```python
class MultistageGraph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v, cost):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append((v, cost))

    def find_shortest_path(self, start, end):
        # Initialize the distance dictionary
        dist = {node: float('inf') for node in self.graph}
        dist[start] = 0

        # Using dynamic programming to find the shortest path
        for i in range(len(self.graph)):
            for u in self.graph:
                if dist[u] != float('inf'):
                    for v, cost in self.graph[u]:
                        if dist.get(v, float('inf')) > dist[u] + cost:
                            dist[v] = dist[u] + cost

        # Check if the end vertex is reachable
        return dist.get(end, float('inf'))

def main():
    graph = MultistageGraph()

    print("Enter edges in the format 'start_vertex end_vertex cost'. Type 'done' to finish:")

    while True:
        user_input = input()
        if user_input.lower() == 'done':
            break
        try:
            u, v, cost = user_input.split()
            cost = int(cost)
            graph.add_edge(u, v, cost)
        except ValueError:
            print("Invalid input. Please enter in the correct format.")
```

```python
    start = input("Enter the start vertex: ")
    end = input("Enter the end vertex: ")

    shortest_path_cost = graph.find_shortest_path(start, end)

    if shortest_path_cost == float('inf'):
        print(f"No path found from {start} to {end}.")
    else:
        print(f"The shortest path cost from {start} to {end} is: {shortest_path_cost}")

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

```
E:\5thsem\DAA\practicals>forward.py
Enter edges in the format 'start_vertex end_vertex cost'. Type 'done' to finish:
1 2 5
1 3 2
2 4 3
2 6 3
3 4 6
3 5 5
3 6 8
4 7 1
4 8 4
5 7 6
5 8 2
6 7 6
6 8 2
7 9 7
8 9 3
done
Enter the start vertex: 1
Enter the end vertex: 9
The shortest path cost from 1 to 9 is: 12
```

```python
class MultistageGraph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v, cost):
        if v not in self.graph:
            self.graph[v] = []
        self.graph[v].append((u, cost))  # Reverse the direction for backward approach

    def find_shortest_path(self, start, end):
        # Initialize the distance dictionary
        dist = {node: float('inf') for node in self.graph}
        dist[end] = 0  # Start from the end vertex

        # Using dynamic programming to find the shortest path
        for i in range(len(self.graph)):
            for u in self.graph:
                if dist[u] != float('inf'):
                    for v, cost in self.graph[u]:
                        if dist.get(v, float('inf')) > dist[u] + cost:
                            dist[v] = dist[u] + cost

        # Check if the start vertex is reachable
        return dist.get(start, float('inf'))

def main():
    graph = MultistageGraph()

    print("Enter edges in the format 'start_vertex end_vertex cost'. Type 'done' to finish:")

    while True:
        user_input = input()
        if user_input.lower() == 'done':
            break
        try:
            u, v, cost = user_input.split()
            cost = int(cost)
            graph.add_edge(u, v, cost)
        except ValueError:
            print("Invalid input. Please enter in the correct format.")
```

```
    start = input("Enter the start vertex: ")
    end = input("Enter the end vertex: ")

    shortest_path_cost = graph.find_shortest_path(start, end)

    if shortest_path_cost == float('inf'):
        print(f"No path found from {start} to {end}.")
    else:
        print(f"The shortest path cost from {start} to {end} is: {shortest_path_cost}")

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

```
E:\5thsem\DAA\practicals>backward.py
Enter edges in the format 'start_vertex end_vertex cost'. Type 'done' to finish:
2 1 5
3 1 2
4 2 3
6 2 3
4 3 6
5 3 5
6 3 8
7 4 1
8 4 4
7 5 6
8 5 2
7 6 6
8 6 2
9 7 7
9 8 3
done
Enter the start vertex: 9
Enter the end vertex: 1
The shortest path cost from 9 to 1 is: 12
```

```python
import sys

def tsp(graph, start_vertex):
    n = len(graph)
    memo = {}

    def dp(mask, pos):
        if mask == (1 << n) - 1:
            return graph[pos][start_vertex]  # Return to starting point

        if (mask, pos) in memo:
            return memo[(mask, pos)]

        ans = sys.maxsize

        for city in range(n):
            if mask & (1 << city) == 0:  # If city is not visited
                new_ans = graph[pos][city] + dp(mask | (1 << city), city)
                ans = min(ans, new_ans)

        memo[(mask, pos)] = ans
        return ans

    return dp(1 << start_vertex, start_vertex)

def main():
    print("Enter the distance matrix row by row (type 'done' when finished):")
    graph = []

    while True:
        line = input().strip()
        if line.lower() == "done":
            break
        # Convert the input line into a list of integers and append to the graph
        row = list(map(int, line.split()))
        graph.append(row)

    print("Enter the start vertex (0 to {}):".format(len(graph) - 1))
    start_vertex = int(input().strip())
```

```
    result = tsp(graph, start_vertex)
    print(f"The minimum cost of visiting all cities starting from vertex {start_vertex} is: {result}")

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

