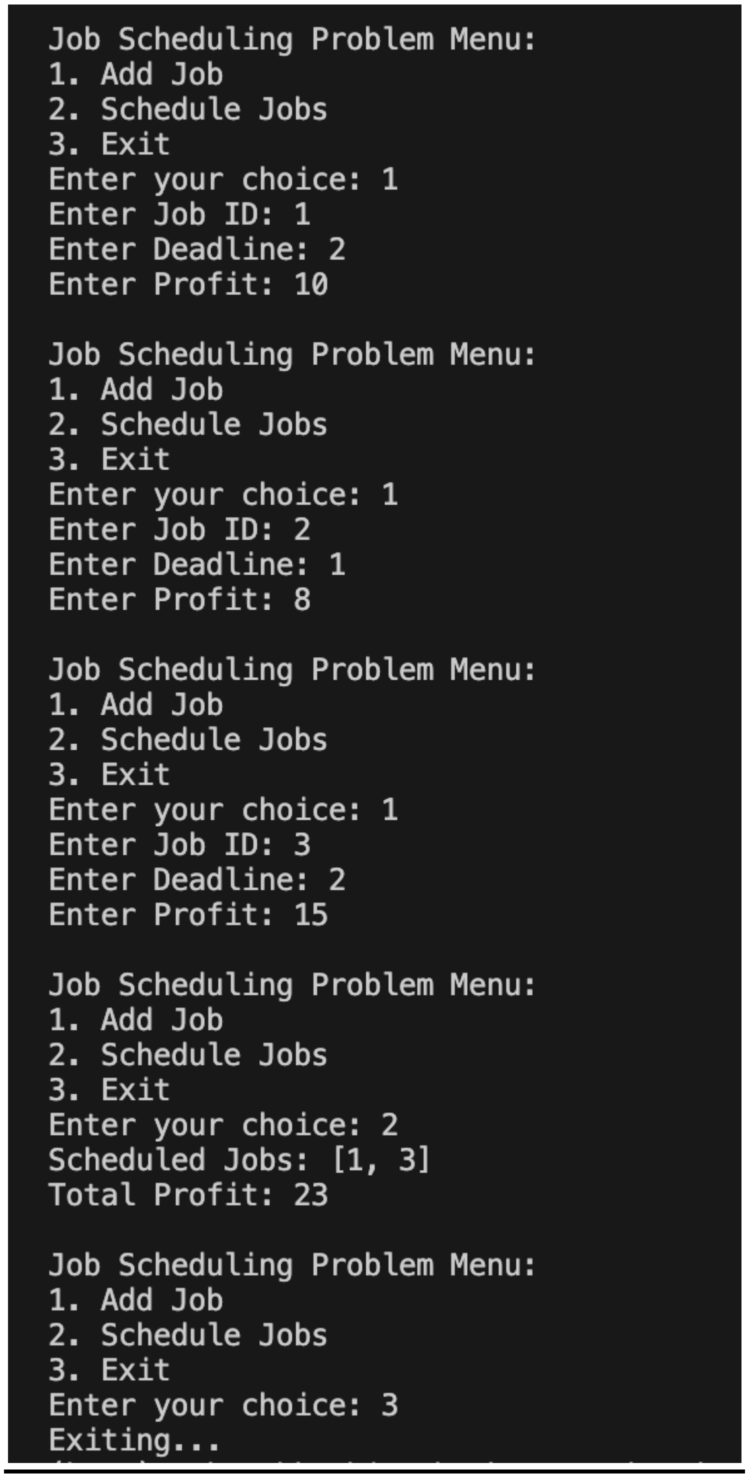


```
Single-Source Shortest Path (Dijkstra's Algorithm) Menu:
1. Create Graph
2. Find Shortest Paths
3. Exit
Enter your choice: 1
Enter the number of vertices: 5
Enter an edge (u v w), or 'q' to finish: 0 1 10
Enter an edge (u v w), or 'q' to finish: 0 2 6
Enter an edge (u v w), or 'q' to finish: 1 2 1
Enter an edge (u v w), or 'q' to finish: 1 3 7
Enter an edge (u v w), or 'q' to finish: 2 3 3
Enter an edge (u v w), or 'q' to finish: 2 4 2
Enter an edge (u v w), or 'q' to finish: 3 4 5
Enter an edge (u v w), or 'q' to finish: q

Single-Source Shortest Path (Dijkstra's Algorithm) Menu:
1. Create Graph
2. Find Shortest Paths
3. Exit
Enter your choice: 2
Enter the source vertex: 0
Shortest distances from source vertex 0
Vertex 0: 0
Vertex 1: 10
Vertex 2: 6
Vertex 3: 9
Vertex 4: 8

Single-Source Shortest Path (Dijkstra's Algorithm) Menu:
1. Create Graph
2. Find Shortest Paths
3. Exit
Enter your choice: 3
Exiting...
```

