soft squared

알고리즘 캠프 3주차

컴공 선배

퀴즈

Quiz

퀴즈 Quiz

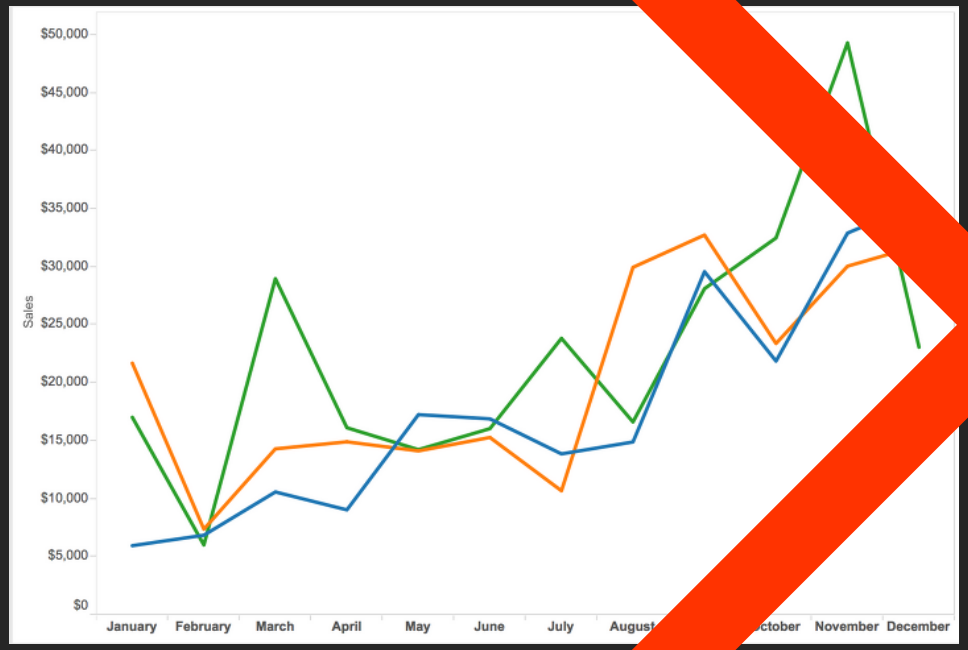
1. Prev Quiz: 23757. 아이들과 선물 상자
   - 문제 풀어서 소스 코드 공개 옵션 '공개'로 백준에 제출하기
2. Signature Quiz: 3986. 좋은 단어
   - 풀이 방법 및 pseudo code를 DM으로 보내주세요

목차

- 그래프
  - 차트와 그래프
  - 정점과 간선
  - 방향 & 순환 & 연결 요소
  - 인접 행렬 & 인접 리스트
- DFS & BFS
- 백트래킹

그래프

Graph


$50,000
$45,000
$40,000
$35,000
$30,000
$25,000
$20,000
$15,000
$10,000
$5,000
$0
Sales
January
February
March
April
May
June
July
August
November
December

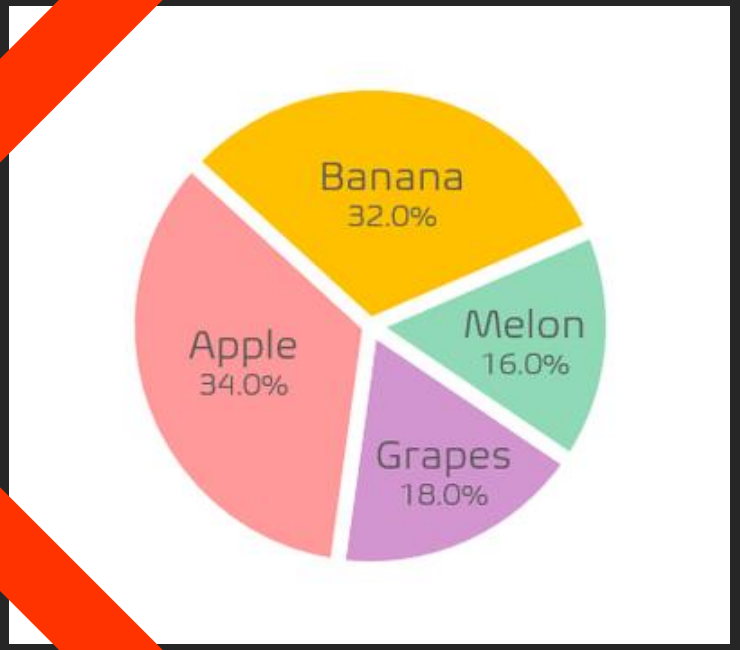
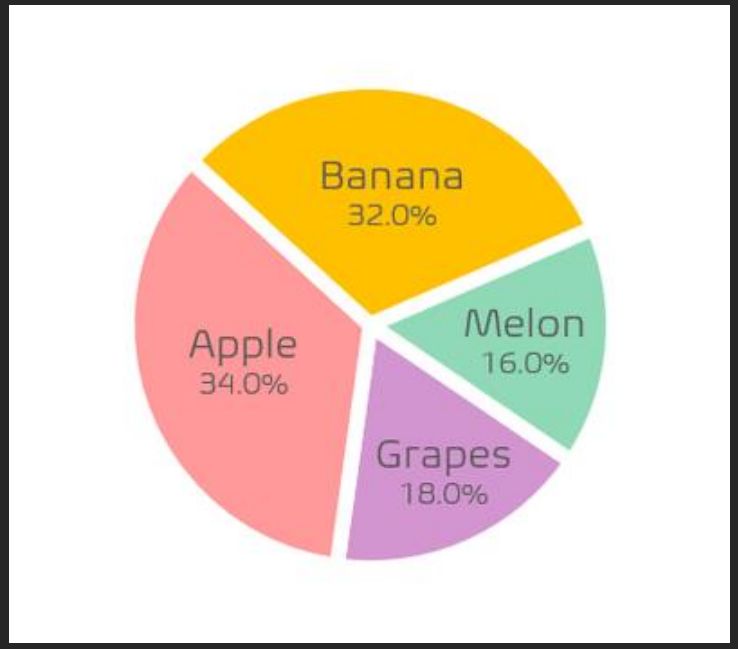

