PYTHON ALGORITHM

그래프 Graph

21/03/19 발표자: 정의진

1.1. 그래프란?

- 노드(Node)와 노드 사이에 연결된 간선(Edges)의 정보를 가지고 있는 자료구조
- 서로 다른 개체가 연결되어 있다는 이야기를 들으면 그래프 알고리즘을 떠올려야 한다.

그래프 구현 방법

- 인접 행렬(Adjacency Matrix) : 2차원 배열을 사용하는 방식
  - Ex) 플로이드 워셜 알고리즘
- 인접 리스트(Adjacency List) : 리스트를 사용하는 방식
  - Ex) 다익스트라 최단 경로 알고리즘

최단 경로 알고리즘

- 다익스트라
- 플로이드 워셜
- 벨만 포드

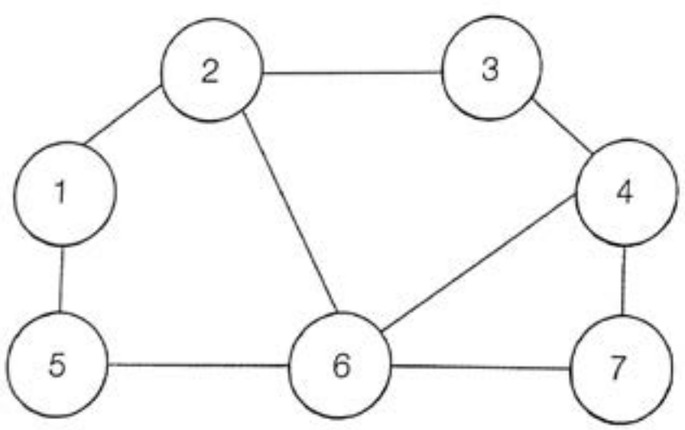
2.1. 최소신장트리

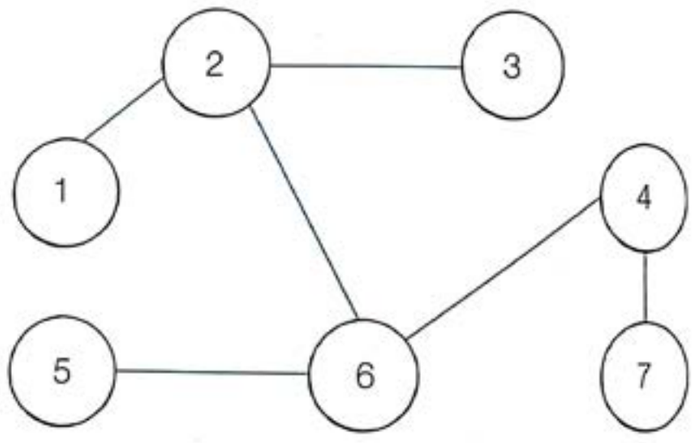
신장 트리(Spanning Tree)

: 하나의 그래프가 있을 때 모든 노드를 포함하면서 사이클이 존재하지 않는 부분 그래프

최소 신장 트리

- 신장트리중에서 최소 비용으로 만들 수 있는 신장트리
- 최소 신장 트리 알고리즘의 대표적인 예로 크루스칼 알고리즘이 있다.





가능한 신장 트리 예시

2.2. 크루스칼 알고리즘(Kruskal Algorithm)

- 신장트리(사이클이 없는 트리) 중에서 최소 비용으로 만들 수 있는 신장 트리를 찾는 '최소 신장 트리 알고리즘' 중 하나
- 그리디 알고리즘으로 분류됨