□ Job Scheduling Problem

```
def job_scheduling(jobs):
    jobs.sort(key=lambda x: x[2], reverse=True)

    max_deadline = max(jobs, key=lambda x: x[1])[1]
    time_slots = [-1] * (max_deadline + 1)

    for job in jobs:
        profit = job[2]
        deadline = job[1]

        # Find the maximum available time slot before the deadline
        for i in range(deadline, 0, -1):
            if i <= max_deadline and time_slots[i] == -1:
                time_slots[i] = job[0]  # Schedule the job
                break

    scheduled_jobs = [time_slots[i] for i in range(1, max_deadline + 1) if time_slots[i] != -1]
    total_profit = sum([jobs[job_id - 1][2] for job_id in scheduled_jobs])

    return scheduled_jobs, total_profit


def print_schedule(jobs, total_profit):
    print("Scheduled Jobs:", jobs)
    print("Total Profit:", total_profit)

def main():
    jobs = []
    while True:
        print("\nJob Scheduling Problem Menu:")
        print("1. Add Job")
        print("2. Schedule Jobs")
        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            job_id = int(input("Enter Job ID: "))
            deadline = int(input("Enter Deadline: "))
            profit = int(input("Enter Profit: "))
            jobs.append((job_id, deadline, profit))
        elif choice == "2":
            if jobs:
                scheduled_jobs, total_profit = job_scheduling(jobs)
                print_schedule(scheduled_jobs, total_profit)
            else:
                print("No jobs added yet. Please add jobs first.")
        elif choice == "3":
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please choose a valid option.")

if __name__ == "__main__":
    main()
```

Output:

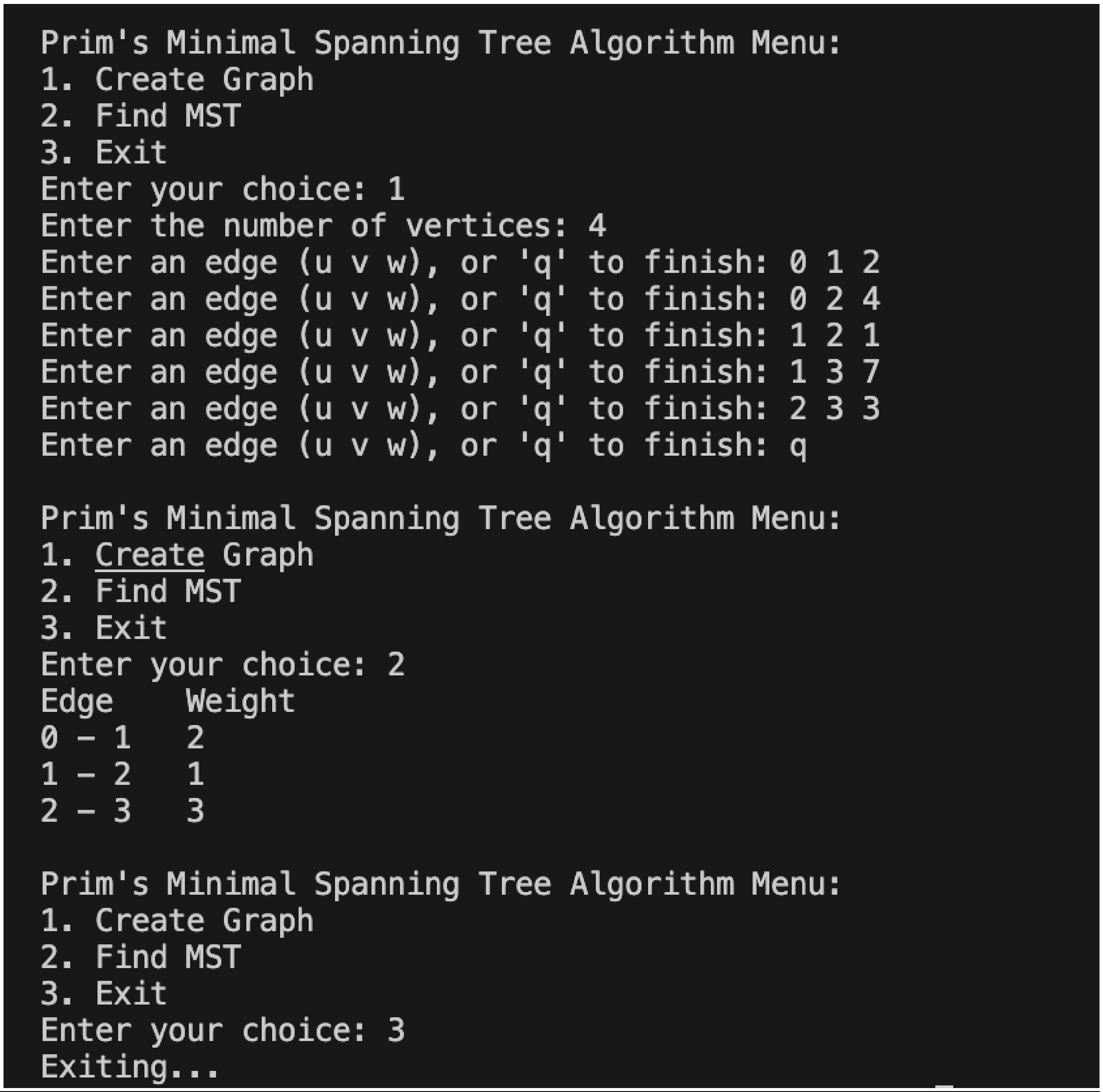


```
Job Scheduling Problem Menu:
1. Add Job
2. Schedule Jobs
3. Exit
Enter your choice: 1
Enter Job ID: 1
Enter Deadline: 2
Enter Profit: 10

Job Scheduling Problem Menu:
1. Add Job
2. Schedule Jobs
3. Exit
Enter your choice: 1
Enter Job ID: 2
Enter Deadline: 1
Enter Profit: 8

Job Scheduling Problem Menu:
1. Add Job
2. Schedule Jobs
3. Exit
Enter your choice: 1
Enter Job ID: 3
Enter Deadline: 2
Enter Profit: 15

Job Scheduling Problem Menu:
1. Add Job
2. Schedule Jobs
3. Exit
Enter your choice: 2
Scheduled Jobs: [1, 3]
Total Profit: 23

Job Scheduling Problem Menu:
1. Add Job
2. Schedule Jobs
3. Exit
Enter your choice: 3
Exiting...
```