Banana
32.0%
Melon
16.0%
Apple
34.0%
Grapes
18.0%

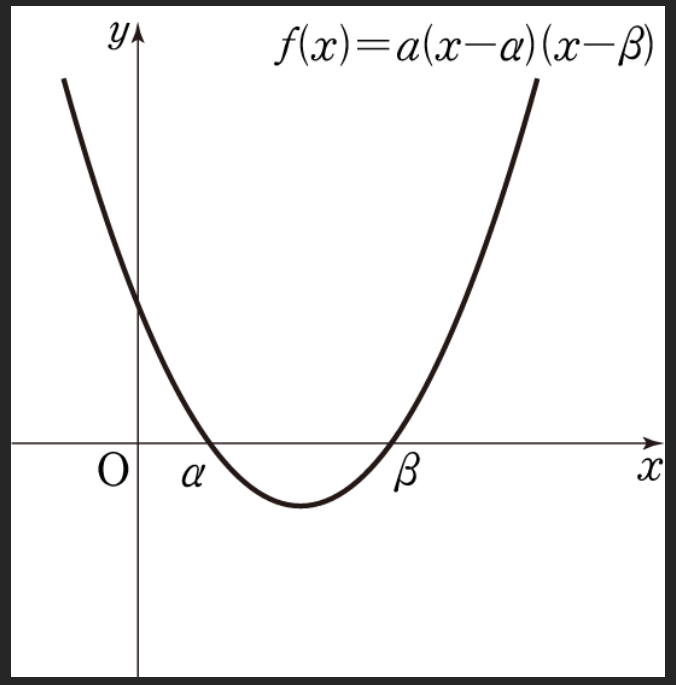
차트 Chart

- 정의
  - 데이터를 시각적으로 분석하기 위해 활용하는 자료



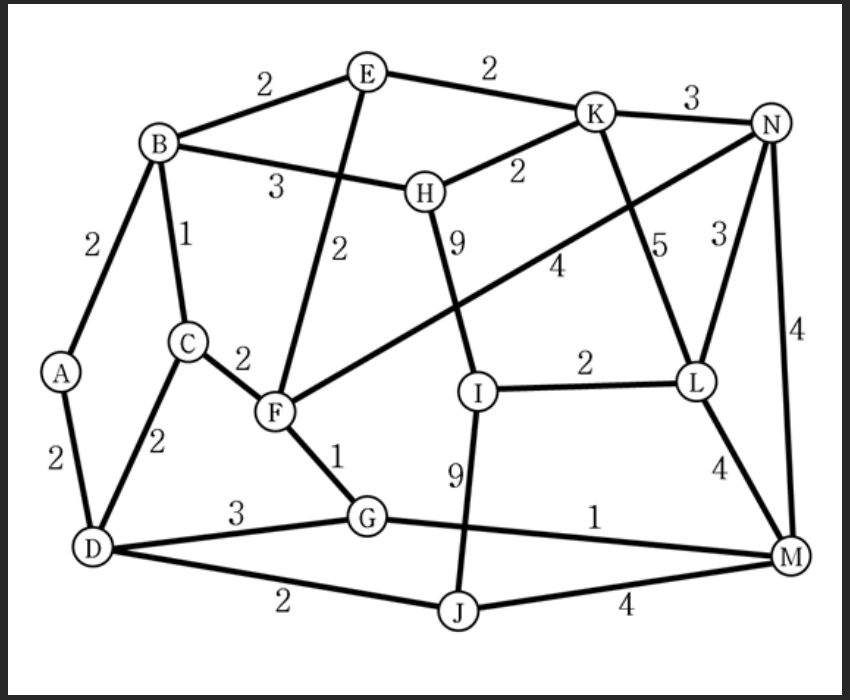
그래프 Graph