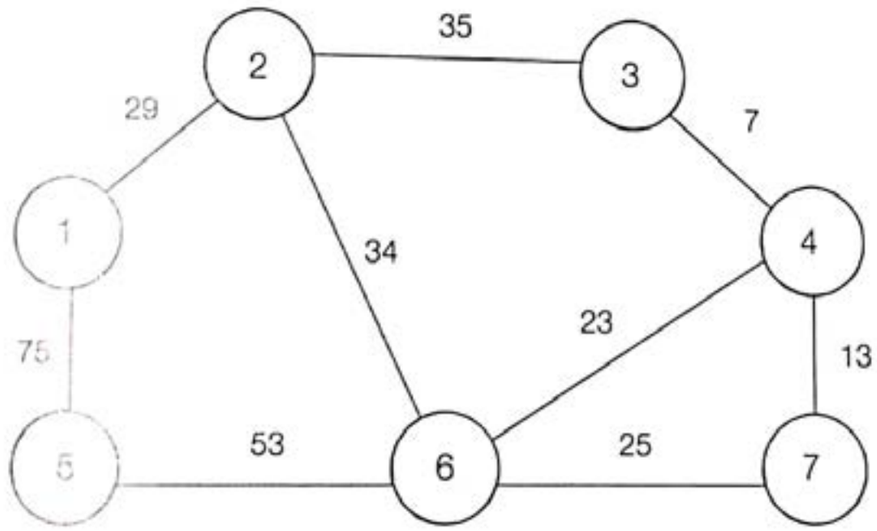
시간 복잡도

- O(ElogE)
- E: 간선의 개수

구현 방법

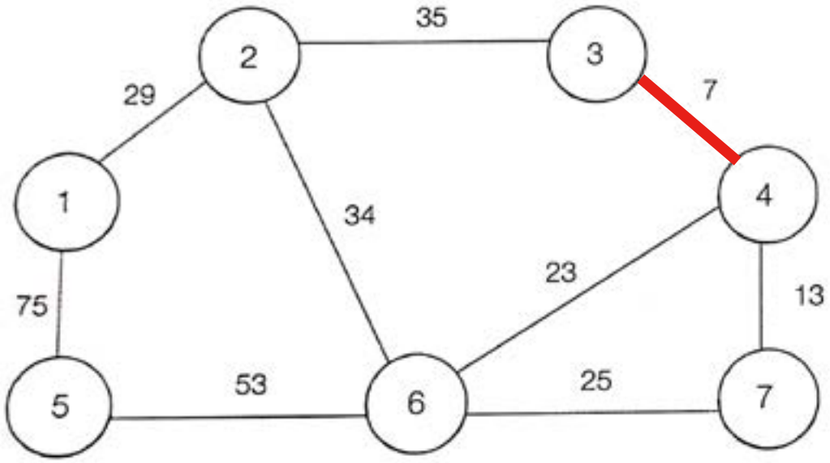
1. 간선 데이터를 비용에 따라 오름차순으로 정렬
2. 간선을 하나씩 확인하며 현재의 간선이 사이클을 발생시키는지 확인
    - 사이클이 발생하지 않는 경우 최소 신장 트리에 포함
    - 사이클이 발생하는 경우 최소 신장 트리에 포함X
3. 모든 간선에 대하여 2번 과정을 반복

2.2. 크루스칼 알고리즘(Kruskal Algorithm)



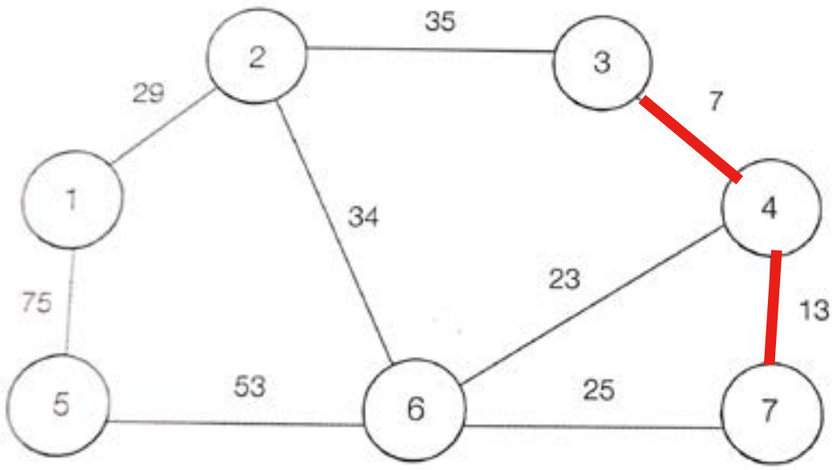
[illegible]	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
[illegible]	29	75	35	34	7	23	13	53	25

