다익스트라 알고리즘

윤혜구

다익스트라 알고리즘

윤혜구

Dijkstra Algorithm

그래프에서 꼭짓점 간의 최단 경로를 찾는 알고리즘

탄생

1956년, 네덜란드의 컴퓨터 과학자
에츠허르 비버 다익스트라 (Edsger Wybe Dijkstra)가 고안

활용 분야

> 네트워크 라우팅 프로토콜(IS-IS, OSPF 등)

> 인공 지능 분야의 균일 비용 탐색

시간 복잡도

$O(V^2)$ or $O(ElogV)$

Edsger Wybe Dijkstra
1930. 5. 11. ~ 2002. 8. 6.

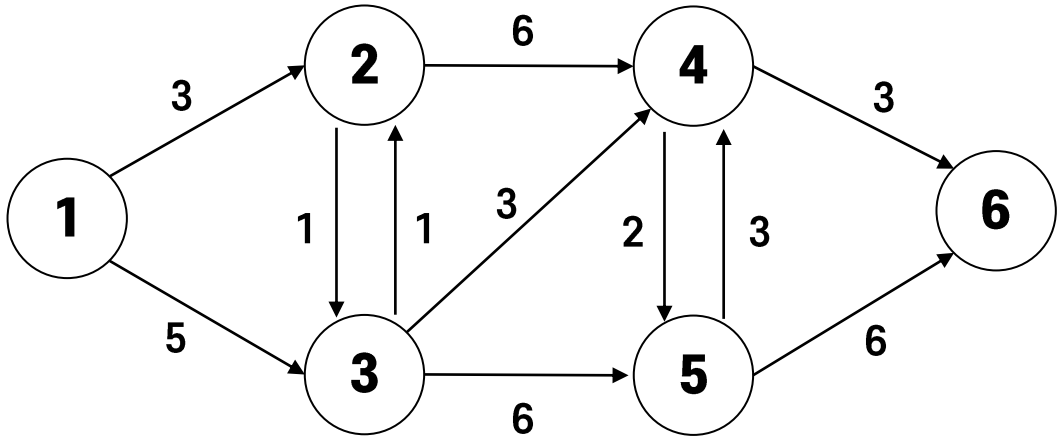
IDEA

1. 경로에 포함된 정점 집합 / 포함되지 않은 정점 집합으로 구분하여 관리
2. 탐욕적 방식의 경로 선택

    2-1. 갈 수 있는 정점들 중 아직 경로에 포함되지 않은 정점들의 (출발점으로부터의) 최단 거리를 계산하여 갱신

    2-2.아직 선택되지 않은 정점들 중 출발지점으로부터의 거리가 가장 작은 값을 가진 정점을 다음 정점으로 선택

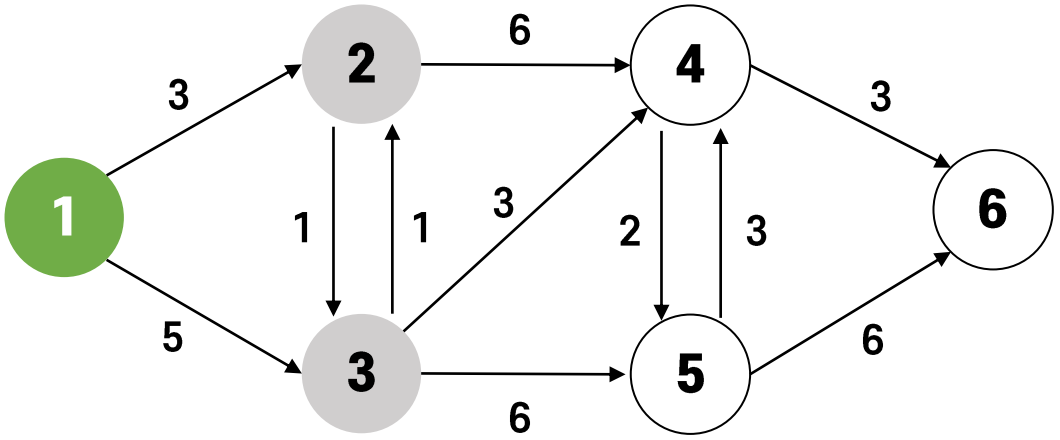
IDEA



정점	1	2	3	4	5	6
거리	0	∞	∞	∞	∞	∞

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

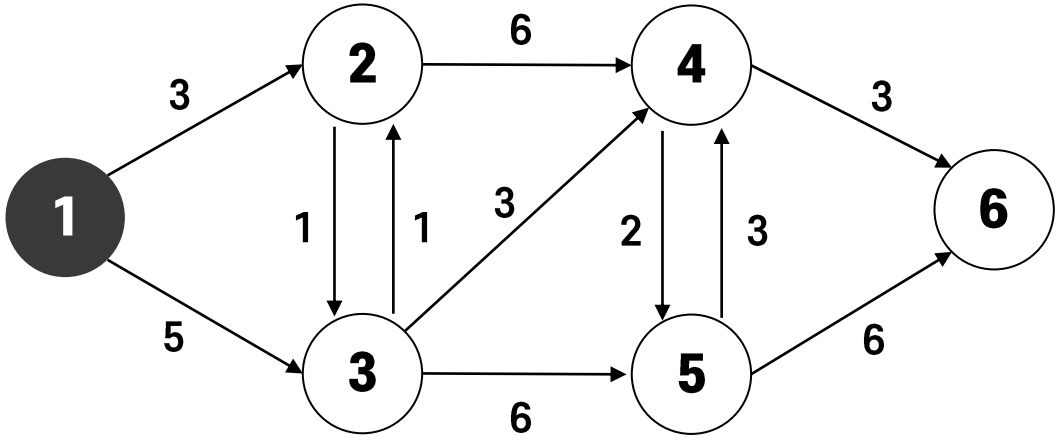
IDEA



정점	1	2	3	4	5	6
거리	0	3	5	∞	∞	∞

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

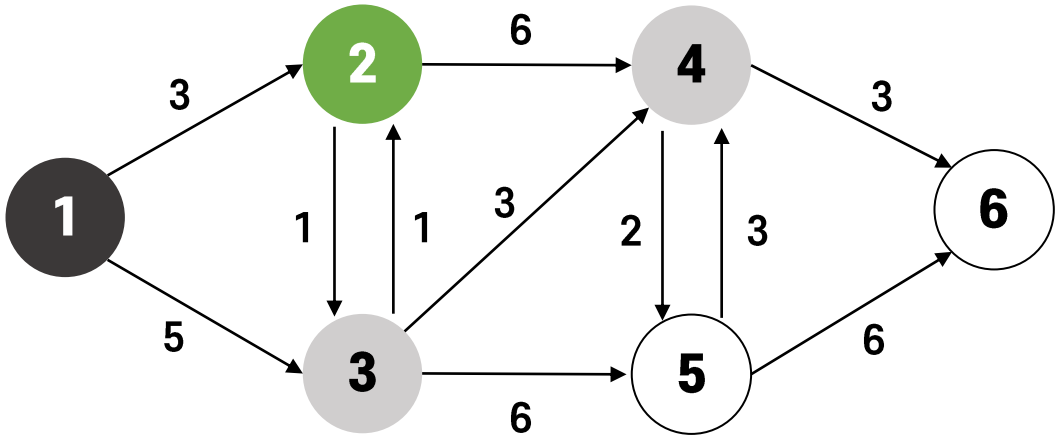
IDEA



정점	1	2	3	4	5	6
거리	0	3	5	∞	∞	∞

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



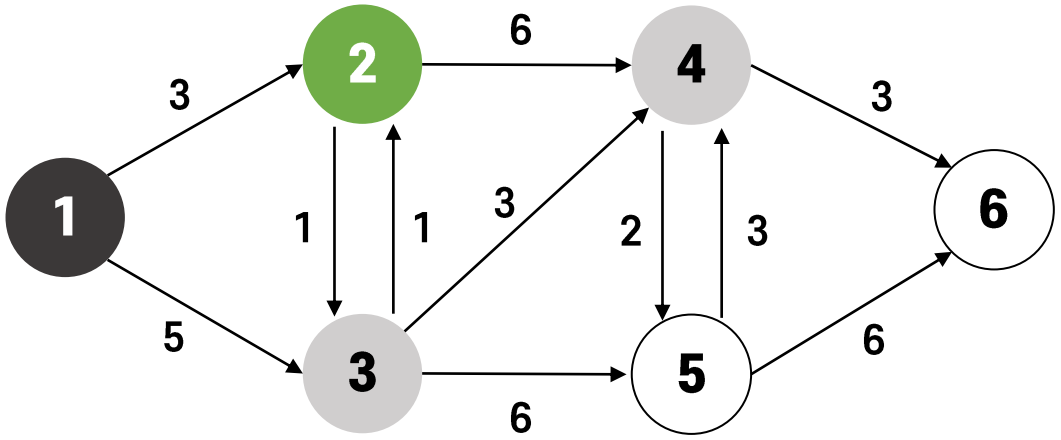
정점	1	2	3	4	5	6
거리	0	3	5	∞	∞	∞

▶ 5 (기존 거리)