- 정의(수학)
  - 데이터의 경향과 변화를 보여주는 것에 초점을 가진 자료



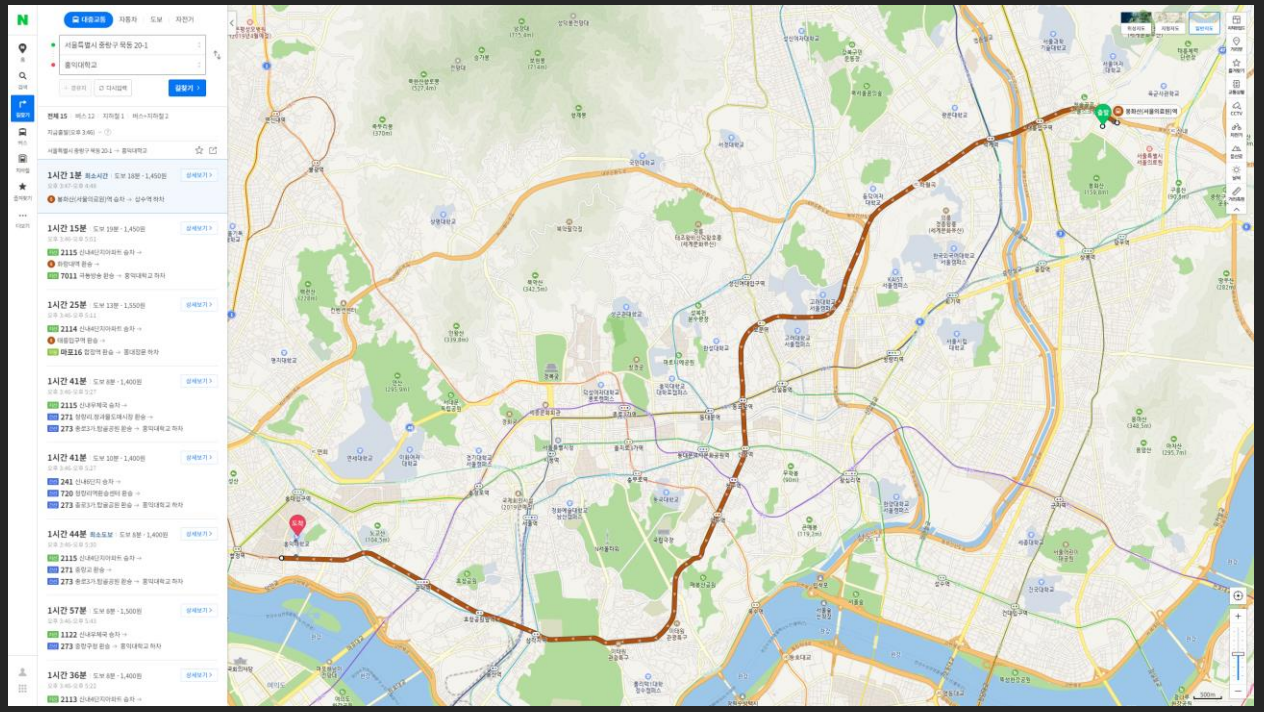
그래프 Graph

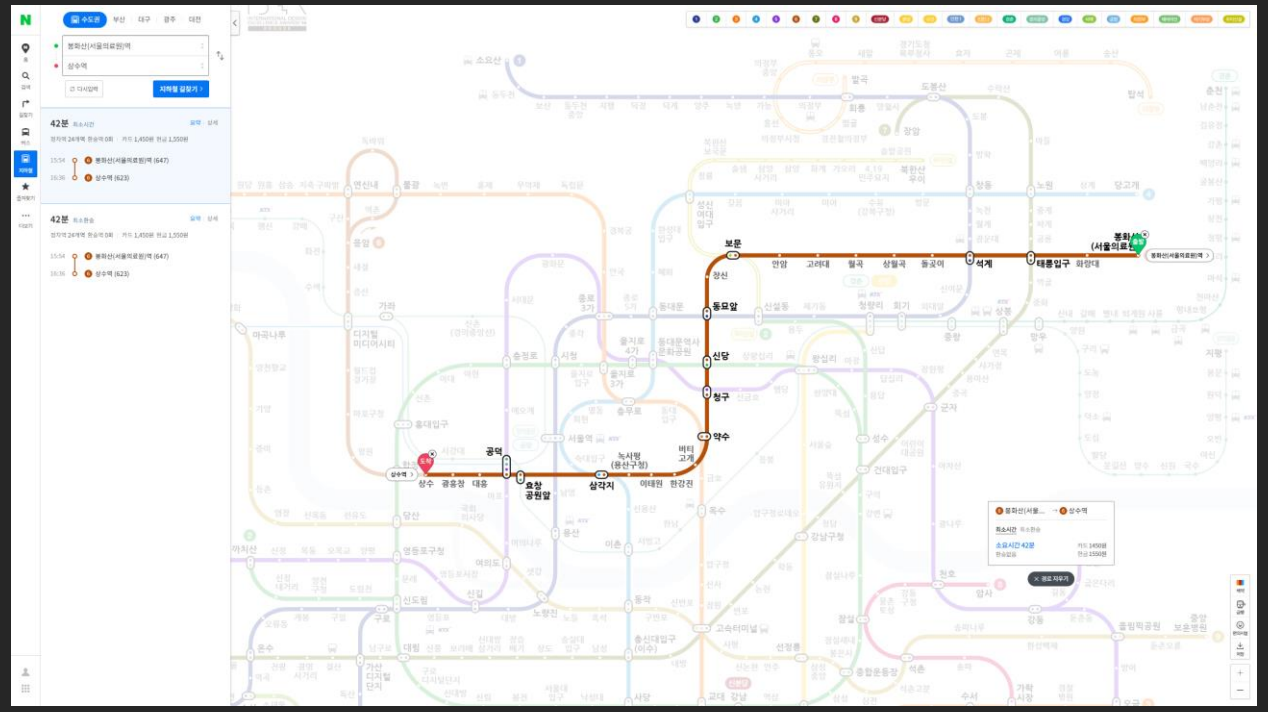
- 정의(IT)
  - 정점과 정점 간의 관계를 표시하는 간선으로 구성된 자료