2.2. 크루스칼 알고리즘(Kruskal Algorithm)



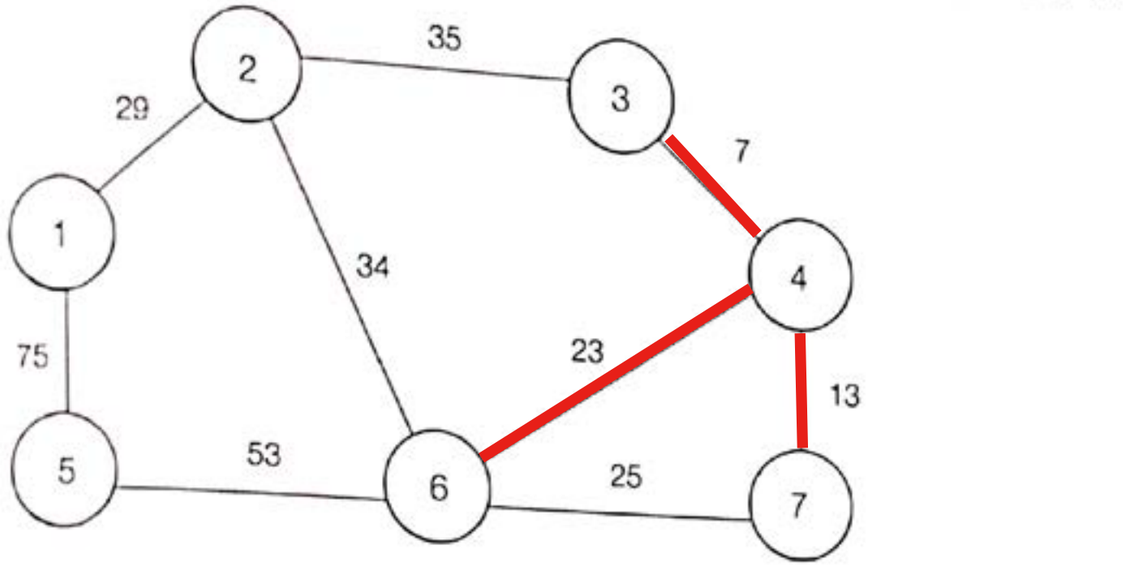
간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서					step 1				

2.2. 크루스칼 알고리즘(Kruskal Algorithm)



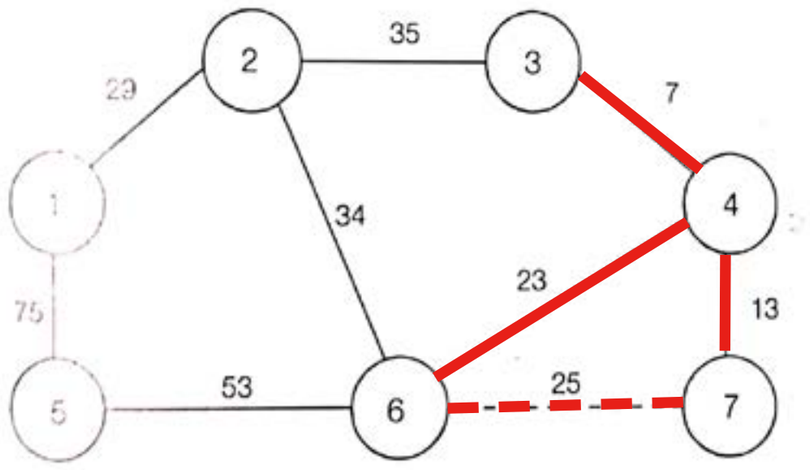
간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서					step 1		step 2		