▶ 4 (3 + 1 / 2번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



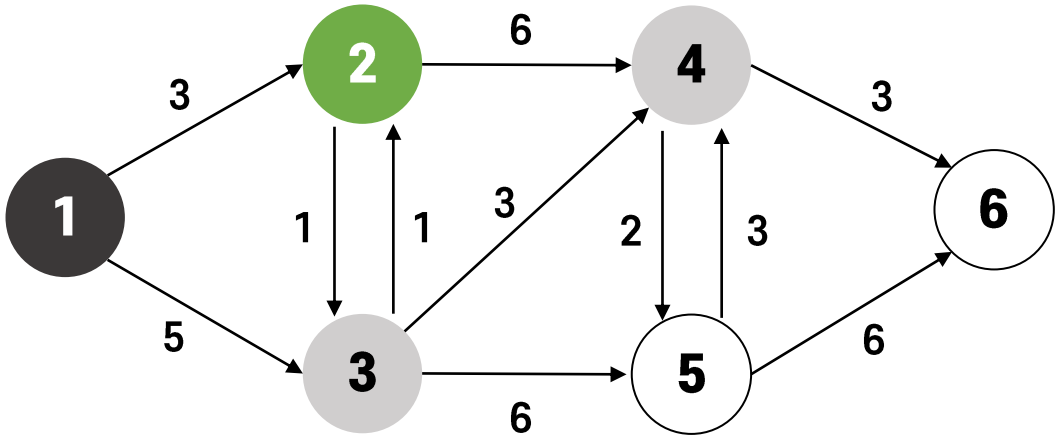
정점	1	2	3	4	5	6
거리	0	3	4	∞	∞	∞

▶ 5 (기존 거리)

▶ 4 (3 + 1 / 2번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



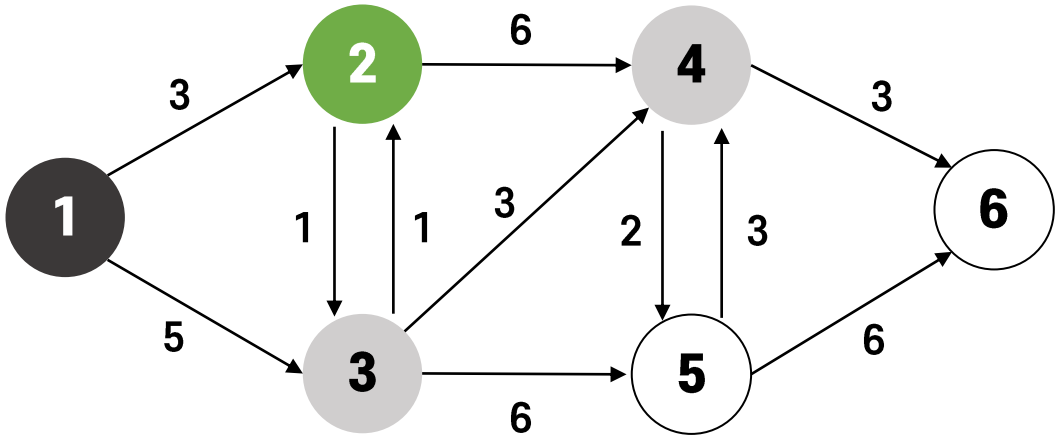
정점	1	2	3	4	5	6
거리	0	3	4	∞	∞	∞

▶ ∞ (기존 거리)

▶ 9 (3 + 6 / 2번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



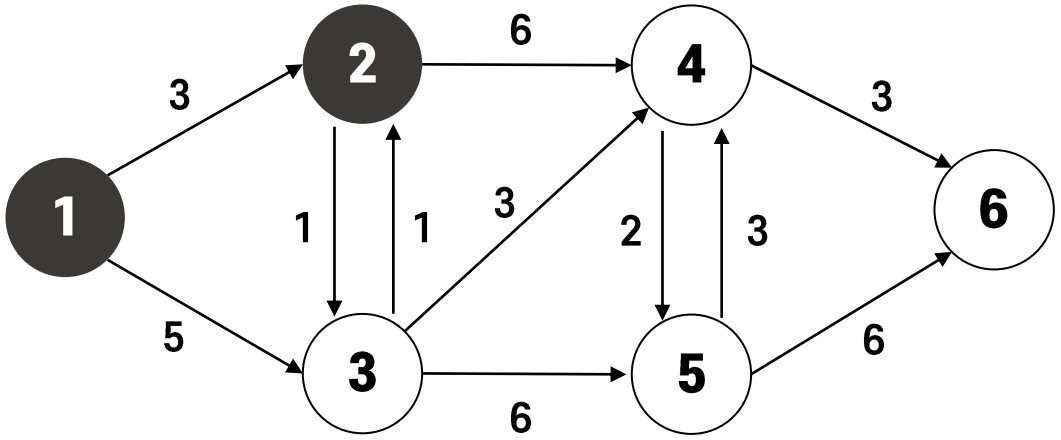
정점	1	2	3	4	5	6
거리	0	3	4	9	∞	∞

▶ ∞ (기존 거리)

▶ 9 (3 + 6 / 2번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

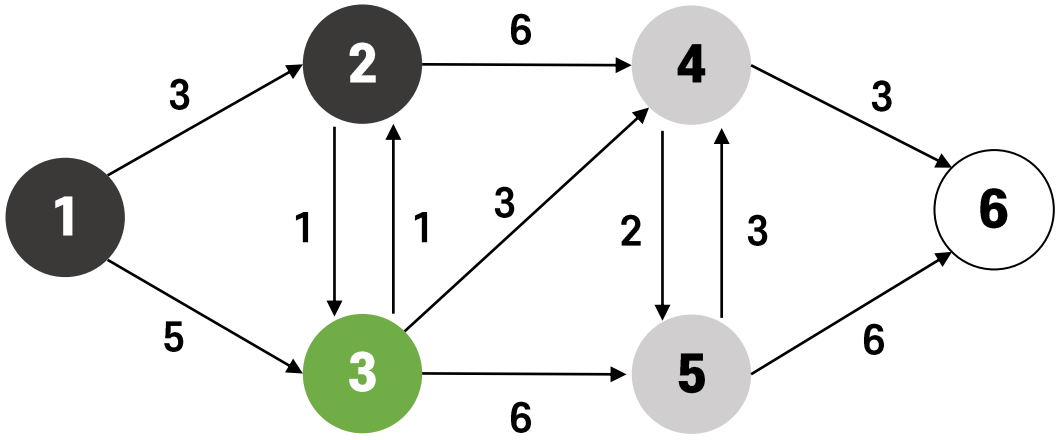
IDEA



정점	1	2	3	4	5	6
거리	0	3	4	9	∞	∞

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



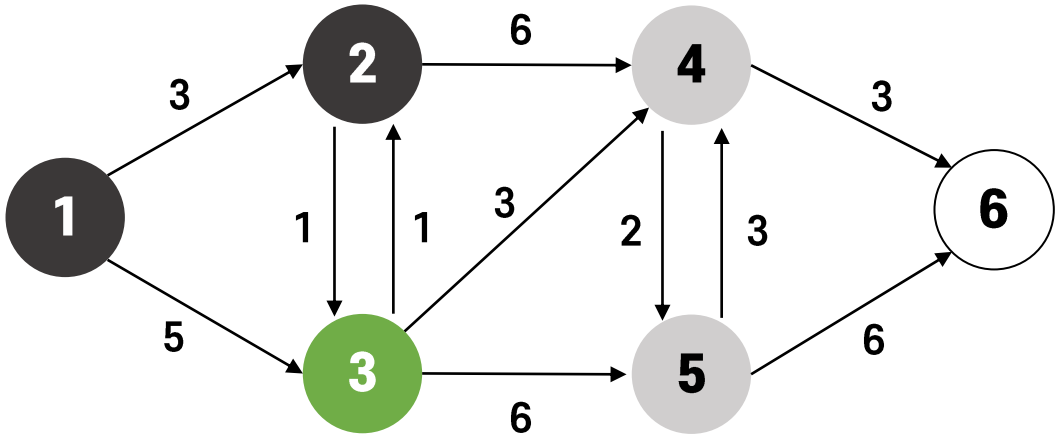
정점	1	2	3	4	5	6
거리	0	3	4	9	∞	∞

▶ 9 (기존 거리)

