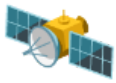


8주차 : 다익스트라

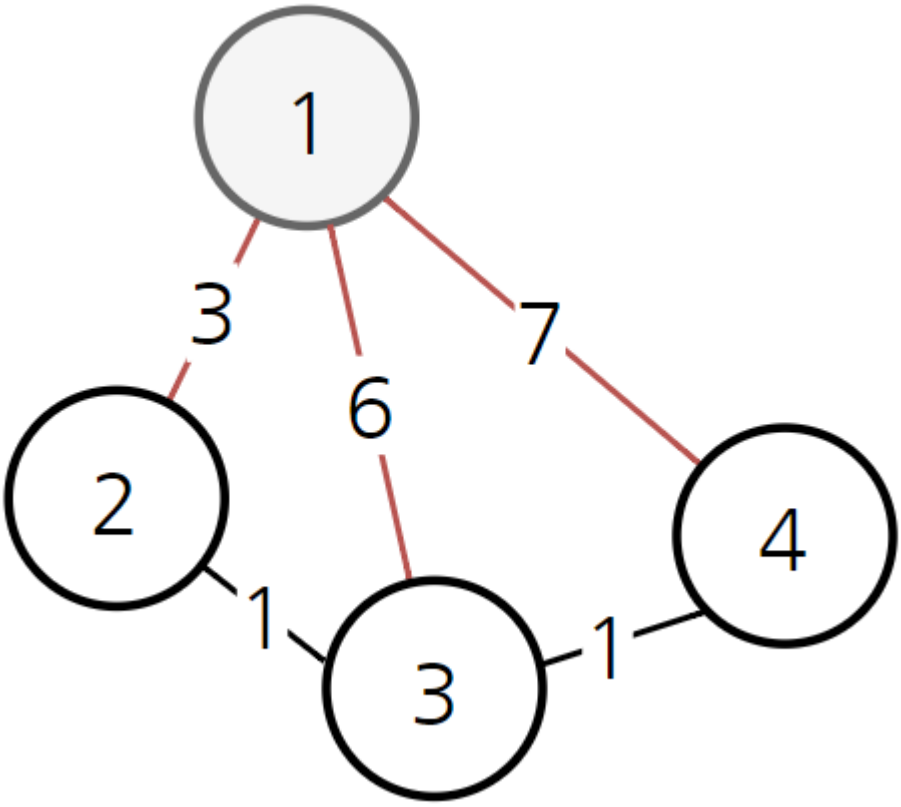
다익스트라(Dijkstra) 알고리즘은 다이나믹 프로그래밍(DP)를 활용한 대표적인 최단경로 탐색 알고리즘이다.

다익스트라 알고리즘은 특정한 **하나의 정점**에서 다른 **모든 정점**으로 가는 최단 경로를 알려준다.

- 음의 가중치를 갖는 간선은 제외! (현실에 사용하기 적합)

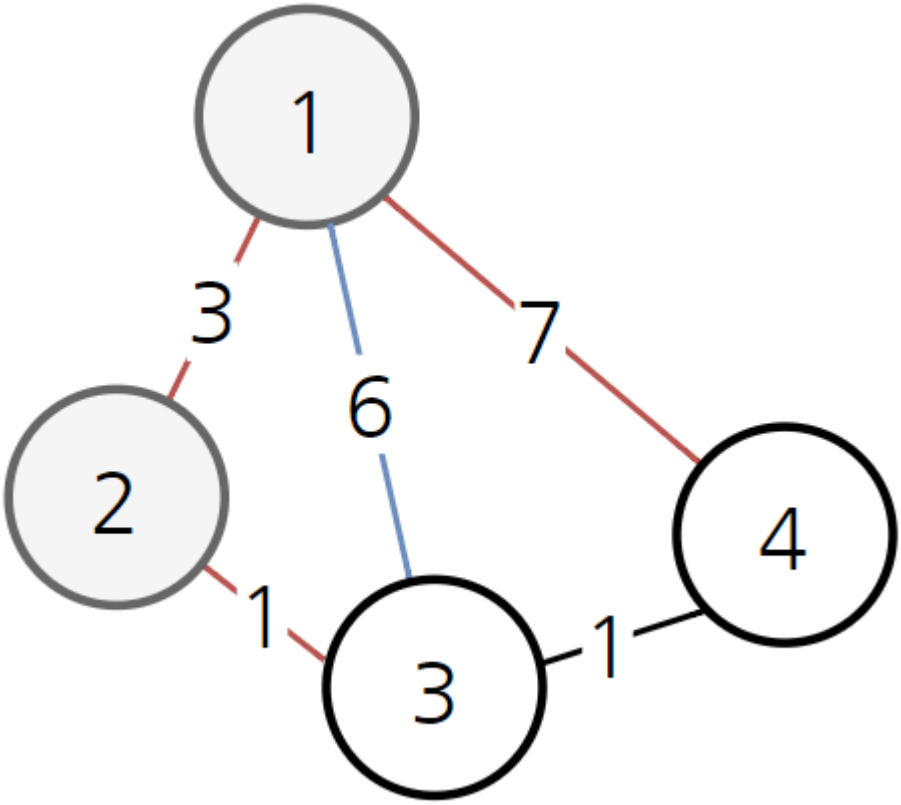
다이나믹 프로그래밍 문제인 이유는 **최단 거리는 여러 개의 최단 거리로 이루어져있기 때문**이다.

