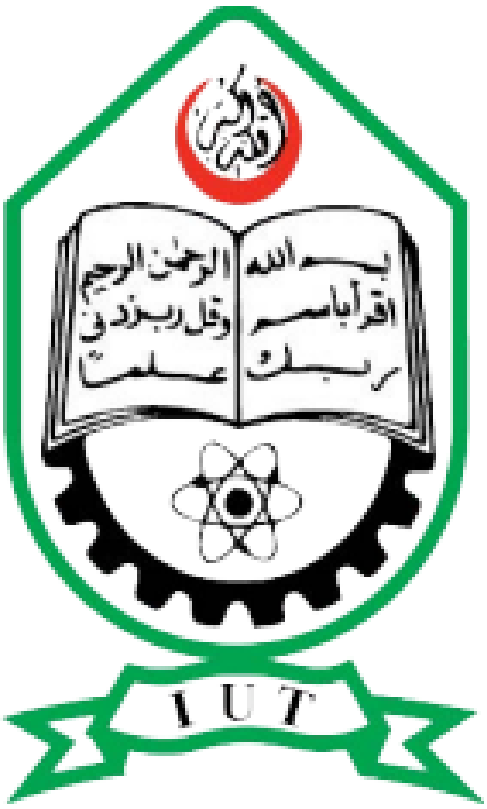

Islamic University of Technology

CSE 4810

Algorithm Engineering Lab

Lab 2

Tasnimul Hasnat

190041113

CSE 1A

March 17, 2024

Task 1

Djikstra

```
def dijkstra(G, source, dest, n):
  Q = set()
  D = [inf for _ in range(n + 1)]
  D[source] = 0
  Q.add((0, source))
  while len(Q):
    d, u = Q.pop()
    for v, w in G[u]:
      if D[u] + w < D[v]:
        D[v] = D[u] + w
        Q.add((D[v], v))
  return D[dest]
```

This is the implementation of Djikstra shortest path algorithm.

The **Time Complexity** of Djikstra shortest path algo is $O(V + E * \log V)$ where V is the number of vertices and E is the number of edges.

Bellman Ford

```
def bellmanford(edges, source, n):
  D = [inf for _ in range(n + 1)]
  D[source] = 0
  for step in range(n - 1):
    for edge in edges:
      u, v, w = edges
      D[v] = min(D[v], D[u] + w)
  return D
```

This is the implementation of Bellman Ford shortest path algorithm.

The **Time Complexity** of Bellman Ford shortest path alo is $O(V * E)$ where V is the number of vertices and E is the number of edges.

Task 2

```
def dijkstra(G, source, dest, n):
  Q = set()
  D = [inf for _ in range(n + 1)]
  D[source] = 0
  Q.add((0, source))
  while len(Q):
    d, u = Q.pop()
    for v, w in G[u]:
      if D[u] + w < D[v]:
        D[v] = D[u] + w
        Q.add((D[v], v))
    return D[dest]

n, m = map(int, input().split())
edges = []

for _ in range(m):
  i, j, k = map(int, input().split())
  edges.append((i, j, k))


def make_graph(n, edges, id):
  G = [[] for _ in range(n + 1)]
  for i, edge in enumerate(edges):
    if i == id: continue
      u, v, w = edge
      G[u].append((v, w))
      G[v].append((u, w))
  return G

minCycle = float("inf")
for i, edge in enumerate(edges):
  u, v, w = edge
  G = make_graph(n, edges, i)
  d = dijkstra(G, u, v, n)
  print(u, v, w, d)
  minCycle = min(minCycle, d + w)

print(minCycle)
```

I implemented Dijkstra's algorithm to find the shortest path from a source node to a destination node in a graph. It then iterates over all edges, removes one edge at a time, recalculates the shortest path, and prints the result. Finally, it finds the minimum cycle by considering the shortest path through all removed edges plus their weights.

Task 3

```
def floyd_warshall(graph, n):
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0
    for u in range(n):
        for v, w in graph[u]:
            dist[u][v] = min(dist[u][v], w)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] != float('inf') and dist[k][j] != float('inf'):
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist


def main():
    while True:
        n, m, q = map(int, input().split())
        if n == 0 and m == 0 and q == 0:
            break

        graph = [[] for _ in range(n)]

        for _ in range(m):
            u, v, w = map(int, input().split())
            graph[u].append((v, w))

        dist = floyd_warshall(graph, n)

        for _ in range(q):
            u, v = map(int, input().split())
            if dist[u][v] == float('inf'):
                print("Impossible")
            elif dist[u][v] == float('-inf'):
                print("-Infinity")
            else:
                print(dist[u][v])
        print()

main()
```

The code's time complexity is $O(m + n^3)$, where 'm' is the number of edges and 'n' is the number of nodes. It arises from graph building (O(m)) and Floyd-Warshall preprocessing (O(n^3)), with each query answered in constant time.