그래프 실생활 예

- 지도, 내비게이션



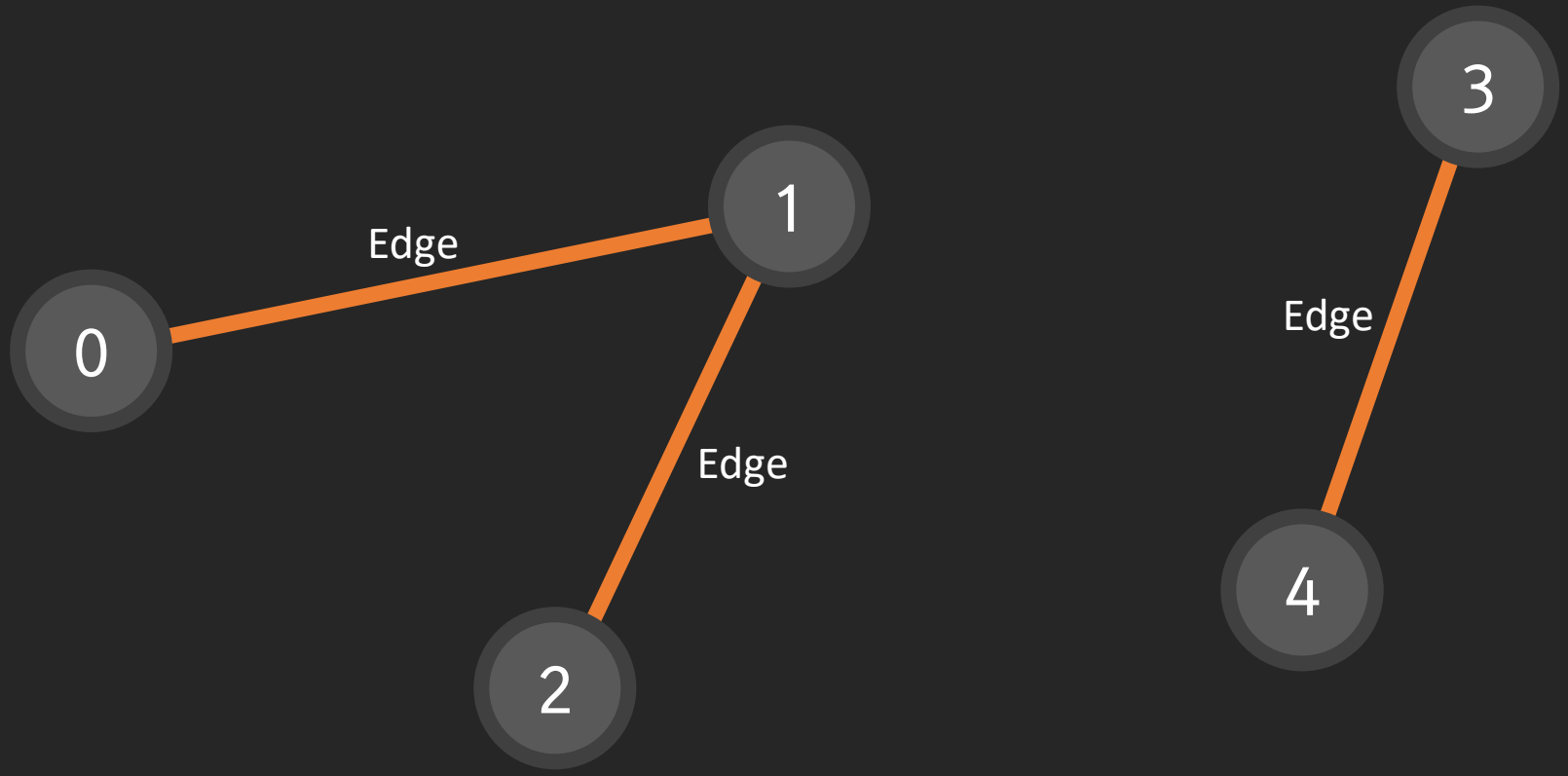