하나의 최단 거리를 구할 때 그 전까지 구했던 최단 거리 정보를 그대로 사용한다



시작 정점이 1번일때 갈 수 있는 정점에 대해 최단거리 계산[

node	**1**	2	3	4
value	0	**3**	**6**	**7**



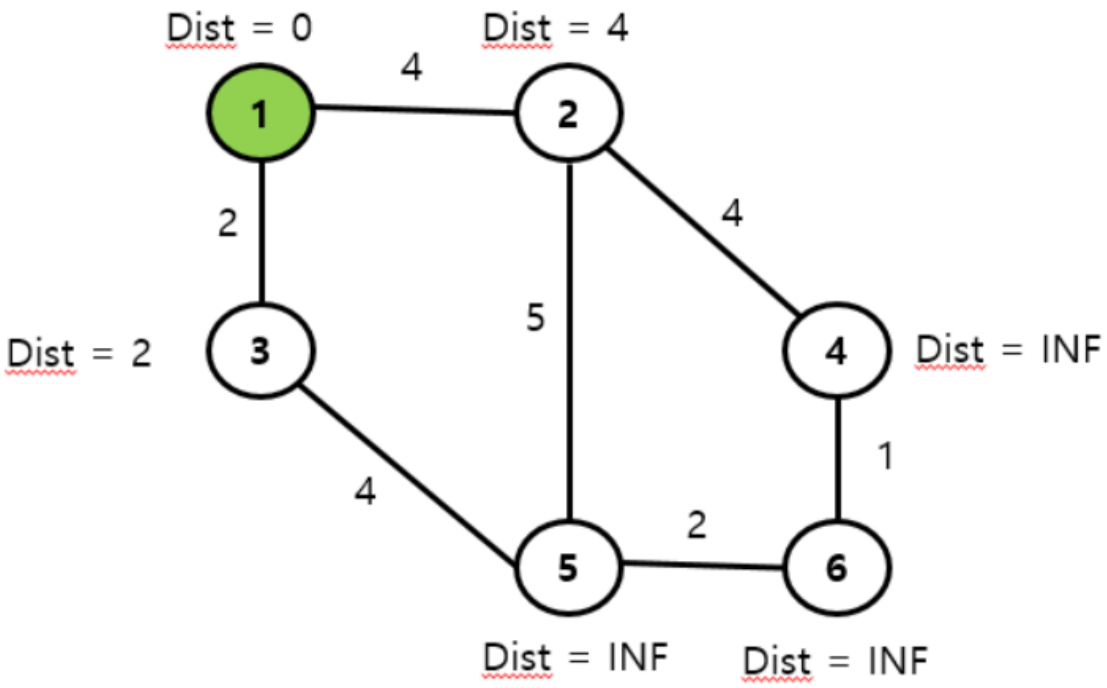
2번 정점 처리시, 갈 수 있는 정점(3)에 대해 최단거리 갱신

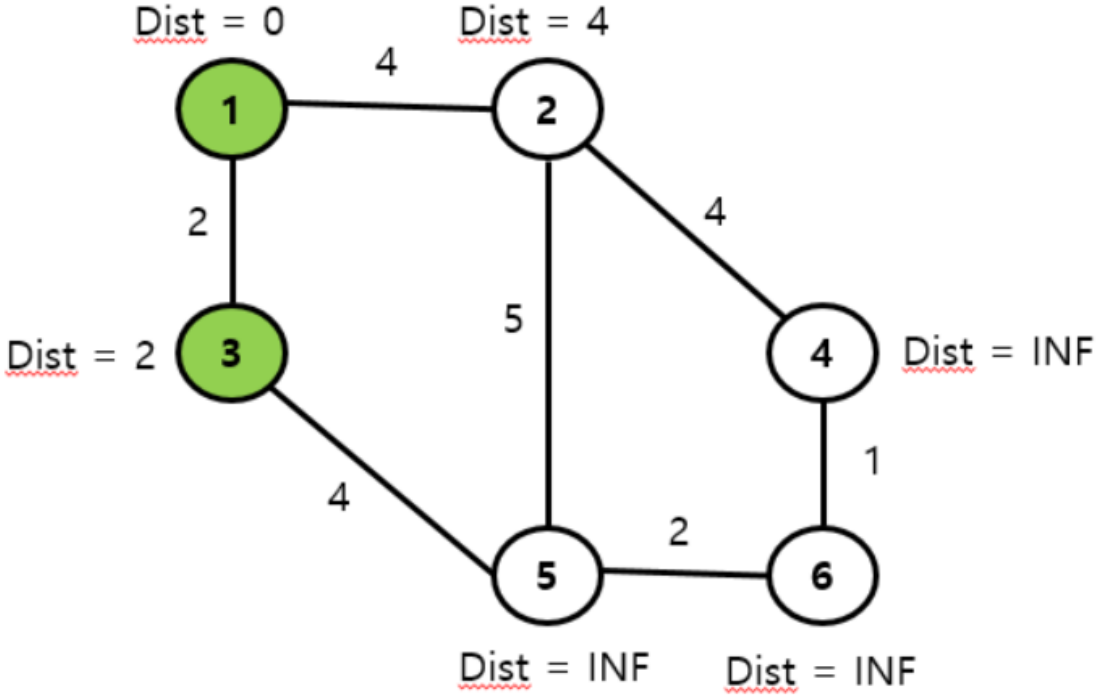
node	1	**2**	3	4
value	0	3	**6→(3+1)4**	7