```
E:\5thsem\DAA\practicals>tsp.py
Enter the distance matrix row by row (type 'done' when finished):
0 10 15 20
5 0 9 10
6 13 0 12
8 8 9 0
done
Enter the start vertex (0 to 3):
0
The minimum cost of visiting all cities starting from vertex 0 is: 35
```

```
def matrix_chain_order(p):
    n = len(p) - 1  # Number of matrices
    m = [[0] * n for _ in range(n)]
    s = [[0] * n for _ in range(n)]  # s[i][j] stores the index of the split

    for length in range(2, n + 1):  # Length of the chain
        for i in range(n - length + 1):
            j = i + length - 1
            m[i][j] = float('inf')
            for k in range(i, j):
                q = m[i][k] + m[k + 1][j] + p[i] * p[k + 1] * p[j + 1]
                if q < m[i][j]:
                    m[i][j] = q
                    s[i][j] = k  # Record the index of the split

    return m, s

def print_optimal_parens(s, i, j):
    if i == j:
        print(f"A{i + 1}", end="")
    else:
        print("(", end="")
        print_optimal_parens(s, i, s[i][j])
        print_optimal_parens(s, s[i][j] + 1, j)
        print(")", end="")

def main():
    print("Enter the number of matrices:")
    n = int(input().strip())

    print("Enter the dimensions of matrices (as a space-separated list):")
    dimensions = list(map(int, input().strip().split()))

    if len(dimensions) != n + 1:
        print("Error: Number of dimensions should be one more than the number of matrices.")
        return

    m, s = matrix_chain_order(dimensions)

    print(f"The minimum number of multiplications is: {m[0][n - 1]}")
```

```
    print("Optimal parenthesization is: ", end="")
    print_optimal_parens(s, 0, n - 1)
    print()  # For a new line

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

```
E:\5thsem\DAA\practicals>matrix.py
Enter the number of matrices:
4
Enter the dimensions of matrices (as a space-separated list):
12 5 45 11 10
The minimum number of multiplications is: 3625
Optimal parenthesization is: (A1((A2A3)A4))
```

```python
def print_board(board):
    for row in board:
        print(" ".join(row))
    print()

def is_safe(board, row, col, n):
    # Check this column on upper side
    for i in range(row):
        if board[i][col] == 'Q':
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if j < 0:
            break
        if board[i][j] == 'Q':
            return False

    # Check upper diagonal on right side
    for i, j in zip(range(row, -1, -1), range(col, n)):
        if j >= n:
            break
        if board[i][j] == 'Q':
            return False

    return True

def solve_n_queens_util(board, row, n):
    if row >= n:
        print_board(board)  # Print the board configuration
        return True

    res = False
    for col in range(n):
        if is_safe(board, row, col, n):
            board[row][col] = 'Q'  # Place the queen
            res = solve_n_queens_util(board, row + 1, n) or res
            board[row][col] = '.'  # Backtrack

    return res
```

```
def solve_n_queens(n):
    board = [['.' for _ in range(n)] for _ in range(n)]  # Create an empty board
    if not solve_n_queens_util(board, 0, n):
        print("No solution exists")

def main():
    n = int(input("Enter the number of queens (N): "))
    solve_n_queens(n)

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

```
E:\5thsem\DAA\practicals>n_queen.py
Enter the number of queens (N): 4
. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .
```

```python
def is_safe(graph, color, vertex, c):
    # Check if the current color assignment is safe for vertex
    for neighbor in graph[vertex]:
        if color[neighbor] == c:
            return False
    return True

def graph_coloring_util(graph, m, color, vertex):
    # If all vertices are assigned a color then return true
    if vertex == len(graph):
        return True

    # Try different colors for vertex
    for c in range(1, m + 1):
        if is_safe(graph, color, vertex, c):
            color[vertex] = c  # Assign color c to vertex
            if graph_coloring_util(graph, m, color, vertex + 1):
                return True
            color[vertex] = 0  # Backtrack

    return False

def graph_coloring(graph, m):
    color = [0] * len(graph)  # Initialize color assignments
    if not graph_coloring_util(graph, m, color, 0):
        print("Solution does not exist")
        return False

    print("Solution exists: Following are the assigned colors:")
    for vertex in range(len(graph)):
        print(f"Vertex {vertex}: Color {color[vertex]}")
    return True

def main():
    n = int(input("Enter the number of vertices: "))
    graph = [[] for _ in range(n)]

    print("Enter the edges in the format 'u v' (one edge per line). Type 'done' to finish:")
    while True:
        edge = input()
```

```python
        if edge.lower() == 'done':
            break
        u, v = map(int, edge.split())
        graph[u].append(v)
        graph[v].append(u)  # Undirected graph

    m = int(input("Enter the number of colors: "))
    graph_coloring(graph, m)

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

```
E:\5thsem\DAA\practicals>coloring.py
Enter the number of vertices: 4
Enter the edges in the format 'u v' (one edge per line). Type 'done' to finish:
0 1
0 3
1 2
2 3
done
Enter the number of colors: 3
Solution exists: Following are the assigned colors:
Vertex 0: Color 1
Vertex 1: Color 2
Vertex 2: Color 1
Vertex 3: Color 2
```