□ Prim's Minimal Spanning Tree Algorithm

```
import sys


class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for _ in range(vertices)] for _ in range(vertices)]


    def add_edge(self, u, v, w):
        self.graph[u][v] = w
        self.graph[v][u] = w


    def prim_mst(self):
        key = [float("inf")] * self.V
        parent = [-1] * self.V
        key[0] = 0
        mst_set = [False] * self.V


        for _ in range(self.V):
            u = self.min_key(key, mst_set)
            mst_set[u] = True


            for v in range(self.V):
                if self.graph[u][v] and not mst_set[v] and key[v] > self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u


        return parent
    def min_key(self, key, mst_set):
        min_val = float("inf")
        min_index = -1

        for v in range(self.V):
            if key[v] < min_val and not mst_set[v]:
                min_val = key[v]
                min_index = v

        return min_index

def print_mst(graph, parent):
    print("Edge \tWeight")
    for i in range(1, graph.V):
        print(f"{parent[i]} - {i} \t{graph.graph[i][parent[i]]}")

def main():
    while True:
        print("\nPrim's Minimal Spanning Tree Algorithm Menu:")
        print("1. Create Graph")
        print("2. Find MST")
        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            vertices = int(input("Enter the number of vertices: "))
            g = Graph(vertices)

            while True:
                edge_info = input("Enter an edge (u v w), or 'q' to finish: ")
                if edge_info == 'q':
                    break
                u, v, w = map(int, edge_info.split())
                g.add_edge(u, v, w)

        elif choice == "2":
            if 'g' in locals():
                parent = g.prim_mst()
                print_mst(g, parent)
            else:
                print("Please create a graph first.")
        elif choice == "3":
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please choose a valid option.")

if __name__ == "__main__":
    main()
```

Output:

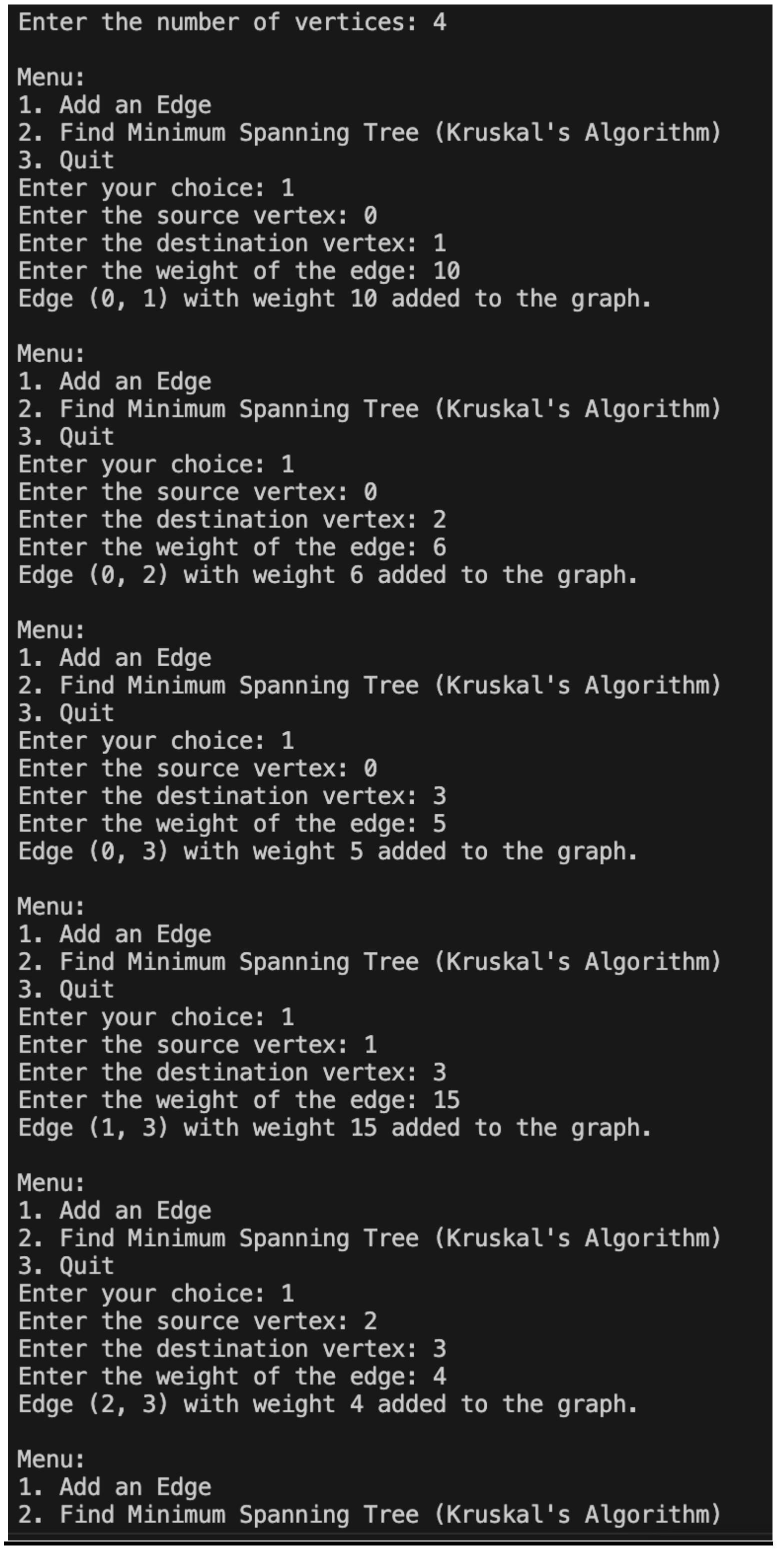


```
Prim's Minimal Spanning Tree Algorithm Menu:
1. Create Graph
2. Find MST
3. Exit
Enter your choice: 1
Enter the number of vertices: 4
Enter an edge (u v w), or 'q' to finish: 0 1 2
Enter an edge (u v w), or 'q' to finish: 0 2 4
Enter an edge (u v w), or 'q' to finish: 1 2 1
Enter an edge (u v w), or 'q' to finish: 1 3 7
Enter an edge (u v w), or 'q' to finish: 2 3 3
Enter an edge (u v w), or 'q' to finish: q

Prim's Minimal Spanning Tree Algorithm Menu:
1. Create Graph
2. Find MST
3. Exit
Enter your choice: 2
Edge    Weight
0 - 1   2
1 - 2   1
2 - 3   3

Prim's Minimal Spanning Tree Algorithm Menu:
1. Create Graph
2. Find MST
3. Exit
Enter your choice: 3
Exiting...
```

