Code:

```python
#   !pip install more-itertools==4.1.0
from itertools import permutations

def calculate_cycle_weight(graph, cycle):
    weight = 0
    n = len(cycle)
    for i in range(n):
        weight += graph[cycle[i]][cycle[(i + 1) % n]]
    return weight
def traveling_salesman(graph, start):
    num_vertices = len(graph)
    min_weight = float('inf')
    shortest_cycle = None
    for perm in permutations(range(num_vertices)):
        # starting node fixed
        if perm[0] == start-1:
            weight = calculate_cycle_weight(graph, perm)
            if weight < min_weight:
                min_weight = weight
                shortest_cycle = perm
    return shortest_cycle, min_weight

weighted_graph = [
    [0, 20, 42, 25],
    [20, 0, 30, 34],
    [42, 30, 0, 10],
    [25, 34, 10, 0]
]
start_city = 1
shortest_cycle, min_weight = traveling_salesman(weighted_graph,
start_city)
print('Shortest Hamiltonian Cycle is: ', end='')
for i in range(len(shortest_cycle)):
    print(shortest_cycle[i] + 1, "-->", end=' ')
print(shortest_cycle[0] + 1)
print("Minimum Weight:", min_weight)Output:
```

Output:

```
Shortest Hamiltonian Cycle is: 1 --> 2 --> 3 --> 4 --> 1
Minimum Weight: 85
```

Code:

```
def TowerOfHanoi(n, from_rod, to_rod, aux_rod):
     if n == 0:
          return
     TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
     print("Move disk", n, "from rod", from_rod, "to rod",
to_rod)
     TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)

N = 3
TowerOfHanoi(N, 'A', 'C', 'B')
```

Output:

```
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
```

Code:

```
from queue import Queue

def bfs(graph, start, visited):
    queue = Queue()
    queue.put(start)
    while not queue.empty():
        vertex = queue.get()
        if vertex not in visited:
            visited.add(vertex)
            print("'" + vertex + "', ", end='')
            for neighbor in graph[vertex]:
                if neighbor not in visited:
                    queue.put(neighbor)

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
print('BFS Traversal: ')
visited = set()  # Initialize the visited set
bfs(graph, 'A', visited)
# If the graph is disconnected
for key in graph:
    if key not in visited:
        bfs(graph, key, visited)
```

Output:

```
BFS Traversal:
'A', 'B', 'C', 'D', 'E', 'F'
```

Code:

```
def dfs(graph, node, visited):
  if node not in visited:
    visited.add(node)
    print("'" + node + "', " , end = '') # Print the current
node
    # Recursively explore all of the adjacent nodes.
    for neighbor in graph[node]:
      if neighbor not in visited:
        dfs(graph, neighbor, visited)

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
visited = set()
print("DFS Traversal:")
dfs(graph, 'A', visited)
# If the graph is disconnected
for node in graph:
    if node not in visited:
      dfs(graph, node, visited)
```

Output:

```
DFS Traversal:
'A', 'B', 'D', 'E', 'F', 'C'
```