# DFS와 BFS의 응용 및 다익스트라 (Dijkstra) 알고리즘

## 1. DFS와 BFS의 응용

### 🔎 1.1. DFS와 BFS는 언제 써야 하는가

#### ✏️ 1.1.1. 인접행렬(Adjacency Array)일 때

> 💡 **이동한 거리의 측정**이 필요한 문제: 무조건 **BFS**

- 인접행렬에서 **BFS**에 쌓이는 **큐의 순서**는 반드시 **거리순**이 된다.
- 인접행렬에서 **DFS**는 '현재 보는 칸으로 부터 인접한 칸은 **거리가 1만큼 떨어져 있다**.' 라는 개념을 적용할 수 없다.

> 💡 단순히 탐색하며 **Clustering** 하거나 **Flood Fill**의 문제: BFS와 DFS 모두 가능

### 🔎 1.2. 다차원 배열에서 BFS 주의할 점

- **시작점**에 **방문**했다는 표시를 남겼는지 확인하기
- **pop 할 때 방문했다는 표시**를 남기면 같은 칸이 큐에 여러번 들어가게 되는 꼴이 되므로, 시간/메모리 초과가 발생할 수 있다. (**push 시 방문 표시**를 해야 한다.)
- 인접한 index으로 구한 값이 다차원 배열의 **범위에 포함**되는지 확인하기.

# 2. 다익스트라(Dijkstra) 알고리즘

## 🔎 2.1. 다익스트라 알고리즘의 기본

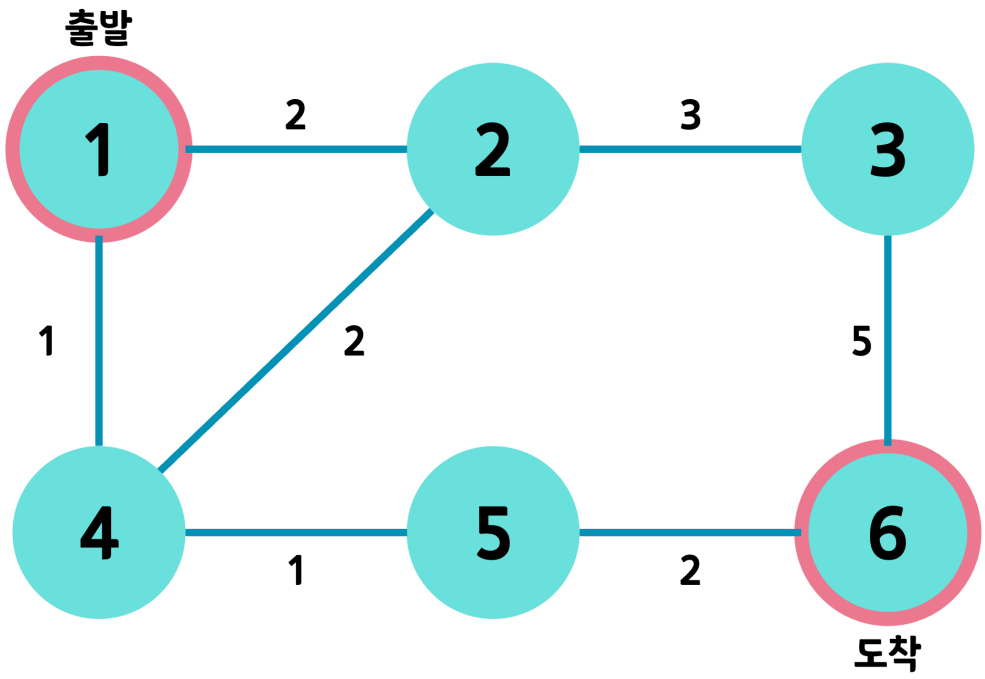
💡 DP을 활용한 탐색 알고리즘으로, **하나의 정점에서 다른 모든 정점으로 가는 최단 경로**를 알려준다.

> DP 문제인 이유는 **최단 거리는 결국, 여러 개의 최단 거리로 이루어져 있기 때문**이다.
>
> 따라서 최단 거리를 구할 때, 직전까지의 최단 거리 정보를 그대로 활용한다.