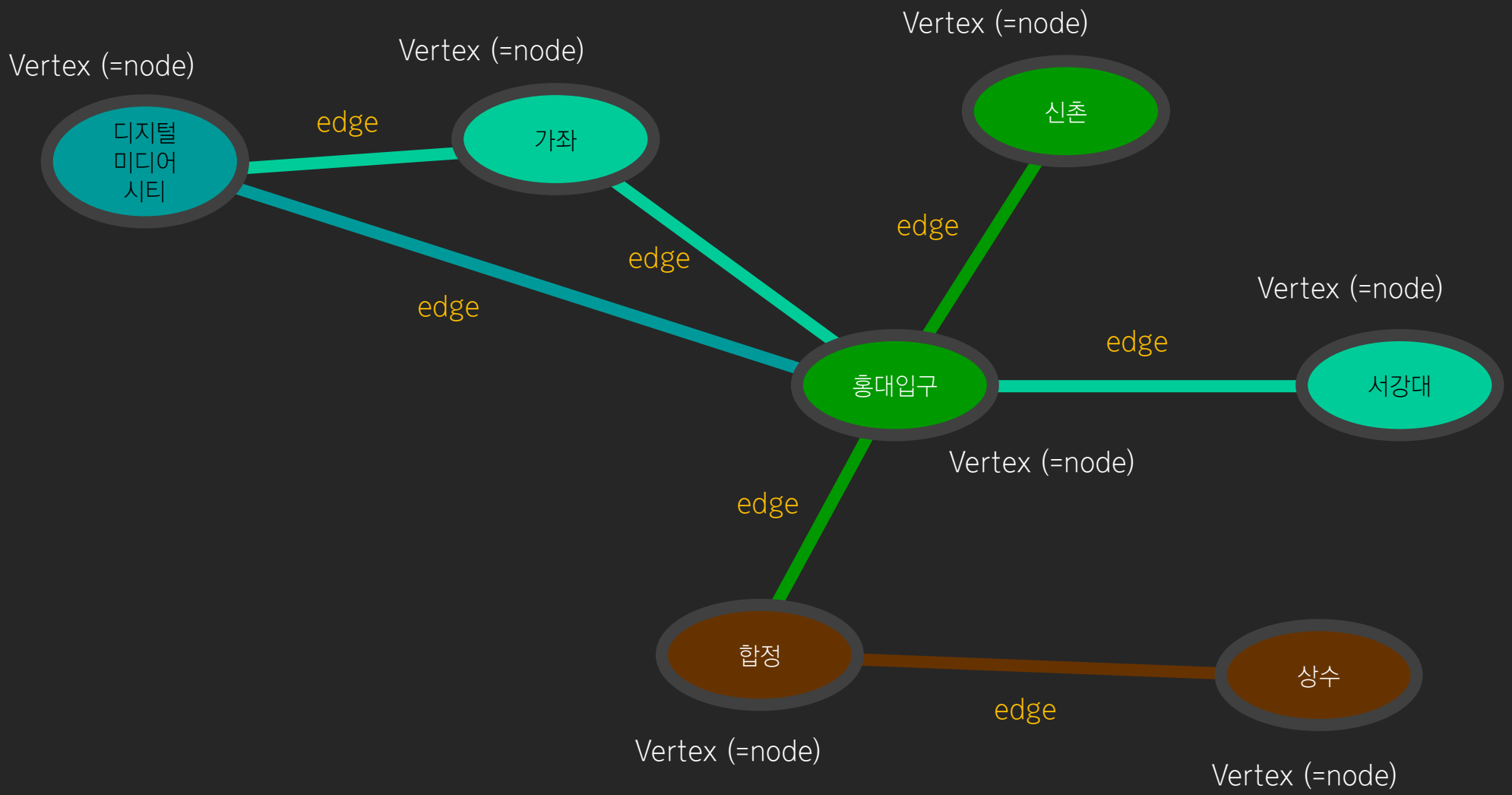
그래프 실생활 예

- SNS / 메신저

정점 Vertex (Node)