2.2. 크루스칼 알고리즘(Kruskal Algorithm)



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서					step 1	step 3	step 2		

2.2. 크루스칼 알고리즘(Kruskal Algorithm)



간선	(1, 2)	(1, 5)	(2, 3)	(2, 6)	(3, 4)	(4, 6)	(4, 7)	(5, 6)	(6, 7)
비용	29	75	35	34	7	23	13	53	25
순서					step 1	step 3	step 2		step 4

2.3. 서로소 집합 자료구조

서로소 집합
: 공통 원소가 없는 집합

서로소 집합 자료구조
: 서로소 부분 집합들로 나누어진 원소들의 데이터를 처리하기 위한 자료구조

- union 과 find 연산으로 조작할 수 있다.
    - union : 2개의 원소가 포함된 집합을 하나의 집합으로 합치는 연산
    - find : 특정한 원소가 속한 집합이 어떤 집합인지를 알려주는 연산

서로소 집합을 활용한 사이클 판별
: 무방향 그래프 내에서의 사이클을 판별할 때 사용할 수 있다

2.3. 파이썬 코드 구현

```python
# 특정 원소가 속한 집합을 찾기
def find_parent(parent, x):
    # 루트 노드가 아니라면, 루트 노드를 찾을 때까지 재귀적으로 호출
    if parent[x] != x:
        parent[x] = find_parent(parent, parent[x])
    return parent[x]

# 두 원소가 속한 집합을 합치기
def union_parent(parent, a, b):
    a = find_parent(parent, a)
    b = find_parent(parent, b)
    if a < b:
        parent[b] = a
    else:
        parent[a] = b

# 노드의 개수와 간선(Union 연산)의 개수 입력 받기
v, e = map(int, input().split())
parent = [0] * (v + 1) # 부모 테이블 초기화하기

# 모든 간선을 담을 리스트와, 최종 비용을 담을 변수
edges = []
result = 0

# 부모 테이블상에서, 부모를 자기 자신으로 초기화
for i in range(1, v + 1):
    parent[i] = i

# 모든 간신에 대한 정보를 입력 받기
for _ in range(e):
    a, b, cost = map(int, input().split())
    # 비용순으로 정렬하기 위해서 튜플의 첫 번째 원소를 비용으로 설정
    edges.append((cost, a, b))

# 간신을 비용순으로 정렬
edges.sort()

# 간선을 하나씩 확인하며
for edge in edges:
    cost, a, b = edge
    # 사이클이 발생하지 않는 경우에만 집합에 포함
    if find_parent(parent, a) != find_parent(parent, b):
        union_parent(parent, a, b)
        result += cost

print(result)
```

3.1. 다익스트라 최단 경로 알고리즘

- 특정한 노드에서 출발하여 다른 노드로 가는 각각의 최단 경로를 구해주는 알고리즘
- 인접리스트를 이용하는 방식
- '각 노드에 대한 현재까지의 최단 거리' 정보를 리스트에 저장하고 계속 갱신한다.