## 🔎 2.2. 동작 과정

💡 **필요한 요소**
- **최단 거리**를 저장할 테이블
- **노드 방문 여부**를 저장할 배열

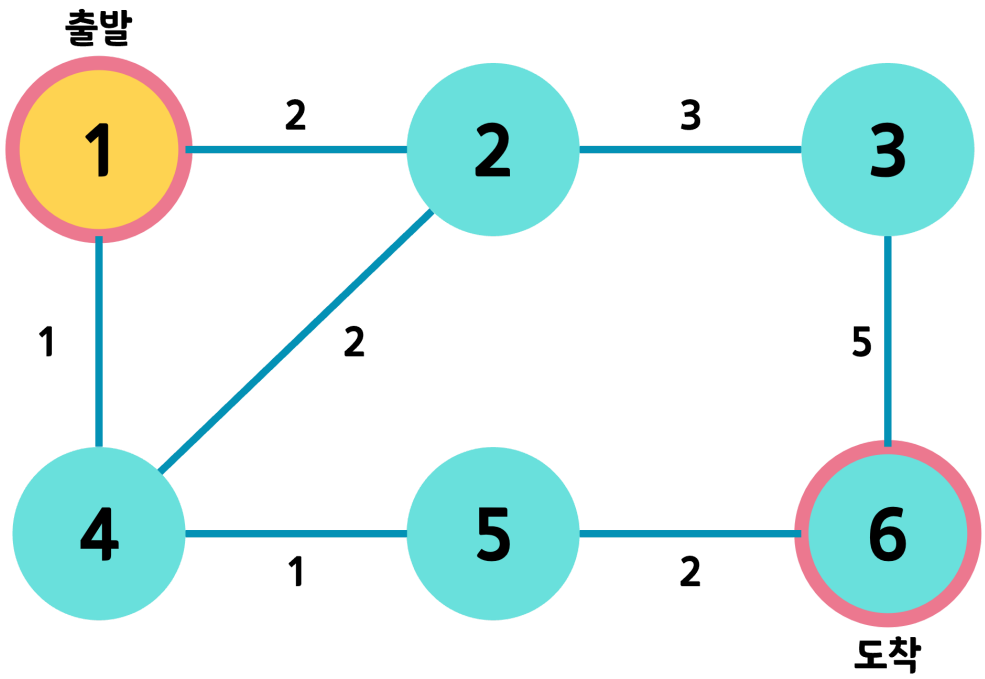
1. **출발 노드**와 **도착 노드**를 설정한다.
2. '**최단 거리 테이블**'을 초기화한다.
3. 현재 위치한 노드의 **인접 노드 중 방문하지 않은 노드**를 구별하고, 방문하지 않은 노드 중 **거리가 가장 짧은 노드**를 선택한다.
   그 노드를 **방문 처리**한다.
4. 해당 노드를 거쳐 **다른 노드로 넘어가는 간선 비용(Weight)을 계산**해 '최단 거리 테이블'을 **업데이트**한다.
5. ③~④의 과정을 **반복**한다.




| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | inf | inf | inf | inf | inf | inf |

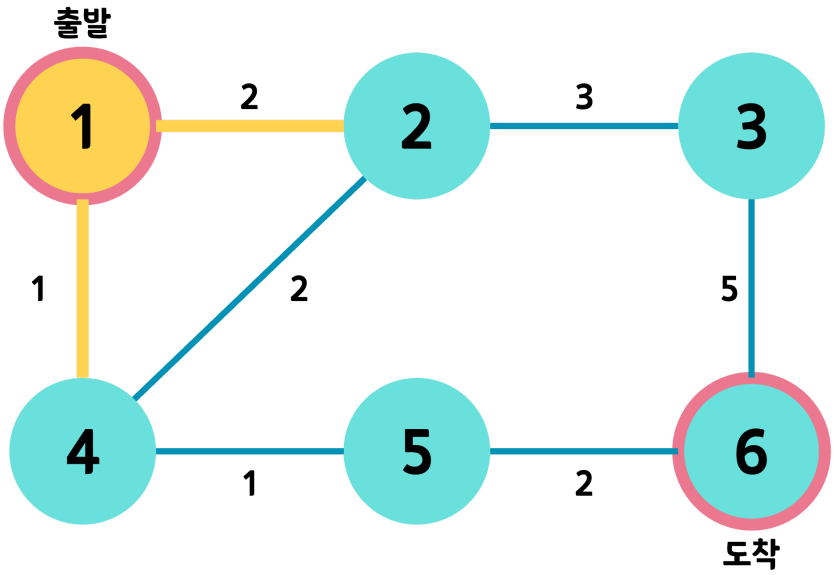
출발 노드를 1번으로, 도착 노드를 6번으로 설정한다.