▶ 7 (4 + 3 / 3번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



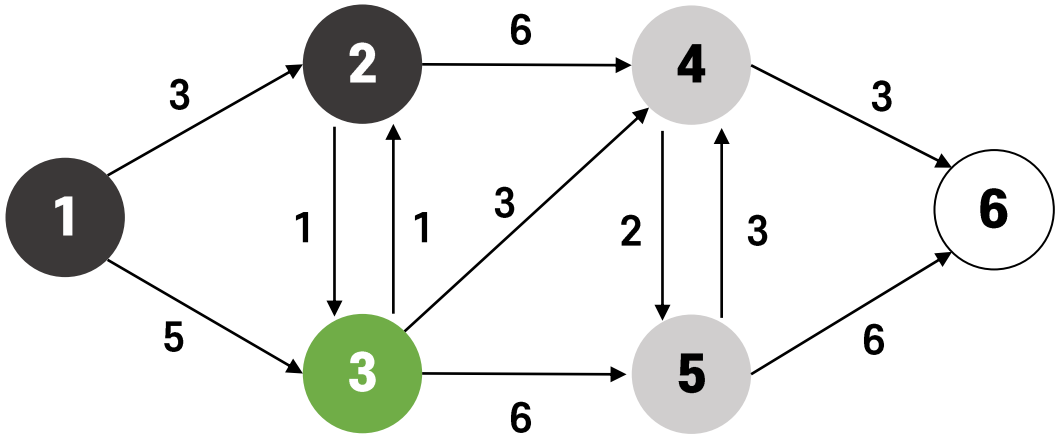
정점	1	2	3	4	5	6
거리	0	3	4	7	∞	∞

▶ 9 (기존 거리)

▶ 7 (4 + 3 / 3번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



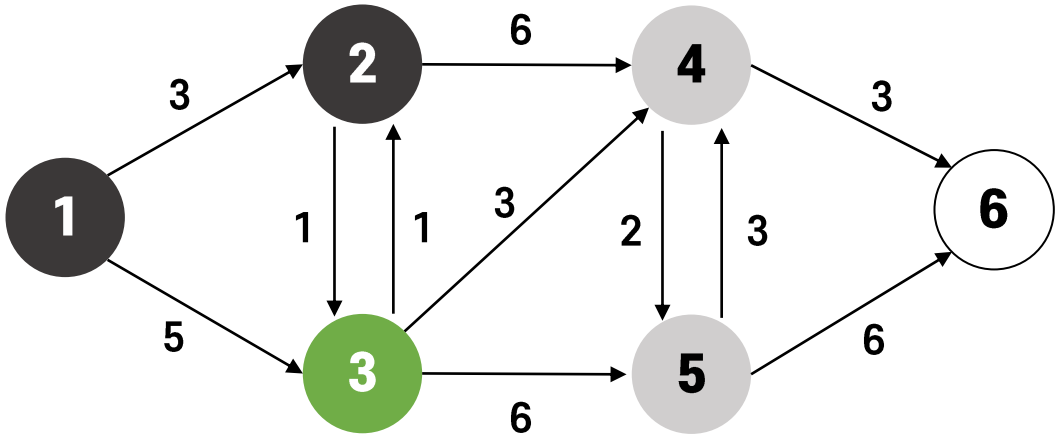
정점	1	2	3	4	5	6
거리	0	3	4	7	∞	∞

▶ ∞ (기존 거리)

▶ 10 (4 + 6 / 3번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



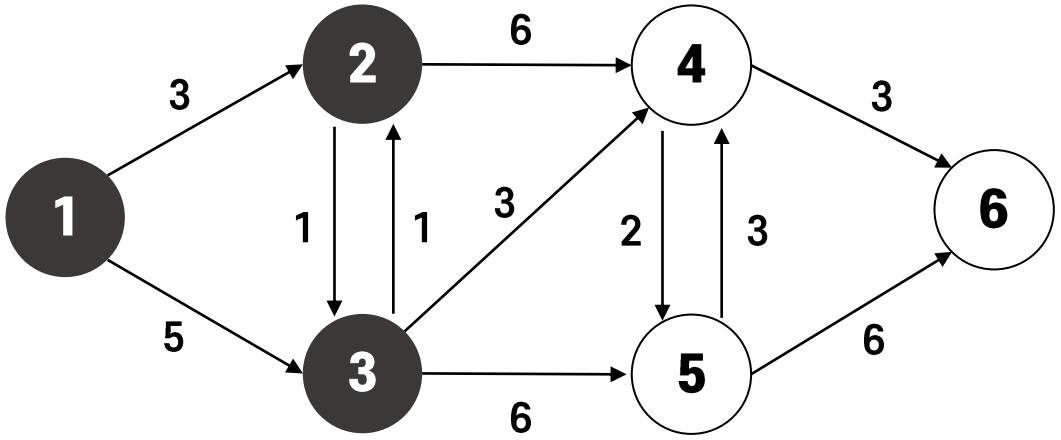
정점	1	2	3	4	5	6
거리	0	3	4	7	10	∞

▶ ∞ (기존 거리)

▶ 10 (4 + 6 / 3번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

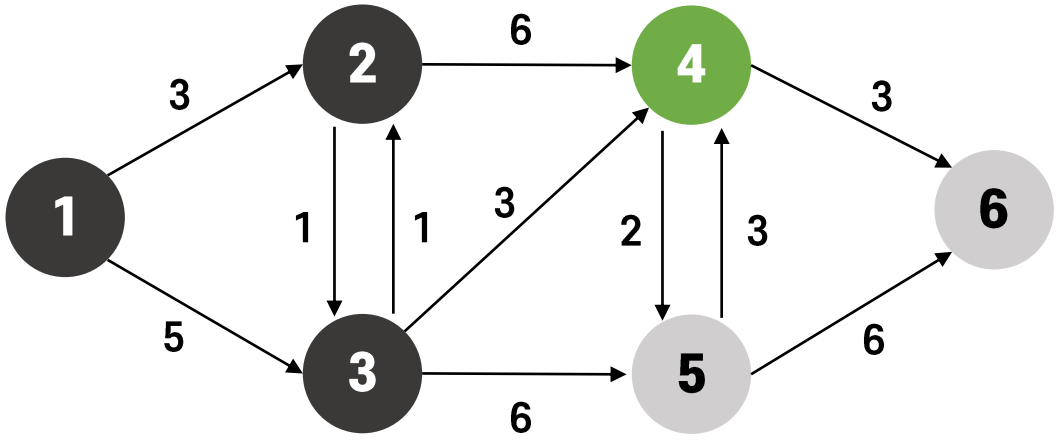
IDEA



정점	1	2	3	4	5	6
거리	0	3	4	7	10	∞

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



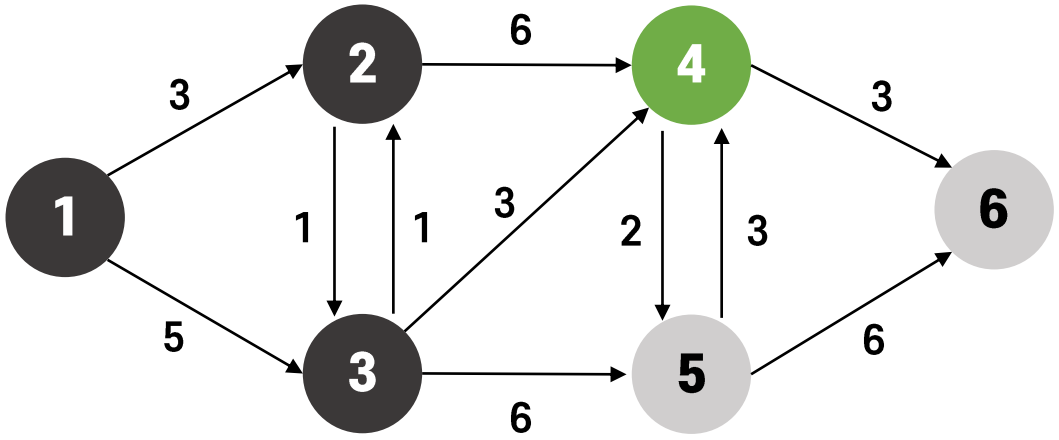
정점	1	2	3	4	5	6
거리	0	3	4	7	10	∞

▶ 10 (기존 거리)

▶ 9 (7 + 2 / 4번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



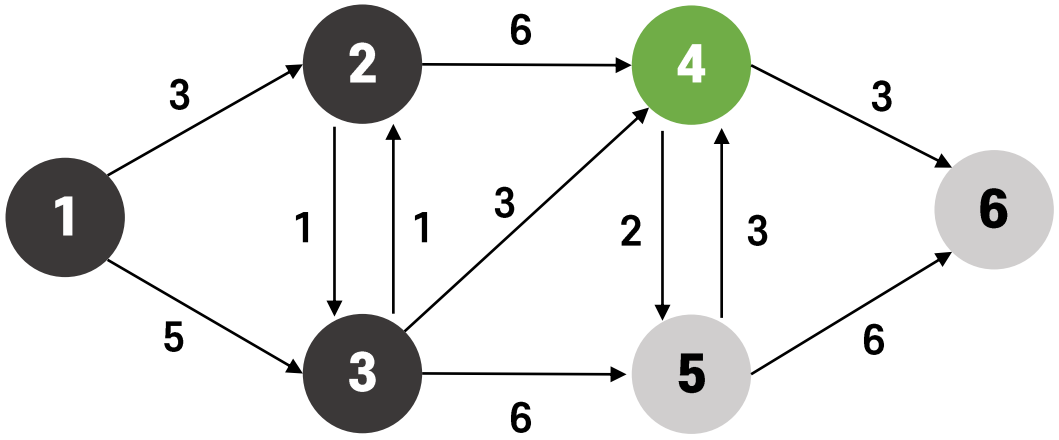
정점	1	2	3	4	5	6
거리	0	3	4	7	9	∞

▶ 10 (기존 거리)