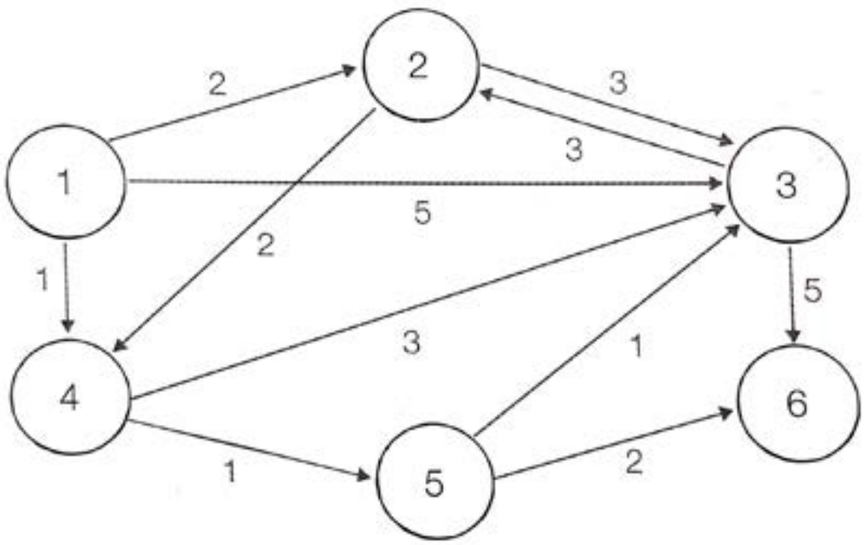
시간 복잡도

- $O(V^2)$
- V : 노드의 개수

구현 방법

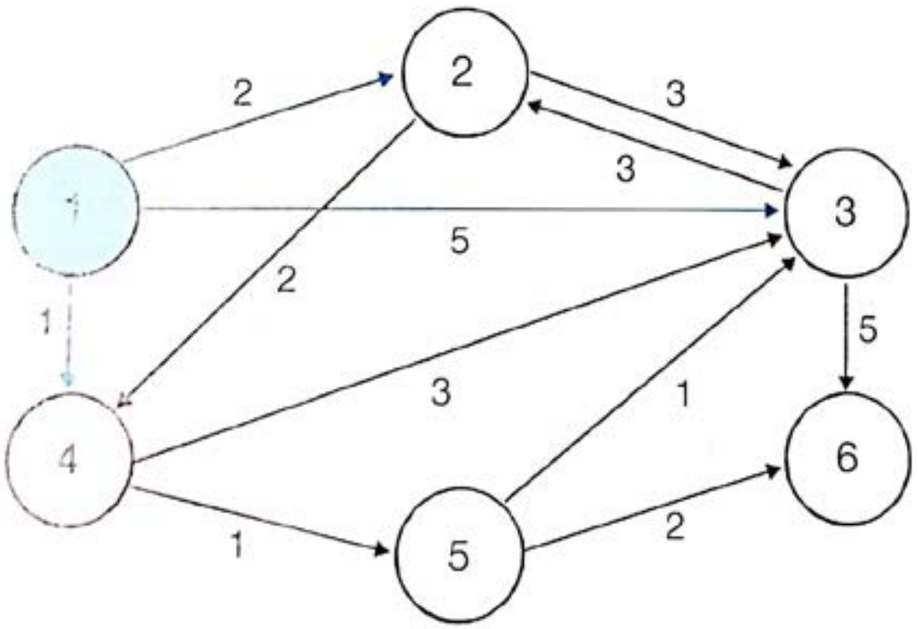
1. 출발 노드를 설정한다.
2. 최단 거리 테이블을 초기화한다.
3. 방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드를 선택한다.
4. 해당 노드를 거쳐 다른 노드로 가는 비용을 계산하여 최단 거리 테이블을 갱신한다.
5. 3번과 4번을 반복한다.