‘최단 거리 테이블’을 **INF(무한대)**로 초기화 한다.




| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | inf | inf | inf | inf | inf |

출발 노드의 거리를 **0**으로 설정한다.



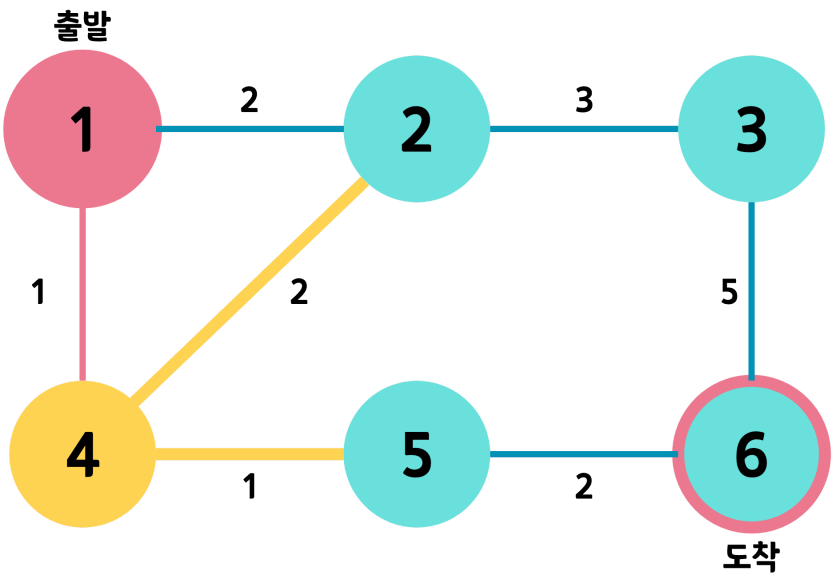

| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | min (inf, 2) | inf | min (inf, 1) | inf | inf |
| | 0 | 2 | inf | 1 | inf | inf |

업데이트

1과 **인접한 노드**는 2, 4이므로 2, 4 노드로 가는 거리 비용을 계산한다.

- **계산한 거리 비용**
  - 노드 1→노드 2의 거리 비용: 2
  - 노드 1→노드 4의 거리 비용: 1

**계산한 거리 비용과 기존의 기존값(INF)을 비교**하여 **최솟값**으로 거리를 업데이트 한다.




| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | min (2,1+2) | inf | 1 | min (inf,1+1) | inf |
| | 0 | 2 | inf | 1 | 2 | inf |