간선 Edge




Vertex (=node)
디지털
미디어
시티
edge
Vertex (=node)
가좌
edge
edge
Vertex (=node)
신촌
edge
Vertex (=node)
서강대
edge
홍대입구
Vertex (=node)
edge
합정
Vertex (=node)
edge
상수
Vertex (=node)

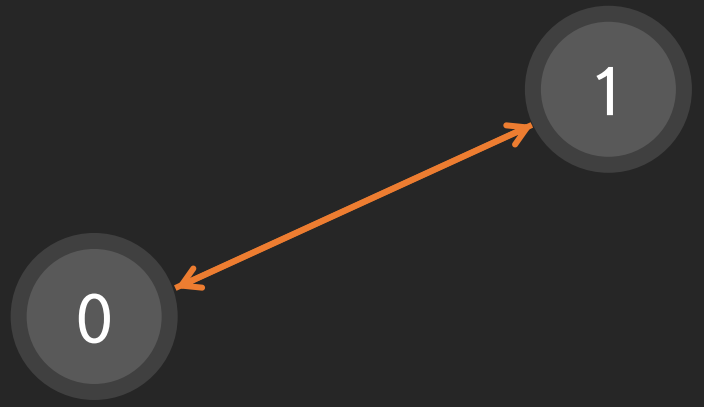
그래프 Graph