▶ 9 (7 + 2 / 4번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



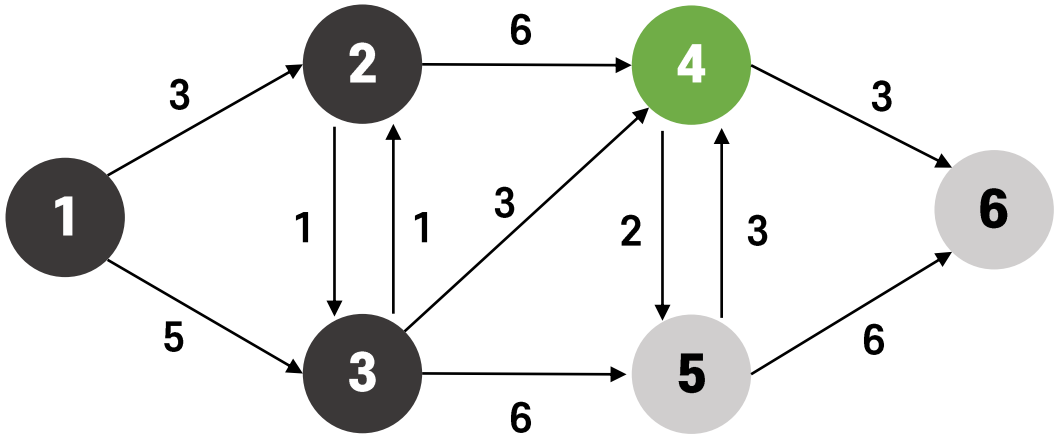
정점	1	2	3	4	5	6
거리	0	3	4	7	9	∞

▶ ∞ (기존 거리)

▶ 10 (7 + 3 / 4번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



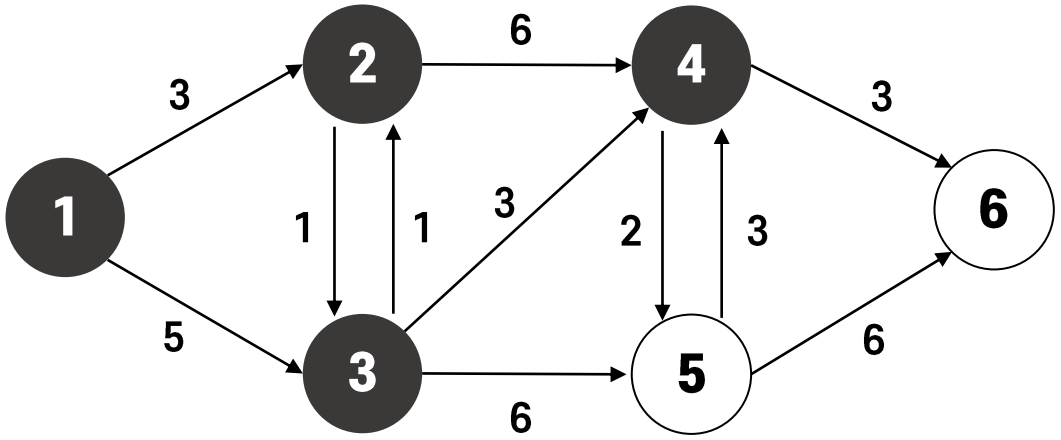
정점	1	2	3	4	5	6
거리	0	3	4	7	9	10

▶ ∞ (기존 거리)

▶ 10 (7 + 3 / 4번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

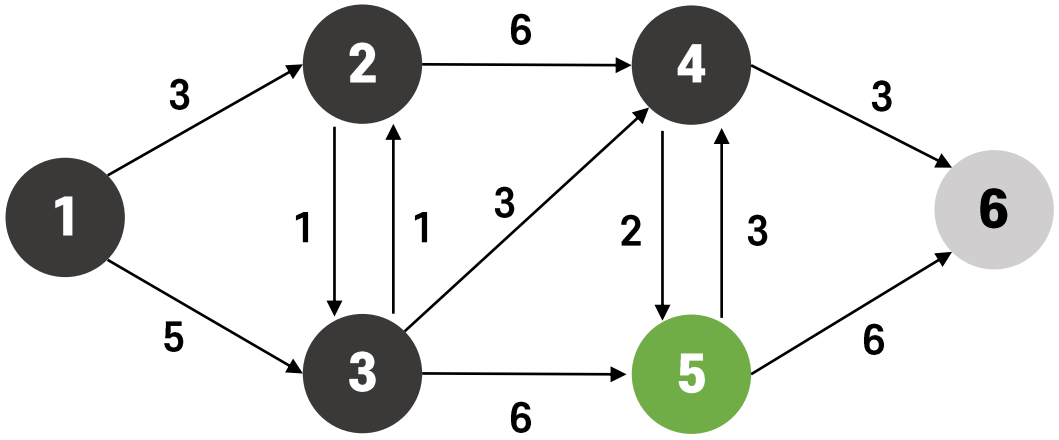
IDEA



정점	1	2	3	4	5	6
거리	0	3	4	7	9	10

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



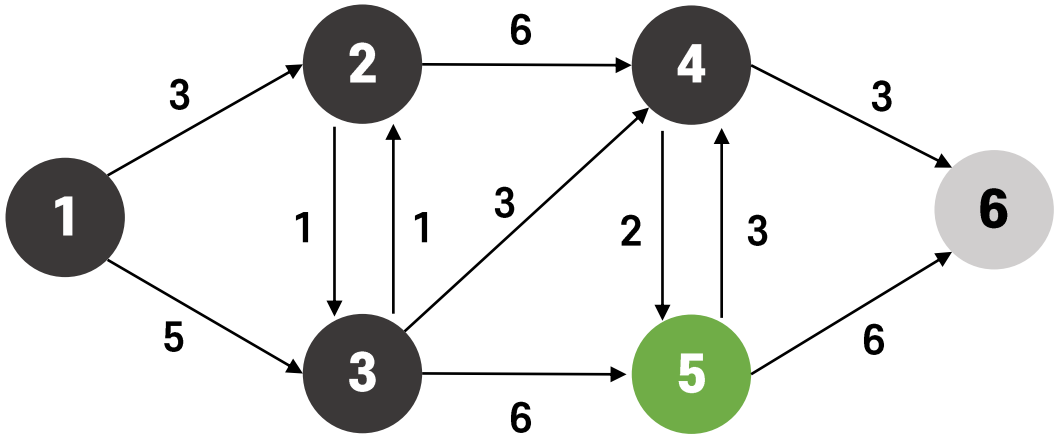
정점	1	2	3	4	5	6
거리	0	3	4	7	9	10

▶ 10 (기존 거리)

▶ 15 (9 + 6 / 5번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



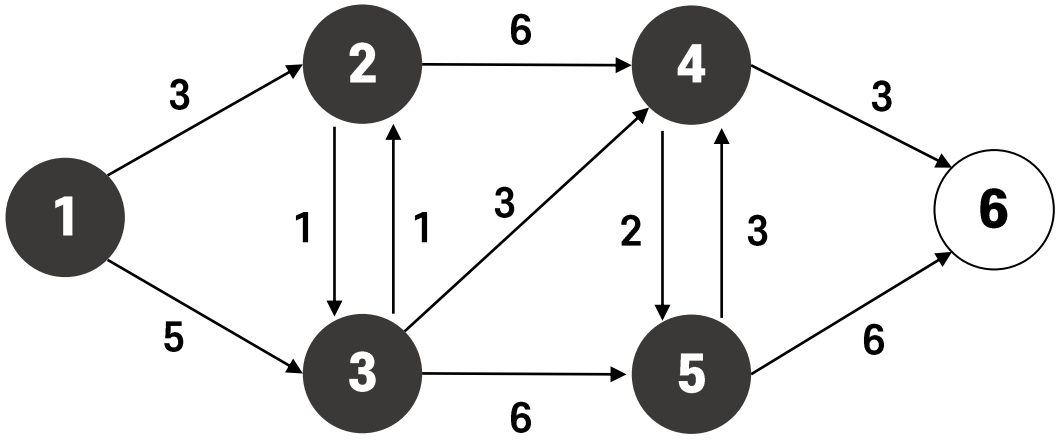
정점	1	2	3	4	5	6
거리	0	3	4	7	9	10

▶ 10 (기존 거리)

▶ 15 (9 + 6 / 5번을 거쳐 오는 거리)