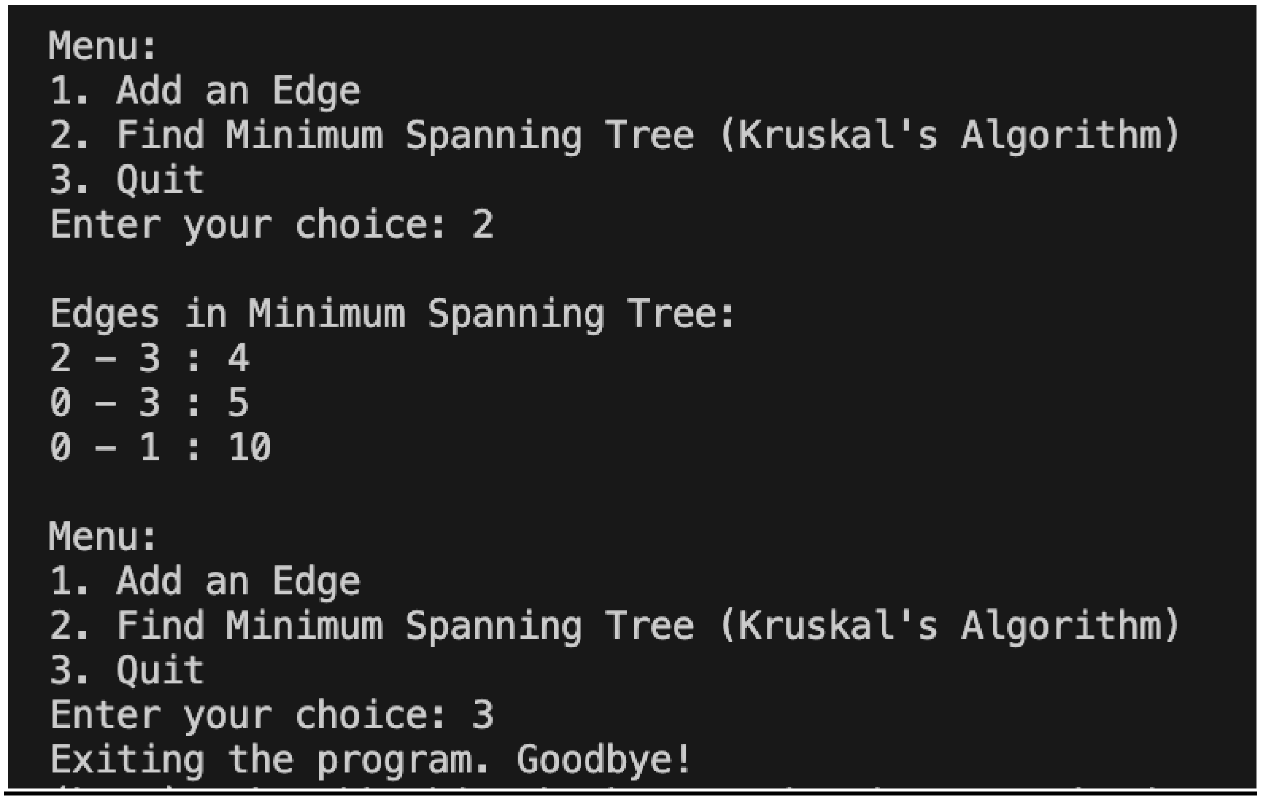
□ Kruskal's Minimal Spanning Tree Algorithm

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):
        root_x = self.find(parent, x)
        root_y = self.find(parent, y)

        if rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
        elif rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        else:
            parent[root_y] = root_x
            rank[root_x] += 1

    def kruskal_mst(self):
        result = []
        i = 0
        e = 0
        self.graph = sorted(self.graph, key=lambda item: item[2])

        parent = []
        rank = []

        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        while e < self.V - 1:
            u, v, w = self.graph[i]
            i += 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            if x != y:
                e += 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)

        return result

def display_menu():
    print("\nMenu:")
    print("1. Add an Edge")
    print("2. Find Minimum Spanning Tree (Kruskal's Algorithm)")
    print("3. Quit")

def add_edge(graph):
    u = int(input("Enter the source vertex: "))
    v = int(input("Enter the destination vertex: "))
    w = int(input("Enter the weight of the edge: "))
    graph.add_edge(u, v, w)
    print(f"Edge ({u}, {v}) with weight {w} added to the graph.")

def find_minimum_spanning_tree(graph):
    minimum_spanning_tree = graph.kruskal_mst()
    print("\nEdges in Minimum Spanning Tree:")
    for edge in minimum_spanning_tree:
        print(f"{edge[0]} - {edge[1]} : {edge[2]}")

def main():
    while True:
        display_menu()
        choice = input("Enter your choice: ")

        if choice == '1':
            add_edge(g)
        elif choice == '2':
            find_minimum_spanning_tree(g)
        elif choice == '3':
            print("Exiting the program. Goodbye!")
            break
        else:
            print("Invalid choice. Please select a valid option.")


if __name__ == "__main__":
    vertices = int(input("Enter the number of vertices: "))
    g = Graph(vertices)


    main()
```

Output:



```
Enter the number of vertices: 4

Menu:
1. Add an Edge
2. Find Minimum Spanning Tree (Kruskal's Algorithm)
3. Quit
Enter your choice: 1
Enter the source vertex: 0
Enter the destination vertex: 1
Enter the weight of the edge: 10
Edge (0, 1) with weight 10 added to the graph.