- 관련 용어
  - 방향
  - 순환
  - 연결요소
  - 가중치
  - 차수
  - 단순 경로 등등…

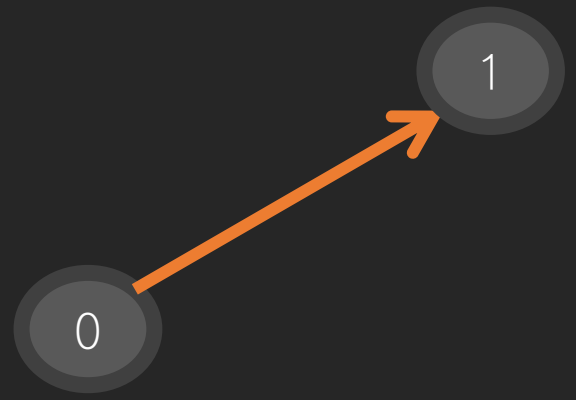
방향 Direction

무방향 그래프: 방향이 없다

= 양방향 그래프

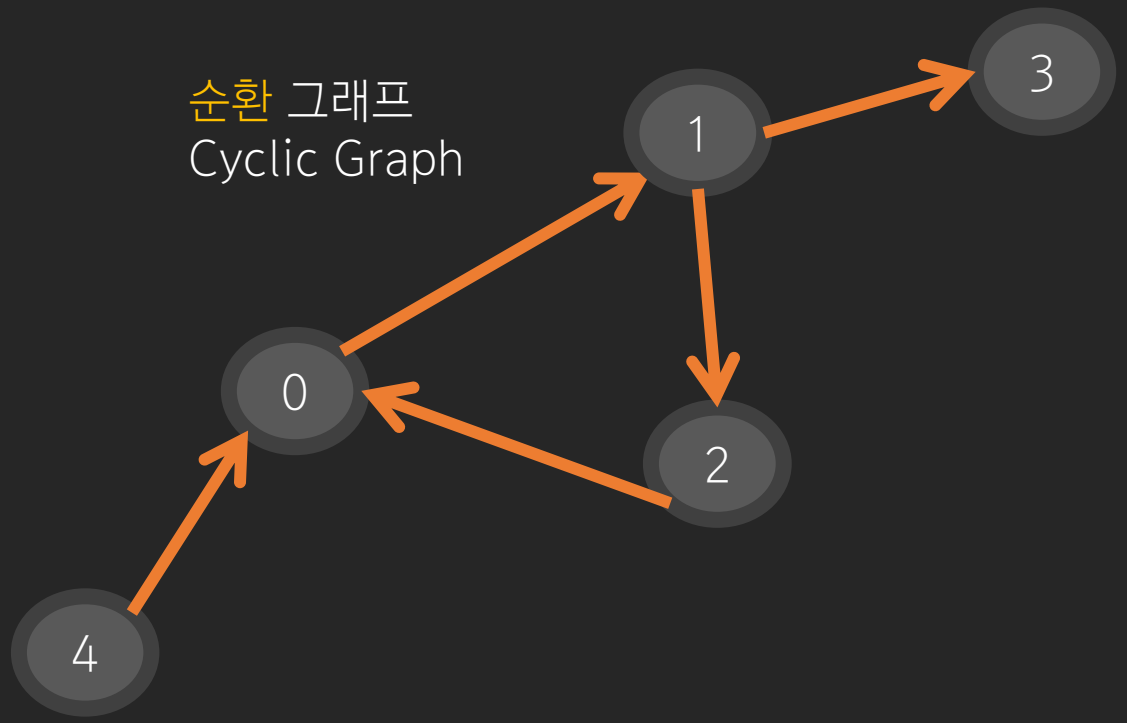


방향 그래프: 방향이 있다

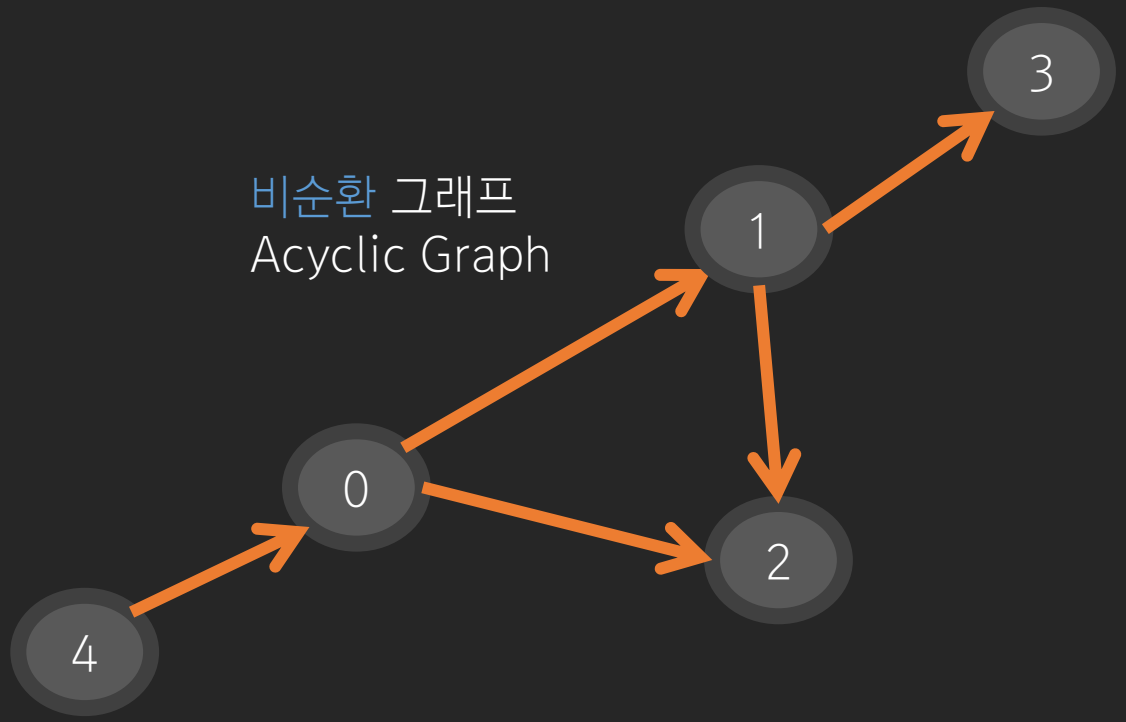