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

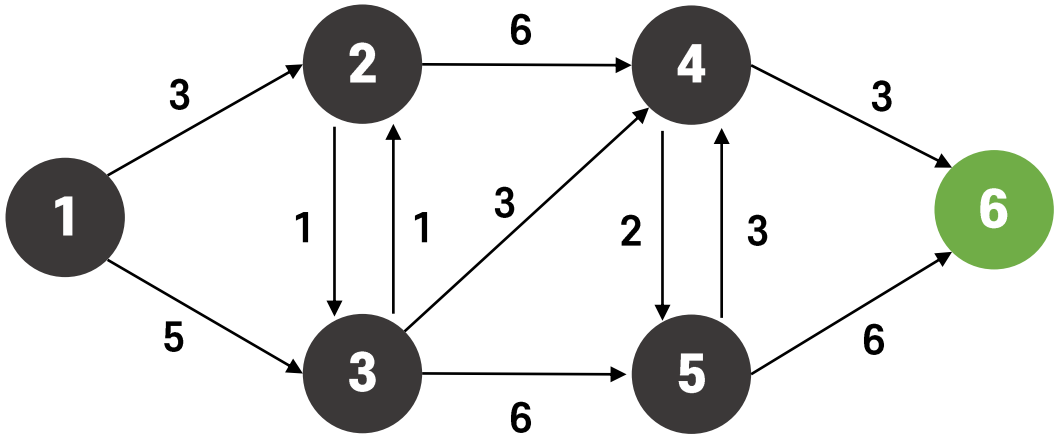
IDEA



정점	1	2	3	4	5	6
거리	0	3	4	7	9	10

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

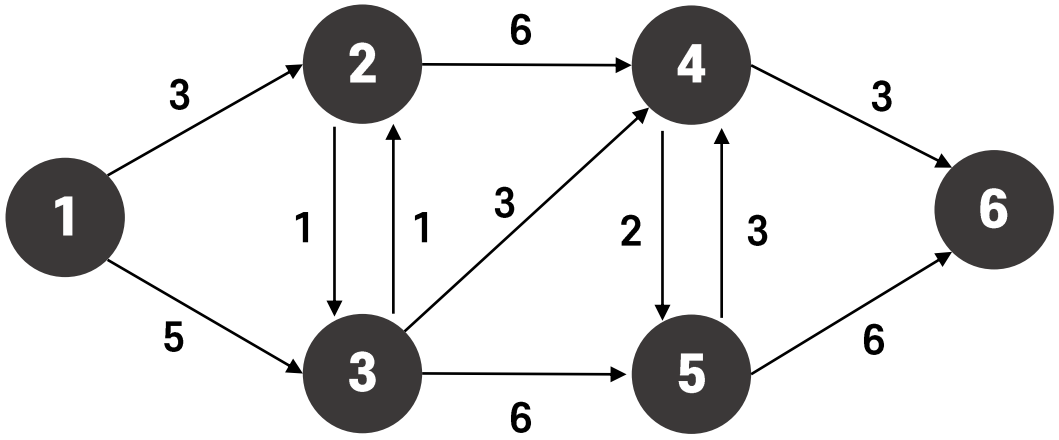
IDEA



정점	1	2	3	4	5	6
거리	0	3	4	7	9	10

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

IDEA



정점	1	2	3	4	5	6
거리	0	3	4	7	9	10

1 미방문 정점 2 방문한 정점 3 현재 정점 4 갱신 대상 정점

구현

인접행렬 방식

입력

```
5 11
0 1 3
0 2 5
1 2 2
1 3 6
2 1 1
2 3 4
2 4 6
3 4 2
3 5 3
4 0 3
4 5 6
```

출력

[0, 3, 5, 9, 11, 12]

```python
def dijkstra(s, V):
    U = [0]*(V+1)       # 비용이 결정된 정점을 표시
    U[s] = 1            # 출발점 비용 결정
    for i in range(V+1):
        D[i] = adjM[s][i]

    # 남은 정점의 비용 결정
    for _ in range(V):      # 남은 정점 개수만큼 반복
        # D[w]가 최소인 w 결정, 비용이 결정되지 않은 정점w 중에서
        minV = INF
        w = 0
        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                w = i

        U[w] = 1                # 비용 결정
        for v in range(V+1):
            if 0 < adjM[w][v] < INF:
                D[v] = min(D[v], D[w]+adjM[w][v])


INF = 10000
V, E = map(int, input().split())
adjM = [[INF]*(V+1) for _ in range(V+1)]

for i in range(V+1):
    adjM[i][i] = 0

for _ in range(E):
    u, v, w = map(int, input().split())
    adjM[u][v] = w

D = [0]*(V+1)
dijkstra(0, V)
print(D)
```

INF	무한대 (주어진 범위 이상의 큰 수) -> 거리가 INF == 방문하지 않음
V	정점 개수
E	간선 개수
adjM	간선 가중치 정보 저장한 인접행렬
u	출발 정점
v	도착 정점
w	가중치
D	최단거리 저장 배열
U	이미 포함된 정점인지 저장하는 배열(1/0)

```
    for _ in range(V):          # 남은 정점 개수만큼 반복
        # D[w]가 최소인 w 결정, 비용이 결정되지 않은 정점w 중에서
        minV = INF
        w = 0
        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                w = i

        U[w] = 1                # 비용 결정
        for v in range(V+1):
            if 0 < adjM[w][v] < INF:
                D[v] = min(D[v], D[w]+adjM[w][v])


INF = 10000
V, E = map(int, input().split())
adjM = [[INF]*(V+1) for _ in range(V+1)]

for i in range(V+1):
    adjM[i][i] = 0

for _ in range(E):
    u, v, w = map(int, input().split())
    adjM[u][v] = w

D = [0]*(V+1)
dijkstra(0, V)
print(D)
```

인접행렬 생성

1. 인접행렬 선언

	0	1	2	3	4	5
0	INF	INF	INF	INF	INF	INF
1	INF	INF	INF	INF	INF	INF
2	INF	INF	INF	INF	INF	INF
3	INF	INF	INF	INF	INF	INF
4	INF	INF	INF	INF	INF	INF
5	INF	INF	INF	INF	INF	INF

INF	무한대 (주어진 범위 이상의 큰 수) -> 거리가 INF == 방문하지 않음
V	정점 개수
E	간선 개수
adjM	간선 가중치 정보 저장한 인접행렬
u	출발 정점
v	도착 정점
w	가중치
D	최단거리 저장 배열
U	이미 포함된 정점인지 저장하는 배열(1/0)

```
    for _ in range(V):          # 남은 정점 개수만큼 반복
        # D[w]가 최소인 w 결정, 비용이 결정되지 않은 정점w 중에서
        minV = INF
        w = 0
        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                w = i

        U[w] = 1                # 비용 결정
        for v in range(V+1):
            if 0 < adjM[w][v] < INF:
                D[v] = min(D[v], D[w]+adjM[w][v])


INF = 10000
V, E = map(int, input().split())
adjM = [[INF]*(V+1) for _ in range(V+1)]

for i in range(V+1):
    adjM[i][i] = 0

for _ in range(E):
    u, v, w = map(int, input().split())
    adjM[u][v] = w

D = [0]*(V+1)
dijkstra(0, V)
print(D)
```

인접행렬 생성

2. 자기 자신과의 거리 0

	0	1	2	3	4	5
0	0	INF	INF	INF	INF	INF
1	INF	0	INF	INF	INF	INF
2	INF	INF	0	INF	INF	INF
3	INF	INF	INF	0	INF	INF
4	INF	INF	INF	INF	0	INF
5	INF	INF	INF	INF	INF	0

INF	무한대 (주어진 범위 이상의 큰 수) -> 거리가 INF == 방문하지 않음
V	정점 개수
E	간선 개수
adjM	간선 가중치 정보 저장한 인접행렬
u	출발 정점
v	도착 정점
w	가중치
D	최단거리 저장 배열
U	이미 포함된 정점인지 저장하는 배열(1/0)

```
    for _ in range(V):          # 남은 정점 개수만큼 반복
        # D[w]가 최소인 w 결정, 비용이 결정되지 않은 정점w 중에서
        minV = INF
        w = 0
        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                w = i

        U[w] = 1                # 비용 결정
        for v in range(V+1):
            if 0 < adjM[w][v] < INF:
                D[v] = min(D[v], D[w]+adjM[w][v])


INF = 10000
V, E = map(int, input().split())
adjM = [[INF]*(V+1) for _ in range(V+1)]

for i in range(V+1):
    adjM[i][i] = 0

for _ in range(E):
    u, v, w = map(int, input().split())
    adjM[u][v] = w

D = [0]*(V+1)
dijkstra(0, V)
print(D)
```

인접행렬 생성

3. u -> v 경로 거리 정보 입력 및 저장

	0	1	2	3	4	5
0	0	3	5	INF	INF	INF
1	INF	0	2	6	INF	INF
2	INF	1	0	4	6	INF
3	INF	INF	INF	0	2	3
4	3	INF	INF	INF	0	6
5	INF	INF	INF	INF	INF	0