Menu:
1. Add an Edge
2. Find Minimum Spanning Tree (Kruskal's Algorithm)
3. Quit
Enter your choice: 1
Enter the source vertex: 0
Enter the destination vertex: 2
Enter the weight of the edge: 6
Edge (0, 2) with weight 6 added to the graph.

Menu:
1. Add an Edge
2. Find Minimum Spanning Tree (Kruskal's Algorithm)
3. Quit
Enter your choice: 1
Enter the source vertex: 0
Enter the destination vertex: 3
Enter the weight of the edge: 5
Edge (0, 3) with weight 5 added to the graph.

Menu:
1. Add an Edge
2. Find Minimum Spanning Tree (Kruskal's Algorithm)
3. Quit
Enter your choice: 1
Enter the source vertex: 1
Enter the destination vertex: 3
Enter the weight of the edge: 15
Edge (1, 3) with weight 15 added to the graph.

Menu:
1. Add an Edge
2. Find Minimum Spanning Tree (Kruskal's Algorithm)
3. Quit
Enter your choice: 1
Enter the source vertex: 2
Enter the destination vertex: 3
Enter the weight of the edge: 4
Edge (2, 3) with weight 4 added to the graph.

Menu:
1. Add an Edge
2. Find Minimum Spanning Tree (Kruskal's Algorithm)
```

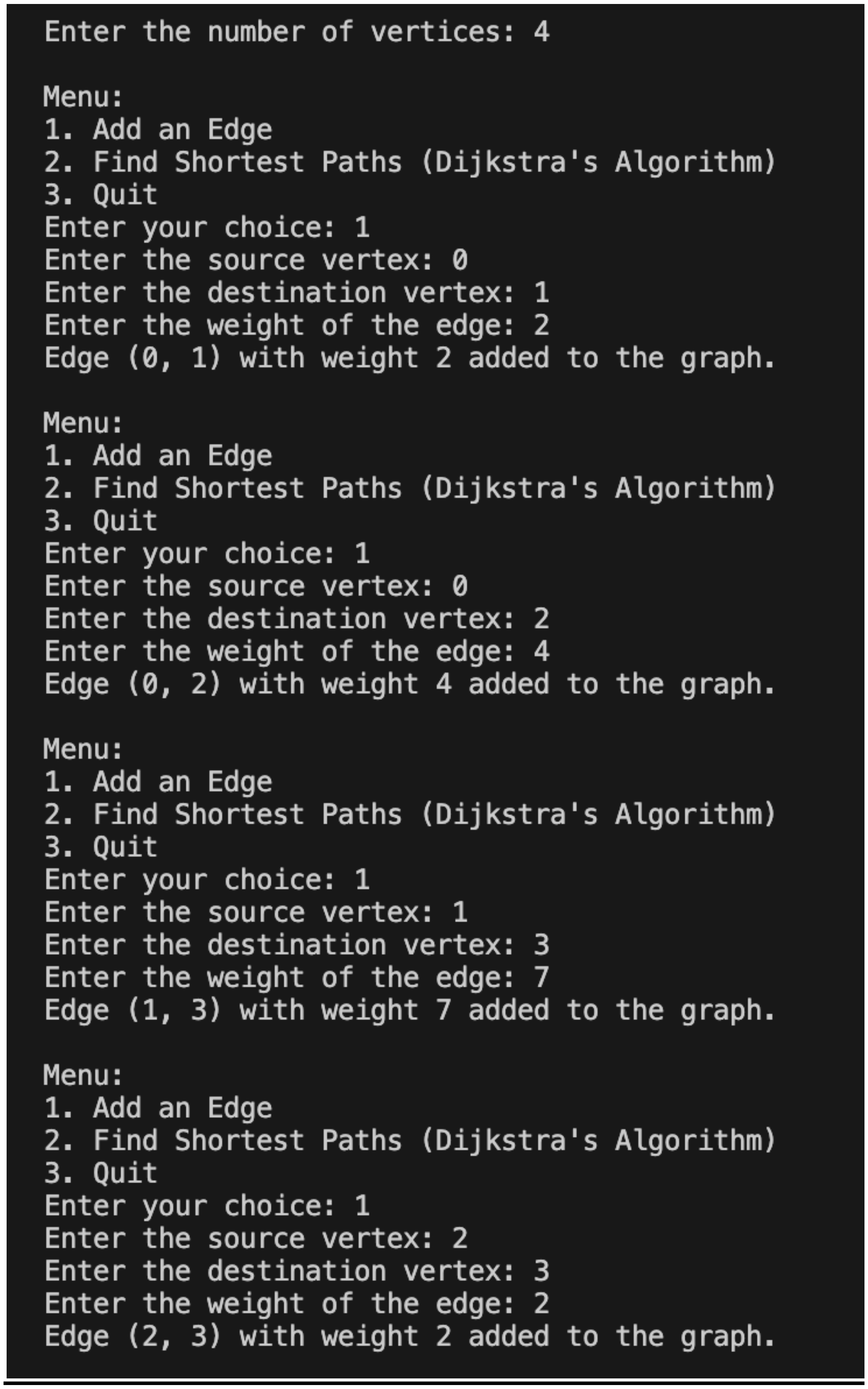