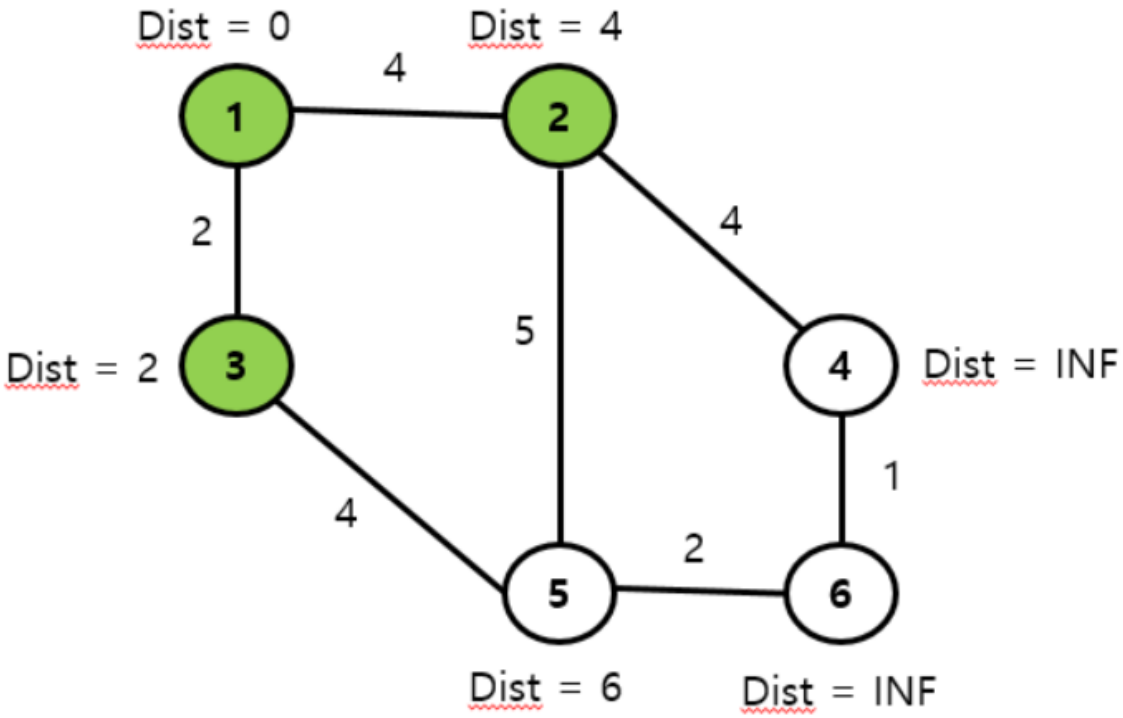
다익스트라의 과정

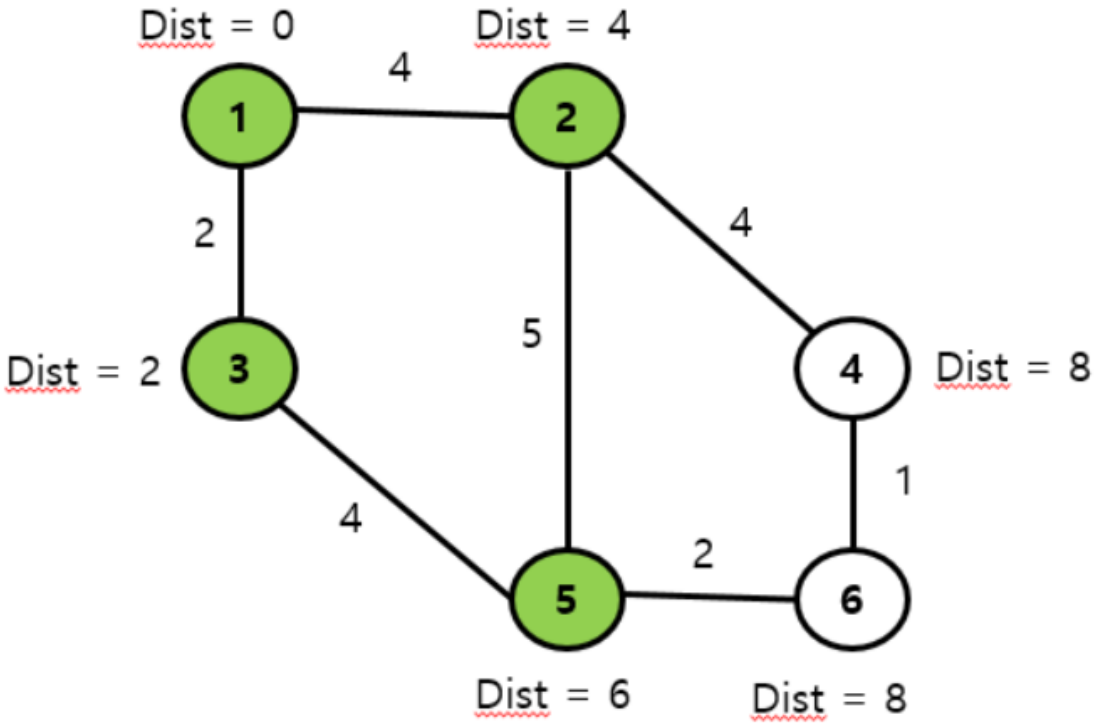
1. 시작 노드 설정 후 방문 노드로 체크.
2. 시작 노드의 인접 노드들에 대해 최단거리 계산
3. 인접 노드 & 방문X 노드 중에서 가장 비용이 적게 드는 노드를 선택. 방문체크
4. 해당 노드의 인접 노드들 간의 거리 계산 및 최소비용 갱신
5. 3번 ~ 4번 과정을 반복

▼ 예시

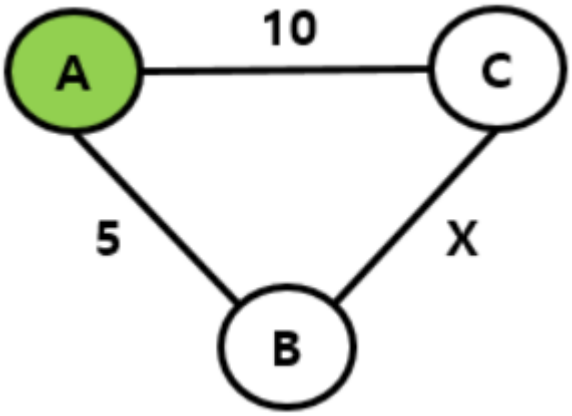

Dist = 0
Dist = 4
4
1
2
2
4
5
Dist = 2
3
4
Dist = INF
1
4
2
5
6
Dist = INF
Dist = INF


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





▼ 방문노드를 재갱신하지 않아도 되는 이유



지금까지 과정에 따르면,

- 가장 비용이 적게드는 A→B 선택
- 이후 방문한 B노드는 갱신되는 일이없음

왜 이럴까?