3.1. 다익스트라 최단 경로 알고리즘



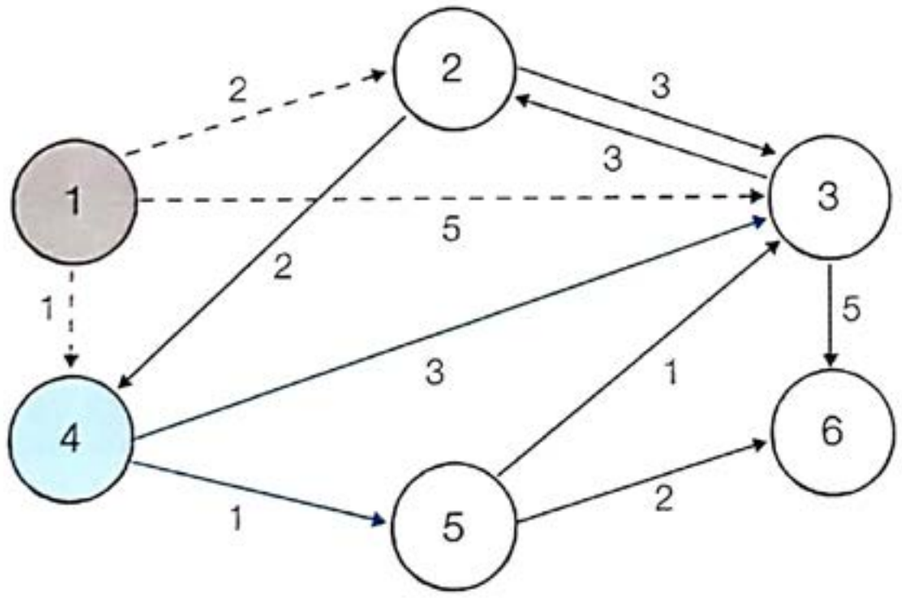
노드 번호	1	2	3	4	5	6
거리	0	무한	무한	무한	무한	무한

3.1. 다익스트라 최단 경로 알고리즘



노드 번호	1	2	3	4	5	6
거리	0	2	5	1	무한	무한

3.1. 다익스트라 최단 경로 알고리즘



노드 번호	1	2	3	4	5	6
거리	0	2	4	1	2	무한

3.2. 파이썬 코드 구현

```python
import sys
input = sys.stdin.readline
INF = int(1e9) # 무한을 의미하는 값으로 10억을 설정

# 노드의 개수, 간선의 개수를 입력받기
n, m = map(int, input().split())
# 시작 노드 번호를 입력받기
start = int(input())
# 각 노드에 연결되어 있는 노드에 대한 정보를 담는 리스트를 만들기
graph = [[] for i in range(n + 1)]
# 방문한 적이 있는지 체크하는 목적의 리스트를 만들기
visited = [False] * (n + 1)
# 최단 거리 테이블을 모두 무한으로 초기화
distance = [INF] * (n + 1)

# 모든 간선 정보를 입력받기
for _ in range(m):
    a, b, c = map(int, input().split())
    # a번 노드에서 b번 노드로 가는 비용이 c라는 의미
    graph[a].append((b, c))

# 방문하지 않은 노드 중에서, 가장 최단 거리가 짧은 노드의 번호를 반환
def get_smallest_node():
    min_value = INF
    index = 0 # 가장 최단 거리가 짧은 노드(인덱스)
    for i in range(1, n + 1):
        if distance[i] < min_value and not visited[i]:
            min_value = distance[i]
            index = i
    return index

def dijkstra(start):
    # 시작 노드에 대해서 초기화
    distance[start] = 0
    visited[start] = True
    for j in graph[start]:
        distance[j[0]] = j[1]
    # 시작 노드를 제외한 전체 n - 1개의 노드에 대해 반복
    for i in range(n - 1):
        # 현재 최단 거리가 가장 짧은 노드를 꺼내서, 방문 처리
        now = get_smallest_node()
        visited[now] = True
        # 현재 노드와 연결된 다른 노드를 확인
        for j in graph[now]:
            cost = distance[now] + j[1]
            # 현재 노드를 거쳐서 다른 노드로 이동하는 거리가 더 짧은 경우
            if cost < distance[j[0]]:
                distance[j[0]] = cost

# 다익스트라 알고리즘을 수행
dijkstra(start)

# 모든 노드로 가기 위한 최단 거리를 출력
for i in range(1, n + 1):
    # 도달할 수 없는 경우, 무한(INFINITY)이라고 출력
    if distance[i] == INF:
        print("INFINITY")
    # 도달할 수 있는 경우 거리를 출력
    else:
        print(distance[i])
```