```
Menu:
1. Add an Edge
2. Find Minimum Spanning Tree (Kruskal's Algorithm)
3. Quit
Enter your choice: 2

Edges in Minimum Spanning Tree:
2 - 3 : 4
0 - 3 : 5
0 - 1 : 10

Menu:
1. Add an Edge
2. Find Minimum Spanning Tree (Kruskal's Algorithm)
3. Quit
Enter your choice: 3
Exiting the program. Goodbye!
```

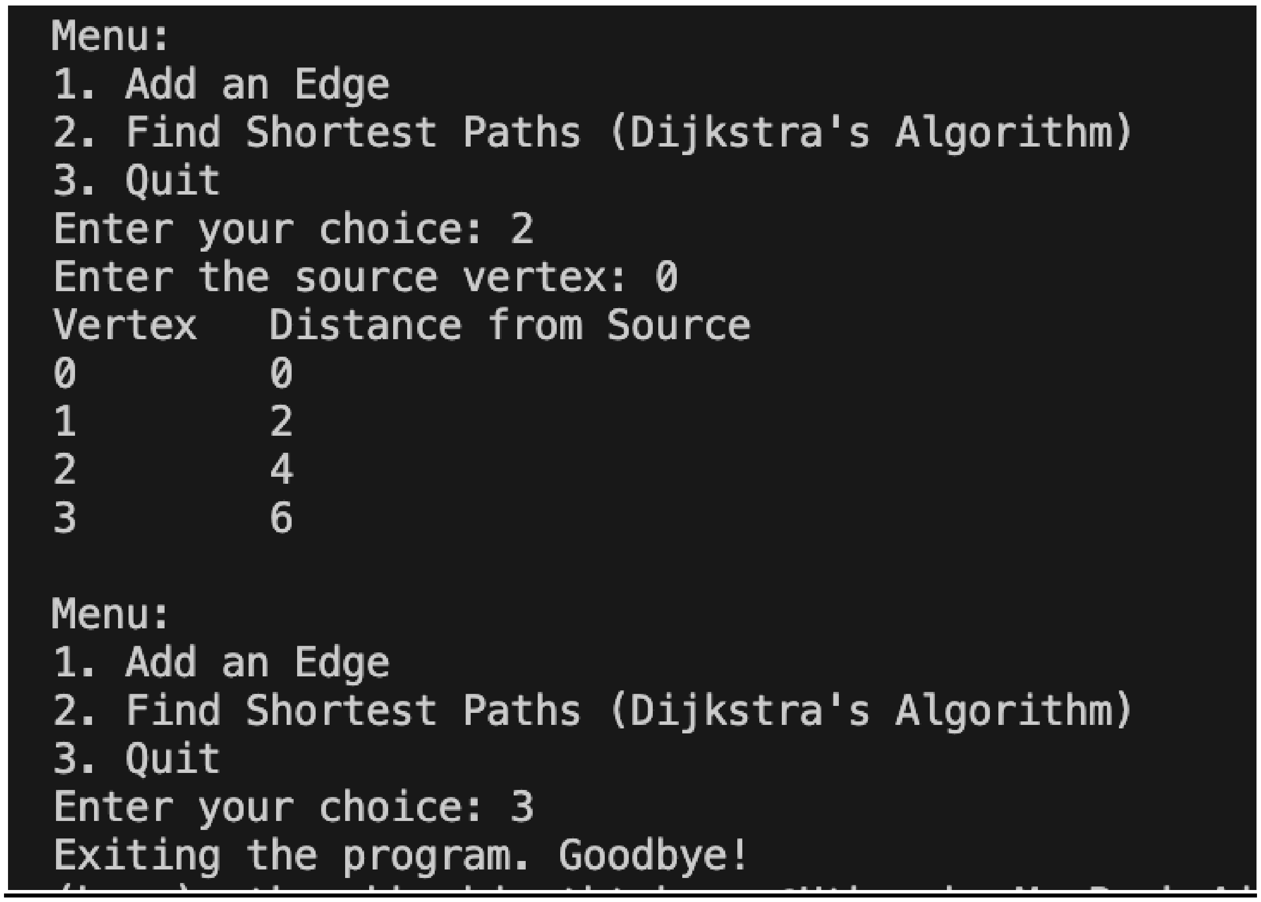
Dijkstra's Minimal Spanning Tree Algorithm