INF	무한대 (주어진 범위 이상의 큰 수) -> 거리가 INF == 방문하지 않음
V	정점 개수
E	간선 개수
adjM	간선 가중치 정보 저장한 인접행렬
u	출발 정점
v	도착 정점
w	가중치
D	최단거리 저장 배열
U	이미 포함된 정점인지 저장하는 배열(1/0)

```python
def dijkstra(s, V):
    U = [0]*(V+1)          # 비용이 결정된 정점을 표시
    U[s] = 1               # 출발점 비용 결정
    for i in range(V+1):
        D[i] = adjM[s][i]

    # 남은 정점의 비용 결정
    for _ in range(V):      # 남은 정점 개수만큼 반복
        # D[w]가 최소인 w 결정, 비용이 결정되지 않은 정점w 중에서
        minV = INF
        w = 0
        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                w = i

        U[w] = 1                # 비용 결정
        for v in range(V+1):
            if 0 < adjM[w][v] < INF:
                D[v] = min(D[v], D[w]+adjM[w][v])


INF = 10000
V, E = map(int, input().split())
adjM = [[INF]*(V+1) for _ in range(V+1)]
```

U, D 배열 초기화

1. U (방문 배열) 출발지점 방문처리
2. D (거리 배열) 각각 출발지로부터 바로 가는 거리로 초기화

INF	무한대 (주어진 범위 이상의 큰 수) -> 거리가 INF == 방문하지 않음
V	정점 개수
E	간선 개수
adjM	간선 가중치 정보 저장한 인접행렬
u	출발 정점
v	도착 정점
w	가중치
D	최단거리 저장 배열
U	이미 포함된 정점인지 저장하는 배열(1/0)

	0	**1**	**2**	**3**	**4**	**5**
0	0	3	5	INF	INF	INF
1	INF	0	2	6	INF	INF
2	INF	1	0	4	6	INF
3	INF	INF	INF	0	2	3
4	3	INF	INF	INF	0	6
5	INF	INF	INF	INF	INF	0

```python
def dijkstra(s, V):
    U = [0]*(V+1)       # 비용이 결정된 정점을 표시
    U[s] = 1            # 출발점 비용 결정
    for i in range(V+1):
        D[i] = adjM[s][i]

    # 남은 정점의 비용 결정
    for _ in range(V):      # 남은 정점 개수만큼 반복
        # D[w]가 최소인 w 결정, 비용이 결정되지 않은 정점w 중에서
        minV = INF
        w = 0
        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                w = i

        U[w] = 1                # 비용 결정
        for v in range(V+1):
            if 0 < adjM[w][v] < INF:
                D[v] = min(D[v], D[w]+adjM[w][v])


INF = 10000
V, E = map(int, input().split())
adjM = [[INF]*(V+1) for _ in range(V+1)]
```

다음 정점 구하기

다음으로 루트에 포함할 정점:

아직 방문하지 않은 정점들 중 출발지로부터 거리가 가장 짧은 정점

w: 해당 정점의 인덱스

minV: 해당정점의 거리

구해지면 해당 정점을 루트에 포함

INF	무한대 (주어진 범위 이상의 큰 수) -> 거리가 INF == 방문하지 않음
V	정점 개수
E	간선 개수
adjM	간선 가중치 정보 저장한 인접행렬
u	출발 정점
v	도착 정점
w	가중치
D	최단거리 저장 배열
U	이미 포함된 정점인지 저장하는 배열(1/0)

	0	**1**	**2**	**3**	**4**	**5**
0	0	3	5	INF	INF	INF
1	INF	0	2	6	INF	INF
2	INF	1	0	4	6	INF
3	INF	INF	INF	0	2	3
4	3	INF	INF	INF	0	6
5	INF	INF	INF	INF	INF	0

```python
def dijkstra(s, V):
    U = [0]*(V+1)       # 비용이 결정된 정점을 표시
    U[s] = 1            # 출발점 비용 결정
    for i in range(V+1):
        D[i] = adjM[s][i]

    # 남은 정점의 비용 결정
    for _ in range(V):      # 남은 정점 개수만큼 반복
        # D[w]가 최소인 w 결정, 비용이 결정되지 않은 정점w 중에서
        minV = INF
        w = 0
        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                w = i

        U[w] = 1                # 비용 결정
        for v in range(V+1):
            if 0 < adjM[w][v] < INF:
                D[v] = min(D[v], D[w]+adjM[w][v])


INF = 10000
V, E = map(int, input().split())
adjM = [[INF]*(V+1) for _ in range(V+1)]
```

최단거리 갱신하기

If 자기 자신이 아니고(>0),

방문할 수 없는 정점이 아니면 (<INF)

== 현재 위치에서 방문할 수 있는 정점이면:

기존의 최단 거리와, 현재 정점을 거쳐 가는 거리를 비교하여 더 작은 값으로 최단 거리 갱신

INF	무한대 (주어진 범위 이상의 큰 수) -> 거리가 INF == 방문하지 않음
V	정점 개수
E	간선 개수
adjM	간선 가중치 정보 저장한 인접행렬
u	출발 정점
v	도착 정점
w	가중치
D	최단거리 저장 배열
U	이미 포함된 정점인지 저장하는 배열(1/0)

	0	1	2	3	4	5
0	0	3	5	INF	INF	INF
1	INF	0	2	6	INF	INF
2	INF	1	0	4	6	INF
3	INF	INF	INF	0	2	3
4	3	INF	INF	INF	0	6
5	INF	INF	INF	INF	INF	0

구현

인접리스트 방식

입력

5 11
0 1 3
0 2 5
1 2 2
1 3 6
2 1 1
2 3 4
2 4 6
3 4 2
3 5 3
4 0 3
4 5 6

출력

[0, 3, 5, 9, 11, 12]

```python
def dijkstra(s, V):
    U = [0]*(V+1)       # 비용이 결정된 정점을 표시
    U[s] = 1            # 출발점 비용 결정
    D[s] = 0
    for v, w in adjL[s]:
        D[v] = w

    # 남은 정점의 비용 결정
    for _ in range(V):      # 남은 정점 개수만큼 반복
        # D[t]가 최소인 t 결정, 비용이 결정되지 않은 정점t 중에서
        minV = INF
        t = 0

        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                t = i

        U[t] = 1                # 비용 결정
        for v, w in adjL[t]:
                D[v] = min(D[v], D[t]+w)


INF = 10000
V, E = map(int, input().split())
adjL = [[] for _ in range(V+1)]

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

D = [INF]*(V+1)
dijkstra(0, V)
print(D)
```

구현

인접리스트

VS

인접행렬

```python
def dijkstra(s, V):
    U = [0]*(V+1)       # 비용이 결정된 정점을 표시
    U[s] = 1            # 출발점 비용 결정
    D[s] = 0
    for v, w in adjL[s]:
        D[v] = w

    # 남은 정점의 비용 결정
    for _ in range(V):      # 남은 정점 개수만큼 반복
        # D[t]가 최소인 t 결정, 비용이 결정되지 않은 정점t 중에서
        minV = INF
        t = 0

        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                t = i

        U[t] = 1                # 비용 결정
        for v, w in adjL[t]:
            D[v] = min(D[v], D[t]+w)


INF = 10000
V, E = map(int, input().split())
adjL = [[] for _ in range(V+1)]

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

D = [INF]*(V+1)
dijkstra(0, V)
print(D)
```

```python
def dijkstra(s, V):
    U = [0]*(V+1)       # 비용이 결정된 정점을 표시
    U[s] = 1            # 출발점 비용 결정
    for i in range(V+1):
        D[i] = adjM[s][i]

    # 남은 정점의 비용 결정
    for _ in range(V):      # 남은 정점 개수만큼 반복
        # D[w]가 최소인 w 결정, 비용이 결정되지 않은 정점w 중에서
        minV = INF
        w = 0
        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                w = i

        U[w] = 1                # 비용 결정
        for v in range(V+1):
            if 0 < adjM[w][v] < INF:
                D[v] = min(D[v], D[w]+adjM[w][v])


INF = 10000
V, E = map(int, input().split())
adjM = [[INF]*(V+1) for _ in range(V+1)]

for i in range(V+1):
    adjM[i][i] = 0

for _ in range(E):
    u, v, w = map(int, input().split())
    adjM[u][v] = w

D = [0]*(V+1)
dijkstra(0, V)
print(D)
```