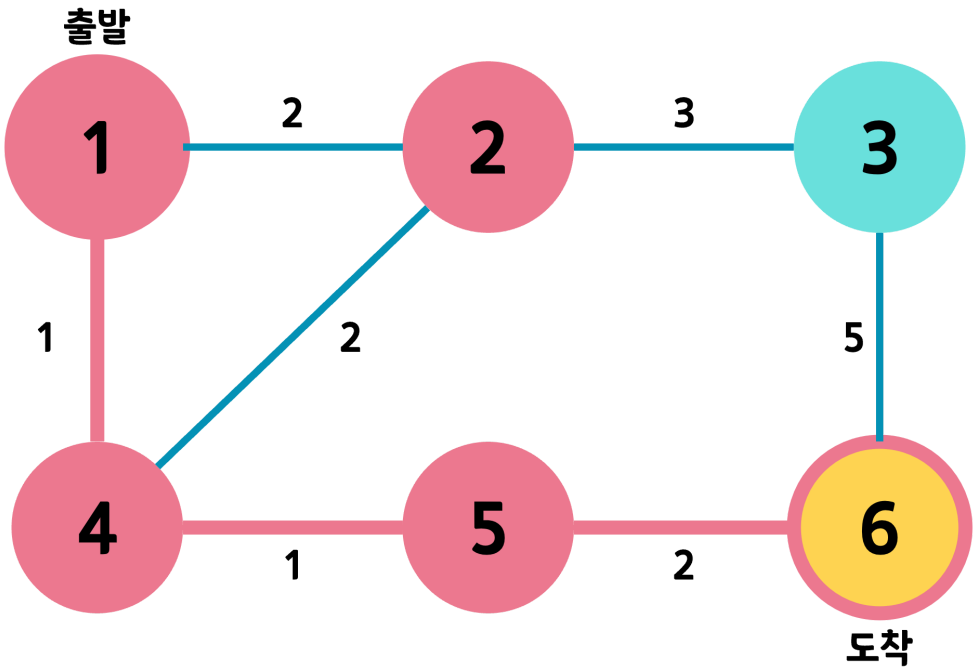
업데이트

위와 반복 과정을 수행한다.

선택된 노드 4에서는 인접한 노드가 2, 5 이므로 **노드 4까지 오는 데에 필요한 거리 비용 + 노드 2, 5로 가는 거리 비용**으로 계산한다.

- **계산한 거리 비용**
  - 노드 4→노드 2의 거리 비용: 1+2
  - 노드 4→노드 5의 거리 비용: 1+1

노드 2의 경우, 1번에서 가는 비용(1)보다 4번을 거쳐 가는 비용(3)이 크므로 업데이트 하지 않는다.




| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | 2 | 5 | 1 | 2 | 4 |

해당 과정을 반복하면, 최종적으로 **1번 노드에서 다른 모든 노드로 가는 거리 비용**을 계산할 수 있게 된다.

## 🔎 2.3. 구현 방법

### ✏️ 2.3.1. 순차 탐색

> 💡 graph을 **순차적으로 탐색**하여 아직 방문하지 않은 노드 중 **가장 거리가 짧은 노드의 인덱스를 반환**하는 방법

노드의 개수가 N일 때, 각 노드마다 최소 거리값을 갖는 노드를 순차 탐색 하므로 **시간 복잡도**는 아래와 같다.

$(N-1) \times N = O(N^2)$

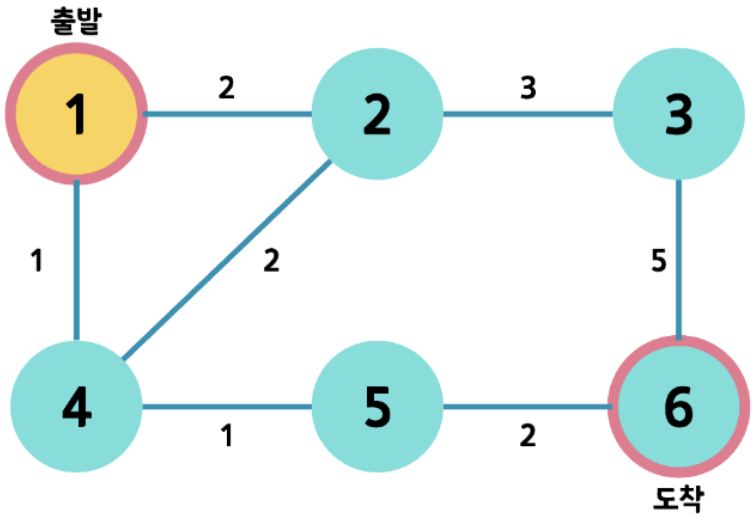
## ✏️ 2.3.2. 우선순위 큐

> 💡 우선순위 큐에서 사용할 '우선순위'의 기준은 '**시작 노드로부터 가장 가까운 노드**'가 된다. 따라서 큐의 정렬은 최단 거리인 노드를 기준으로 **최단 거리를 가지는 노드**를 앞에 배치한다.

우선순위 큐를 사용하면 **방문 여부를 기록할 배열**이 필요 없다.