3.3. 개선된 다익스트라 알고리즘

- 힙Heap) 자료구조 이용 (우선순위큐)
- 최단거리가 가장 짧은 노드를 힙 자료구조를 이용하여 더 빠르게 찾을 수 있다

시간복잡도

- O(ElogV)
- V : 노드의 개수 E: 간선의 개수

3.4. 파이썬 코드 구현

```python
import heapq
import sys
input = sys.stdin.readline
INF = int(1e9) # 무한을 의미하는 값으로 10억을 설정

# 노드의 개수, 간선의 개수를 입력받기
n, m = map(int, input().split())
# 시작 노드 번호를 입력받기
start = int(input())
# 각 노드에 연결되어 있는 노드에 대한 정보를 담는 리스트를 만들기
graph = [[] for i in range(n + 1)]
# 최단 거리 테이블을 모두 무한으로 초기화
distance = [INF] * (n + 1)

# 모든 간선 정보를 입력받기
for _ in range(m):
    a, b, c = map(int, input().split())
    # a번 노드에서 b번 노드로 가는 비용이 c라는 의미
    graph[a].append((b, c))

def dijkstra(start):
    q = []
    # 시작 노드로 가기 위한 최단 경로는 0으로 설정하여, 큐에 삽입
    heapq.heappush(q, (0, start))
    distance[start] = 0
    while q: # 큐가 비어있지 않다면
        # 가장 최단 거리가 짧은 노드에 대한 정보 꺼내기
        dist, now = heapq.heappop(q)
        # 현재 노드가 이미 처리된 적이 있는 노드라면 무시
        if distance[now] < dist:
            continue
        # 현재 노드와 연결된 다른 인접한 노드들을 확인
        for i in graph[now]:
            cost = dist + i[1]
            # 현재 노드를 거쳐서, 다른 노드로 이동하는 거리가 더 짧은 경우
            if cost < distance[i[0]]:
                distance[i[0]] = cost
                heapq.heappush(q, (cost, i[0]))

# 다익스트라 알고리즘을 수행
dijkstra(start)

# 모든 노드로 가기 위한 최단 거리를 출력
for i in range(1, n + 1):
    # 도달할 수 없는 경우, 무한(INFINITY)이라고 출력
    if distance[i] == INF:
        print("INFINITY")
    # 도달할 수 있는 경우 거리를 출력
    else:
        print(distance[i])
```

4.1. 플로이드 와샬 알고리즘

- 모든 지점에서 다른 모든 지점까지의 최단 경로를 구해야 하는 경우에 사용하는 알고리즘
- 다익스트라 알고리즘과 마찬가지로 해당 노드를 거쳐가는 경로를 확인하며, 최단 거리 테이블을 갱신
- 단, 최단 거리를 갖는 노드를 찾을 필요가 없다는 점이 다르다.
- 2차원 리스트에 '최단 거리' 정보를 저장

시간복잡도
- $O(N^3)$
- N : 노드의 개수

4.2. 파이썬 코드 구현

```python
INF = int(1e9) # 무한을 의미하는 값으로 10억을 설정

# 노드의 개수 및 간선의 개수를 입력받기
n = int(input())
m = int(input())
# 2차원 리스트(그래프 표현)를 만들고, 모든 값을 무한으로 초기화
graph = [[INF] * (n + 1) for _ in range(n + 1)]

# 자기 자신에서 자기 자신으로 가는 비용은 0으로 초기화
for a in range(1, n + 1):
    for b in range(1, n + 1):
        if a == b:
            graph[a][b] = 0

# 각 간선에 대한 정보를 입력 받아, 그 값으로 초기화
for _ in range(m):
    # A에서 B로 가는 비용은 C라고 설정
    a, b, c = map(int, input().split())
    graph[a][b] = c

# 점화식에 따라 플로이드 워셜 알고리즘을 수행
for k in range(1, n + 1):
    for a in range(1, n + 1):
        for b in range(1, n + 1):
            graph[a][b] = min(graph[a][b], graph[a][k] + graph[k][b])

# 수행된 결과를 출력
for a in range(1, n + 1):
    for b in range(1, n + 1):
        # 도달할 수 없는 경우, 무한(INFINITY)이라고 출력
        if graph[a][b] == 1e9:
            print("INFINITY", end=" ")
        # 도달할 수 있는 경우 거리를 출력
        else:
            print(graph[a][b], end=" ")
    print()
```