구현

인접리스트

VS

인접행렬

시간 복잡도

$O(V^2)$

```python
def dijkstra(s, V):
    U = [0]*(V+1)       # 비용이 결정된 정점을 표시
    U[s] = 1            # 출발점 비용 결정
    D[s] = 0
    for v, w in adjL[s]:
        D[v] = w

    # 남은 정점의 비용 결정
    for _ in range(V):      # 남은 정점 개수만큼 반복
        # D[t]가 최소인 t 결정, 비용이 결정되지 않은 정점t 중에서
        minV = INF
        t = 0

        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                t = i

        U[t] = 1                # 비용 결정
        for v, w in adjL[t]:
                D[v] = min(D[v], D[t]+w)


INF = 10000
V, E = map(int, input().split())
adjL = [[] for _ in range(V+1)]

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

D = [INF]*(V+1)
dijkstra(0, V)
print(D)
```

```python
def dijkstra(s, V):
    U = [0]*(V+1)       # 비용이 결정된 정점을 표시
    U[s] = 1            # 출발점 비용 결정
    for i in range(V+1):
        D[i] = adjM[s][i]

    # 남은 정점의 비용 결정
    for _ in range(V):      # 남은 정점 개수만큼 반복
        # D[w]가 최소인 w 결정, 비용이 결정되지 않은 정점w 중에서
        minV = INF
        w = 0
        for i in range(V+1):
            if U[i] == 0 and minV > D[i]:
                minV = D[i]
                w = i

        U[w] = 1                # 비용 결정
        for v in range(V+1):
            if 0 < adjM[w][v] < INF:
                D[v] = min(D[v], D[w]+adjM[w][v])


INF = 10000
V, E = map(int, input().split())
adjM = [[INF]*(V+1) for _ in range(V+1)]

for i in range(V+1):
    adjM[i][i] = 0

for _ in range(E):
    u, v, w = map(int, input().split())
    adjM[u][v] = w

D = [0]*(V+1)
dijkstra(0, V)
print(D)
```

구현

우선순위 큐를
이용한 구현

시간 복잡도

O(ElogV)

```python
import heapq  # 우선순위 큐 구현을 위함

def dijkstra(graph, start):
  distances = {node: float('inf') for node in graph}  # start로 부터의 거리 값을 저장하기 위함
  print(distances)
  distances[start] = 0  # 시작 값은 0이어야 함
  queue = []
  heapq.heappush(queue, [distances[start], start])  # 시작 노드부터 탐색 시작 하기 위함.

  while queue:  # queue에 남아 있는 노드가 없으면 끝
    current_distance, current_destination = heapq.heappop(queue)  # 탐색 할 노드, 거리를 가져옴.

    if distances[current_destination] < current_distance:  # 기존에 있는 거리보다 길다면, 볼 필요도 없음
      continue

    for new_destination, new_distance in graph[current_destination].items():
      distance = current_distance + new_distance  # 해당 노드를 거쳐 갈 때 거리
      if distance < distances[new_destination]:  # 알고 있는 거리 보다 작으면 갱신
        distances[new_destination] = distance
        heapq.heappush(queue, [distance, new_destination])  # 다음 인접 거리를 계산 하기 위해 큐에 삽입

  return distances


graph = {
    'A': {'B': 8, 'C': 1, 'D': 2},
    'B': {},
    'C': {'B': 5, 'D': 2},
    'D': {'E': 3, 'F': 5},
    'E': {'F': 1},
    'F': {'A': 5}
}

print(dijkstra(graph, 'A'))
```

https://justkode.kr/algorithm/python-dijkstra

우선순위 큐

들어간 순서에 따라 처리되는 First-in First-out 구조를 가진 일반적인 큐와 달리

각 원소가 우선순위를 가지고 있고, **높은 우선순위를 가진 원소를 가진 원소가 먼저 처리되는 큐**

(우선순위가 같을 경우 큐에 위치한 순서대로 처리)

우선순위 큐의 구현

일반적으로 **힙**(Heap)이라는 자료구조를 통해 구현
(가장 효율적이기 때문)

일반 배열 or 연결리스트: 삭제 O(1) 삽입 O(n)
힙: 삭제 O(logN) 삽입 O(logN)

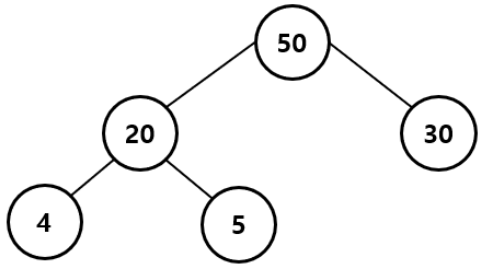
*** 완전 이진트리(Complete binary tree)**

마지막 레벨을 제외하고 모든 레벨이 완전히 채워져 있으며,
마지막 레벨의 모든 노드는 가능한 한 가장 왼쪽에 있는 트리

힙(Heap)

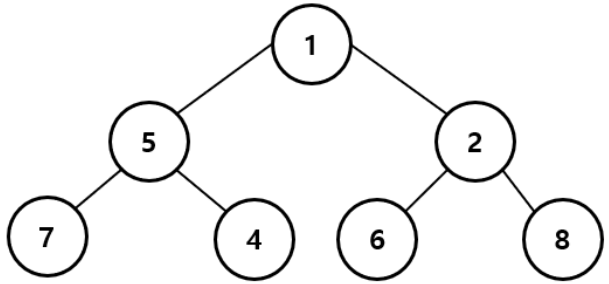
최댓값 및 최솟값을 찾아내는 연산을 빠르게 하기 위해 고안된
완전 이진트리(complete binary tree)를 기본으로 한 자료구조

- 느슨한 정렬 상태
- 중복 값 허용



최대 힙

루트 노드로 올라갈 수록 값이 커짐



최소 힙

루트 노드로 올라갈 수록 값이 작아짐

heapq

최소 힙 자료구조를 통해 원소들을 정렬하는 우선순위 큐를 제공하는 파이썬 모듈

```
Import heapq
```

heapq.**heappush(*heap*, item)**

힙 불변성을 유지하면서, item 값을 heap으로 push

heapq.**heappop(*heap*)**

힙 불변성을 유지하면서, heap에서 가장 작은 항목을 pop하고 반환

* 힙큐가 별개의 자료구조를 제공하는 것이 아니라 일반적인 리스트를 최소 힙처럼 다룰 수 있도록 도와주는 것이기 때문에 첫번째 인자로 정렬할 리스트를 넘겨주어야 함

https://docs.python.org/ko/3/library/heapq.html

구현

우선순위 큐를
이용한 구현

출력

{'A': inf, 'B': inf, 'C': inf, 'D': inf, 'E': inf, 'F': inf}

{'A': 0, 'B': 6, 'C': 1, 'D': 2, 'E': 5, 'F': 6}

```python
import heapq  # 우선순위 큐 구현을 위함

def dijkstra(graph, start):
  distances = {node: float('inf') for node in graph}  # start로 부터의 거리 값을 저장하기 위함
  print(distances)
  distances[start] = 0  # 시작 값은 0이어야 함
  queue = []
  heapq.heappush(queue, [distances[start], start])  # 시작 노드부터 탐색 시작 하기 위함.

  while queue:  # queue에 남아 있는 노드가 없으면 끝
    current_distance, current_destination = heapq.heappop(queue)  # 탐색 할 노드, 거리를 가져옴.

    if distances[current_destination] < current_distance:  # 기존에 있는 거리보다 길다면, 볼 필요도 없음
      continue

    for new_destination, new_distance in graph[current_destination].items():
      distance = current_distance + new_distance  # 해당 노드를 거쳐 갈 때 거리
      if distance < distances[new_destination]:  # 알고 있는 거리 보다 작으면 갱신
        distances[new_destination] = distance
        heapq.heappush(queue, [distance, new_destination])  # 다음 인접 거리를 계산 하기 위해 큐에 삽입

  return distances


graph = {
    'A': {'B': 8, 'C': 1, 'D': 2},
    'B': {},
    'C': {'B': 5, 'D': 2},
    'D': {'E': 3, 'F': 5},
    'E': {'F': 1},
    'F': {'A': 5}
}

print(dijkstra(graph, 'A'))
```

https://justkode.kr/algorithm/python-dijkstra

구현

우선순위 큐를
이용한 구현2

입력

```
5 11
0 1 3
0 2 5
1 2 2
1 3 6
2 1 1
2 3 4
2 4 6
3 4 2
3 5 3
4 0 3
4 5 6
```

출력

[0, 3, 5, 9, 11, 12]

```python
import heapq

def dijkstra(s, V):
    D[s] = 0
    queue = []
    heapq.heappush(queue, [D[s], s])

    while queue:
        print(queue)
        cur_distance, i = heapq.heappop(queue)

        if D[i] < cur_distance:
            continue

        for v in range(V+1):
            if 0 < adjM[i][v] < INF:
                d = cur_distance + adjM[i][v]
                if d < D[v]:
                    D[v] = d
                    heapq.heappush(queue, [D[v], v])

INF = 10000
V, E = map(int, input().split())
adjM = [[INF]*(V+1) for _ in range(V+1)]
for i in range(V+1):
    adjM[i][i] = 0
for _ in range(E):
    u, v, w = map(int, input().split())
    adjM[u][v] = w

D = [INF]*(V+1)
dijkstra(0, V)
print(D)
```

관련 문제

최소비용 구하기

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
0.5 초	128 MB	43013	12515	7925	31.867%

문제

N개의 도시가 있다. 그리고 한 도시에서 출발하여 다른 도시에 도착하는 M개의 버스가 있다. 우리는 A번째 도시에서 B번째 도시까지 가는데 드는 버스 비용을 최소화 시키려고 한다. A번째 도시에서 B번째 도시까지 가는데 드는 최소비용을 출력하여라. 도시의 번호는 1부터 N까지이다.