```
import sys


class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for _ in range(vertices)] for _ in range(vertices)]

    def add_edge(self, u, v, w):
        self.graph[u][v] = w
        self.graph[v][u] = w  # If the graph is undirected

    def dijkstra(self, src):
        visited = [False] * self.V
        distance = [sys.maxsize] * self.V
        distance[src] = 0

        for _ in range(self.V):
            u = self.min_distance(distance, visited)
            visited[u] = True

            for v in range(self.V):
                if (
                    self.graph[u][v] > 0
                    and visited[v] is False
                    and distance[v] > distance[u] + self.graph[u][v]
                ):
                    distance[v] = distance[u] + self.graph[u][v]
        self.print_solution(distance)

    def min_distance(self, distance, visited):
        min_dist = sys.maxsize
        min_index = 0

        for v in range(self.V):
            if distance[v] < min_dist and not visited[v]:
                min_dist = distance[v]
                min_index = v

        return min_index

    def print_solution(self, distance):
        print("Vertex \t Distance from Source")
        for node in range(self.V):
            print(f"{node} \t {distance[node]}")


def display_menu():
    print("\nMenu:")
    print("1. Add an Edge")
    print("2. Find Shortest Paths (Dijkstra's Algorithm)")
    print("3. Quit")

def add_edge(graph):
    u = int(input("Enter the source vertex: "))
    v = int(input("Enter the destination vertex: "))
    w = int(input("Enter the weight of the edge: "))
    graph.add_edge(u, v, w)
    print(f"Edge ({u}, {v}) with weight {w} added to the graph.")


def find_shortest_paths(graph):
    source = int(input("Enter the source vertex: "))
    graph.dijkstra(source)


def main():
    while True:
        display_menu()
        choice = input("Enter your choice: ")

        if choice == '1':
            add_edge(g)
        elif choice == '2':
            find_shortest_paths(g)
        elif choice == '3':
            print("Exiting the program. Goodbye!")
            break
        else:
            print("Invalid choice. Please select a valid option.")
if __name__ == "__main__":
    vertices = int(input("Enter the number of vertices: "))
    g = Graph(vertices)

    main()
```

Output:



```
Enter the number of vertices: 4

Menu:
1. Add an Edge
2. Find Shortest Paths (Dijkstra's Algorithm)
3. Quit
Enter your choice: 1
Enter the source vertex: 0
Enter the destination vertex: 1
Enter the weight of the edge: 2
Edge (0, 1) with weight 2 added to the graph.

Menu:
1. Add an Edge
2. Find Shortest Paths (Dijkstra's Algorithm)
3. Quit
Enter your choice: 1
Enter the source vertex: 0
Enter the destination vertex: 2
Enter the weight of the edge: 4
Edge (0, 2) with weight 4 added to the graph.

Menu:
1. Add an Edge
2. Find Shortest Paths (Dijkstra's Algorithm)
3. Quit
Enter your choice: 1
Enter the source vertex: 1
Enter the destination vertex: 3
Enter the weight of the edge: 7
Edge (1, 3) with weight 7 added to the graph.

Menu:
1. Add an Edge
2. Find Shortest Paths (Dijkstra's Algorithm)
3. Quit
Enter your choice: 1
Enter the source vertex: 2
Enter the destination vertex: 3
Enter the weight of the edge: 2
Edge (2, 3) with weight 2 added to the graph.
```



```
Menu:
1. Add an Edge
2. Find Shortest Paths (Dijkstra's Algorithm)
3. Quit
Enter your choice: 2
Enter the source vertex: 0
Vertex   Distance from Source
0        0
1        2
2        4
3        6

Menu:
1. Add an Edge
2. Find Shortest Paths (Dijkstra's Algorithm)
3. Quit
Enter your choice: 3
Exiting the program. Goodbye!
```