- 만약, 다른 노드를 거쳐서 B로 가는 경우가 더 작게 되려면
- A→B > A→C→B 어야함.
    - 5 > 10 + X ⇒ **X < -5** ??? (음의 가중치 간선을 제외하는 이유)

다익스트라 코드

```
import sys
input = sys.stdin.readline
INF = int(1e9)

n, m = map(int, input().split())
start = int(input())
# 주어지는 그래프 정보 담는 N개 길이의 리스트
graph = [[] for _ in range(n+1)]
visited = [False] * (n+1)  # 방문처리 기록용
distance = [INF] * (n+1)   # 거리 테이블용

for _ in range(m):
    a, b, c = map(int, input().split())
    graph[a].append((b, c))

# 방문X, 최단거리인 노드 반환
def get_smallest_node():
```

```
    min_value = INF
    index = 0
    for i in range(1, n+1):
        if not visited[i] and distance[i] < min_value:
            min_value = distance[i]
            index = i
    return index

# 다익스트라 알고리즘
def dijkstra(start):
    # 1. 시작 노드를 설정 후 방문 노드로 체크.
    distance[start] = 0
    visited[start] = True
    # 2. 시작노드의 인접한 노드들에 대해 최단거리 계산
    for i in graph[start]:
        distance[i[0]] = i[1]

    # 시작노드 제외한 n-1개의 다른 노드들 처리 5. 3~4번 반복
    for _ in range(n-1):
        # 3. 인접 노드 & 방문X 노드 중에서 가장 비용이 적게 드는 노드를 선택. 방문체크
        now = get_smallest_node()  # 방문X, 최단거리인 노드 반환
        visited[now] = True        # 해당 노드 방문처리
        # 4. 해당 노드의 인접 노드들 간의 거리 계산
        for next in graph[now]:
            cost = distance[now] + next[1]  # 시작->now 거리 + now->now의 인접노드 거리
            if cost < distance[next[0]]:    # cost < 시작->now의 인접노드 다이렉트 거리
                distance[next[0]] = cost


dijkstra(start)

for i in range(1, n+1):
    if distance[i] == INF:
        print('INF')
    else:
        print(distance[i])
```

V가 노드의 개수라고 가정할 때, **시간 복잡도가 O(V^2)**

방문하지 않은 노드 중에서 최단 거리가 가장 짧은 노드를 선택하기 위해 매 단계마다 거리 테이블의 길이(=노드의 개수) 만큼 순차탐색을 수행하기 때문

개선된 다익스트라 (heapq 사용)

```
import sys
import heapq
input = sys.stdin.readline
n, m = map(int, input().split())
start = int(input())
INF = int(1e9)
distance = [INF] * (n+1)
graph = [[] for _ in range(n+1)]

for _ in range(m):
```

```
    a, b, c = map(int, input().split())
    graph[a].append((b, c))


def dijkstra(start):
    q = []
    visited[start] = True
    heapq.heappush(q, (0, start))  # 시작노드 정보 우선순위 큐에 삽입
    distance[start] = 0            # 시작노드->시작노드 거리 기록
    while q:
        dist, node = heapq.heappop(q)
        # 큐에서 뽑아낸 거리가 이미 갱신된 거리보다 클 경우(=방문한 셈) 무시
        if distance[node] < dist:
            continue
        # 큐에서 뽑아낸 노드와 연결된 인접노드들 탐색
        for next in graph[node]:
            cost = distance[node] + next[1]   # 시작->node거리 + node->node의인접노드 거리
            if cost < distance[next[0]]:      # cost < 시작->node의인접노드 거리
                distance[next[0]] = cost
                heapq.heappush(q, (cost, next[0]))


dijkstra(start)

for i in range(1, len(distance)):
    if distance[i] == INF:
        print('INF')
    else:
        print(distance[i])
```

문제

최단경로 (Gold4) : https://www.acmicpc.net/problem/1753

집 구하기 (Gold2) : https://www.acmicpc.net/problem/13911