입력

첫째 줄에 도시의 개수 N(1 ≤ N ≤ 1,000)이 주어지고 둘째 줄에는 버스의 개수 M(1 ≤ M ≤ 100,000)이 주어진다. 그리고 셋째 줄부터 M+2줄까지 다음과 같은 버스의 정보가 주어진다. 먼저 처음에는 그 버스의 출발 도시의 번호가 주어진다. 그리고 그 다음에는 도착지의 도시 번호가 주어지고 또 그 버스 비용이 주어진다. 버스 비용은 0보다 크거나 같고, 100,000보다 작은 정수이다.

그리고 M+3째 줄에는 우리가 구하고자 하는 구간 출발점의 도시번호와 도착점의 도시번호가 주어진다. 출발점에서 도착점을 갈 수 있는 경우만 입력으로 주어진다.

출력

첫째 줄에 출발 도시에서 도착 도시까지 가는데 드는 최소 비용을 출력한다.

https://www.acmicpc.net/problem/1916

관련 문제

최단경로

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
1 초	256 MB	107844	29869	14536	23.767%

문제

방향그래프가 주어지면 주어진 시작점에서 다른 모든 정점으로의 최단 경로를 구하는 프로그램을 작성하시오. 단, 모든 간선의 가중치는 10 이하의 자연수이다.

입력

첫째 줄에 정점의 개수 V와 간선의 개수 E가 주어진다. ($1 \le V \le 20,000$, $1 \le E \le 300,000$) 모든 정점에는 1부터 V까지 번호가 매겨져 있다고 가정한다. 둘째 줄에는 시작 정점의 번호 K($1 \le K \le V$)가 주어진다. 셋째 줄부터 E개의 줄에 걸쳐 각 간선을 나타내는 세 개의 정수 (u, v, w)가 순서대로 주어진다. 이는 u에서 v로 가는 가중치 w인 간선이 존재한다는 뜻이다. u와 v는 서로 다르며 w는 10 이하의 자연수이다. 서로 다른 두 정점 사이에 여러 개의 간선이 존재할 수도 있음에 유의한다.

출력

첫째 줄부터 V개의 줄에 걸쳐, i번째 줄에 i번 정점으로의 최단 경로의 경로값을 출력한다. 시작점 자신은 0으로 출력하고, 경로가 존재하지 않는 경우에는 INF를 출력하면 된다.

https://www.acmicpc.net/problem/1753

관련 문제

녹색 옷 입은 애가 젤다지? 다국어

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
1 초	256 MB	11775	6182	4305	50.989%

문제

젤다의 전설 게임에서 화폐의 단위는 루피(rupee)다. 그런데 간혹 '도둑루피'라 불리는 검정색 루피도 존재하는데, 이걸 획득하면 오히려 소지한 루피가 감소하게 된다!

젤다의 전설 시리즈의 주인공, 링크는 지금 도둑루피만 가득한 N x N 크기의 동굴의 제일 왼쪽 위에 있다. [0][0]번 칸이기도 하다. 왜 이런 곳에 들어왔냐고 묻는다면 밖에서 사람들이 자꾸 "젤다의 전설에 나오는 녹색 애가 젤다지?"라고 물어봤기 때문이다. 링크가 녹색 옷을 입은 주인공이고 젤다는 그냥 잡혀있는 공주인데, 게임 타이틀에 젤다가 나와 있다고 자꾸 사람들이 이렇게 착각하니까 정신병에 걸릴 위기에 놓인 것이다.

하여튼 젤다...아니 링크는 이 동굴의 반대편 출구, 제일 오른쪽 아래 칸인 [N-1][N-1]까지 이동해야 한다. 동굴의 각 칸마다 도둑루피가 있는데, 이 칸을 지나면 해당 도둑루피의 크기만큼 소지금을 잃게 된다. 링크는 잃는 금액을 최소로 하여 동굴 건너편까지 이동해야 하며, 한 번에 상하좌우 인접한 곳으로 1칸씩 이동할 수 있다.

링크가 잃을 수밖에 없는 최소 금액은 얼마일까?

입력

입력은 여러 개의 테스트 케이스로 이루어져 있다.

각 테스트 케이스의 첫째 줄에는 동굴의 크기를 나타내는 정수 N이 주어진다. ($2 \le N \le 125$) N = 0인 입력이 주어지면 전체 입력이 종료된다.

이어서 N개의 줄에 걸쳐 동굴의 각 칸에 있는 도둑루피의 크기가 공백으로 구분되어 차례대로 주어진다. 도둑루피의 크기가 k면 이 칸을 지나면 k루피를 잃는다는 뜻이다. 여기서 주어지는 모든 정수는 0 이상 9 이하인 한 자리 수다.

https://www.acmicpc.net/problem/4485

관련 문제

녹색 옷 입은 애가 젤다지? 다국어

예제 입력 1 복사

```
3
5 5 4
3 9 1
3 2 7
5
3 7 2 0 1
2 8 0 9 1
1 2 1 8 1
9 8 9 2 0
3 6 5 1 5
7
9 0 5 1 1 5 3
4 1 2 1 6 5 3
0 7 6 1 6 8 5
1 1 7 8 3 2 3
9 4 0 7 6 4 1
5 8 3 2 4 8 3
7 4 8 4 8 3 4
0
```

예제 출력 1 복사

```
Problem 1: 20
Problem 2: 19
Problem 3: 36
```

https://www.acmicpc.net/problem/4485

관련 문제

SWEA 1795. 인수의 생일파티

```python
def dijkstra(adj, D, X, N):
    U = [0] * (N + 1)
    U[X] = 1
    for i in range(1, N + 1):
        D[i] = adj[X][i]
    for _ in range(N - 1):
        w = 0
        minV = INF
        for i in range(1, N + 1):
            if not U[i] and minV > D[i]:
                minV = D[i]
                w = i
        U[w] = 1
        for v in range(1, N + 1):
            if 0 < adj[w][v] < INF:
                D[v] = min(D[v], D[w] + adj[w][v])


INF = 987654321
for tc in range(1, int(input()) + 1):
    N, M, X = map(int, input().split())
    adjM = [[INF] * (N + 1) for _ in range(N + 1)]
    adjMR = [[INF] * (N + 1) for _ in range(N + 1)]
    for i in range(1, N + 1):
        adjM[i][i] = 0
        adjMR[i][i] = 0
    for _ in range(M):
        x, y, c = map(int, input().split())
        adjM[x][y] = c
        adjMR[y][x] = c

    Dout = [0] * (N + 1)
    Din = [0] * (N + 1)
    dijkstra(adjM, Dout, X, N)
    dijkstra(adjMR, Din, X, N)
    maxV = 0
    for i in range(1, N + 1):
        if maxV < Din[i] + Dout[i]:
            maxV = Din[i] + Dout[i]
    print(f'#{tc} {maxV}')
```

https://swexpertacademy.com/main/code/problem/problemDetail.do?contestProbId=AV4xuqCqBeUDFAUx&categoryId=AV4xuqCqBeUDFAUx&categoryType=CODE&problemTitle=%EC%9D%B8%EC%88%98&orderBy=FIRST_REG_DATETIME&selectCodeLang=ALL&select-1=&pageSize=10&pageIndex=1

관련 알고리즘 비교

가중치 유향 그래프의 최단 경로 탐색 알고리즘

다익스트라

비교적 빠름
가중치가 음수인 경우 처리할 수 x

벨만-포드

다익스트라에 비해 느림: O(VE)
가중치가 음수인 경우 처리할 수 o
음수 사이클의 존재 여부를 알 수 있음

(음수 사이클이 존재하는 경우 해당 사이클을 계속 돌수록 더 작은 값으로 갱신되기 때문에 최단 거리를 구하지 못하고 무한루프에 빠짐)

https://velog.io/@younge/Python-%EC%B5%9C%EB%8B%A8-%EA%B2%BD%EB%A1%9C-%EB%B2%A8%EB%A7%8C-%ED%8F%AC%EB%93%9CBellman-Ford-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98

참고

https://swexpertacademy.com/main/talk/solvingClub/boardCommuView.do?solveclubId=AXsHTyBaqJgDFARX&commuId=AXx9myJad9QDFARs

https://swexpertacademy.com/main/learn/course/lectureVideoPlayer.do

https://justkode.kr/algorithm/python-dijkstra

https://ko.wikipedia.org/wiki/%EB%8D%B0%EC%9D%B4%ED%81%AC%EC%8A%A4%ED%8A%B8%EB%9D%BC_%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98

https://doocong.com/algorithm/dijkstra/

https://chanhuiseok.github.io/posts/ds-4/

https://ko.wikipedia.org/wiki/%EC%9A%B0%EC%84%A0%EC%88%9C%EC%9C%84_%ED%81%90

https://www.daleseo.com/python-heapq/