최소 힙으로 구현된 우선순위 큐가 **최단 거리의 노드를 앞으로 정렬**하기 때문이다.

우선순위 큐를 이용한 **시간 복잡도**는 간선의 수를 E(Edge), 노드의 수를 N(Vertex)라고 했을 때, $O(E\ logN)$가 된다.



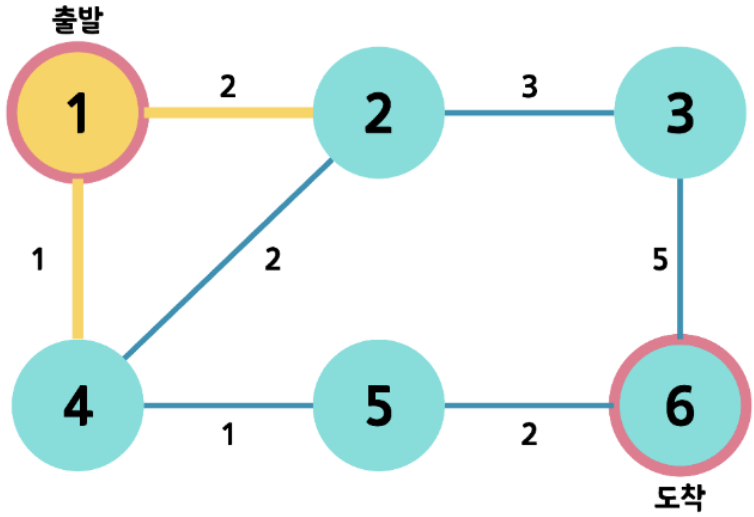

거리 배열

| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | inf | inf | inf | inf | inf |

꺼낸 노드: X

우선순위 큐: < 거리 0, 노드 1 >

출발 노드 1번의 거리를 0으로 업데이트 하고, 우선순위 큐에 넣는다.




**거리 배열**

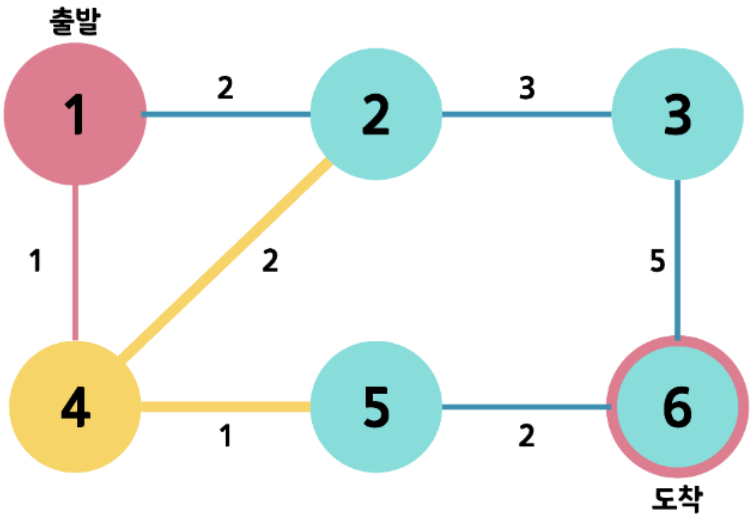
| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | 2 | inf | 1 | inf | inf |

**꺼낸 노드** : < 거리 0, 노드 1 >

**우선순위 큐** : < 거리 1, 노드 4 > , < 거리 2, 노드 2 >

큐에서 가장 앞에 있던 **<거리 0, 노드 1>을 pop**하고, 해당 노드의 **인접 노드**를 조사한다.

인접 노드에 해당하는 2와 4의 **거리 비용을 계산**한 뒤, **최소 거리로 새로 업데이트 된 노드**만 우선순위 큐에 넣는다. (우선순위 큐는 거리값이 작은 순서대로 정렬한다.)




**거리 배열**

| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | 2 | inf | 1 | 2 | inf |

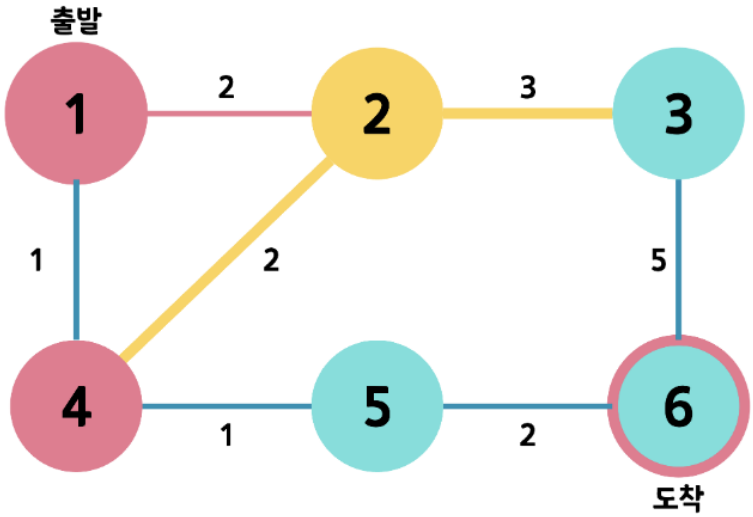
**꺼낸 노드** : < 거리 1, 노드 4 >

**우선순위 큐** : < 거리 2, 노드 2 > , < 거리 2, 노드 5 >

큐에서 가장 앞에 있던 **<거리 1, 노드 4>을 pop**하고, 해당 노드의 **인접 노드**를 조사한다.

인접 노드에 해당하는 2와 5의 **거리 비용을 계산**한다.

이 때, **새로 업데이트 된 노드**는 5번만 해당하므로 이 요소만 우선순위 큐에 push 한다.




거리 배열

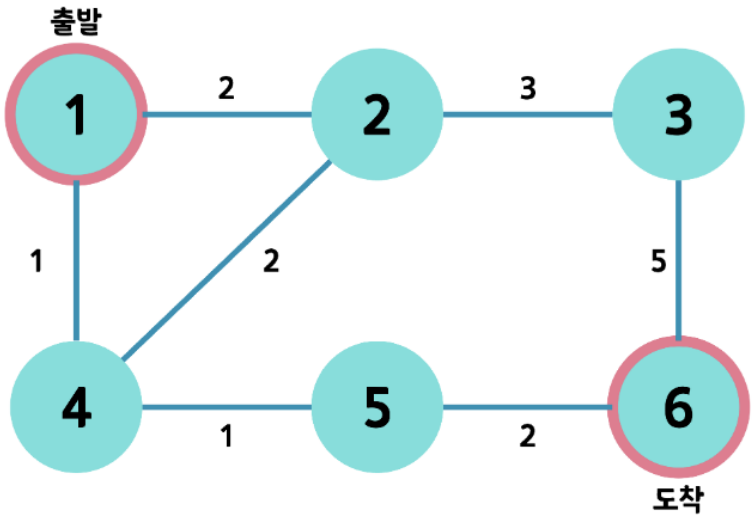
| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | 2 | 5 | 1 | 2 | inf |

꺼낸 노드: < 거리 2, 노드 2 >

우선순위 큐: < 거리 2, 노드 5 > , < 거리 5, 노드 3 >

같은 과정을 반복하여, **<거리 2, 노드 2>을 pop**하고, 해당 노드의 **인접 노드**를 조사한다.

새로 갱신된 노드는 3번이므로 3번에 대한 거리만 우선순위 큐에 넣는다.




거리 배열

| 노드 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 거리 | 0 | 2 | 5 | 1 | 2 | 4 |

위의 과정을 **우선순위 큐가 빌 때까지 반복**하면, 최종적으로 도착 노드의 거리값이 최소 거리로 구해지게 된다.

---

## 📝 참고

- **Flood Fill**에 대한 개념 https://nooblette.tistory.com/264?category=990410
- **다차원 배열**에서의 **DFS**와 **BFS**
  https://velog.io/@dnflrhkddyd/알고리즘-BFS

- **DFS**와 **BFS**의 **차이**

  https://iancoding.tistory.com/329
- **다익스트라 알고리즘** 상세설명

  https://velog.io/@717lumos/알고리즘-다익스트라Dijkstra-알고리즘