순환 Cycle

순환 그래프
Cyclic Graph

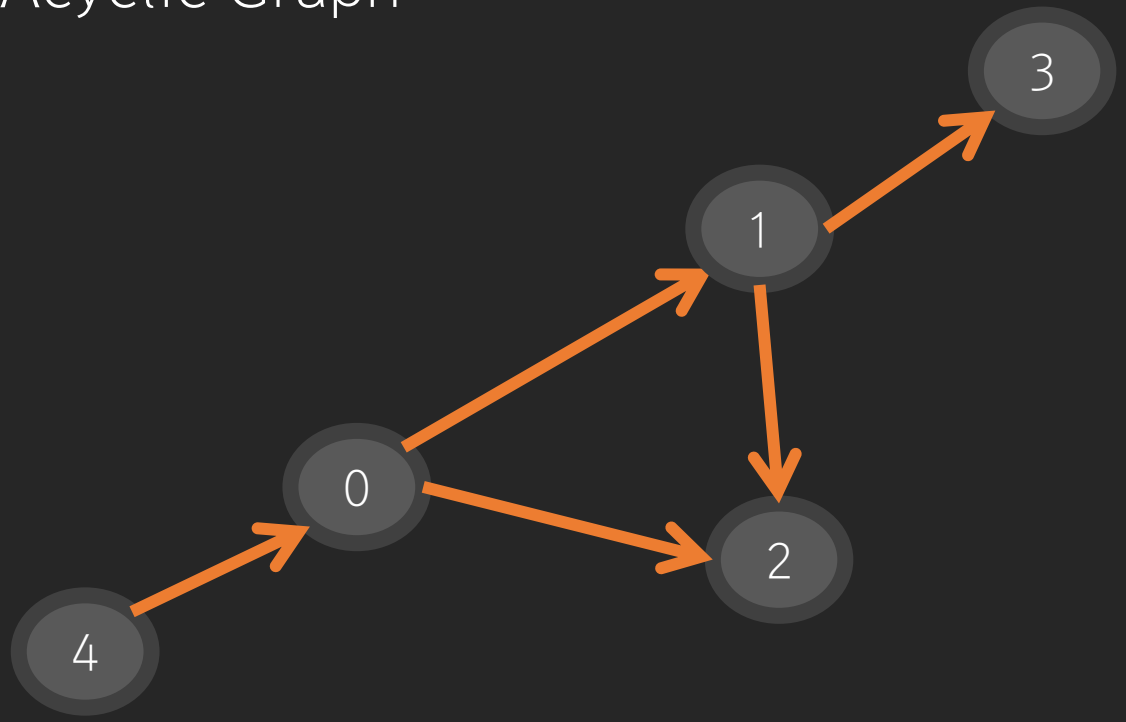


비순환 그래프
Acyclic Graph

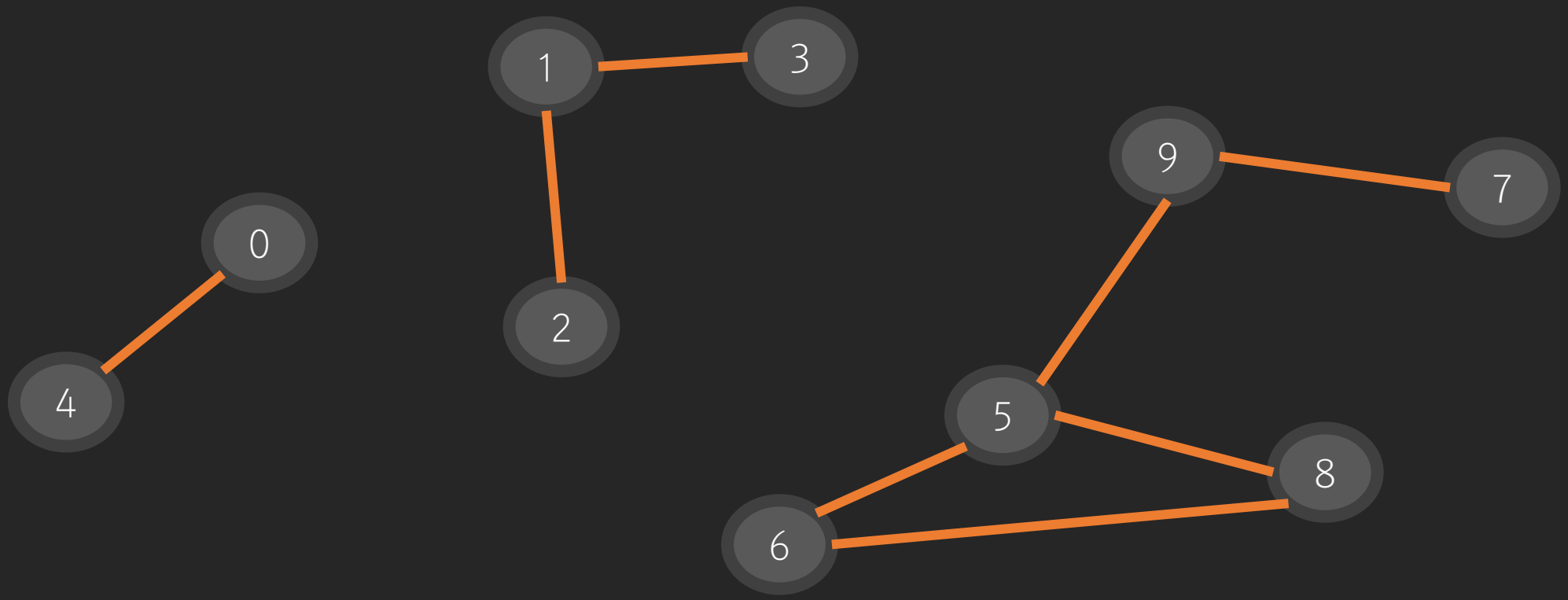


방향성 비순환 그래프

DAG Directed Acyclic Graph



연결 요소 Connected Component



그래프 Graph

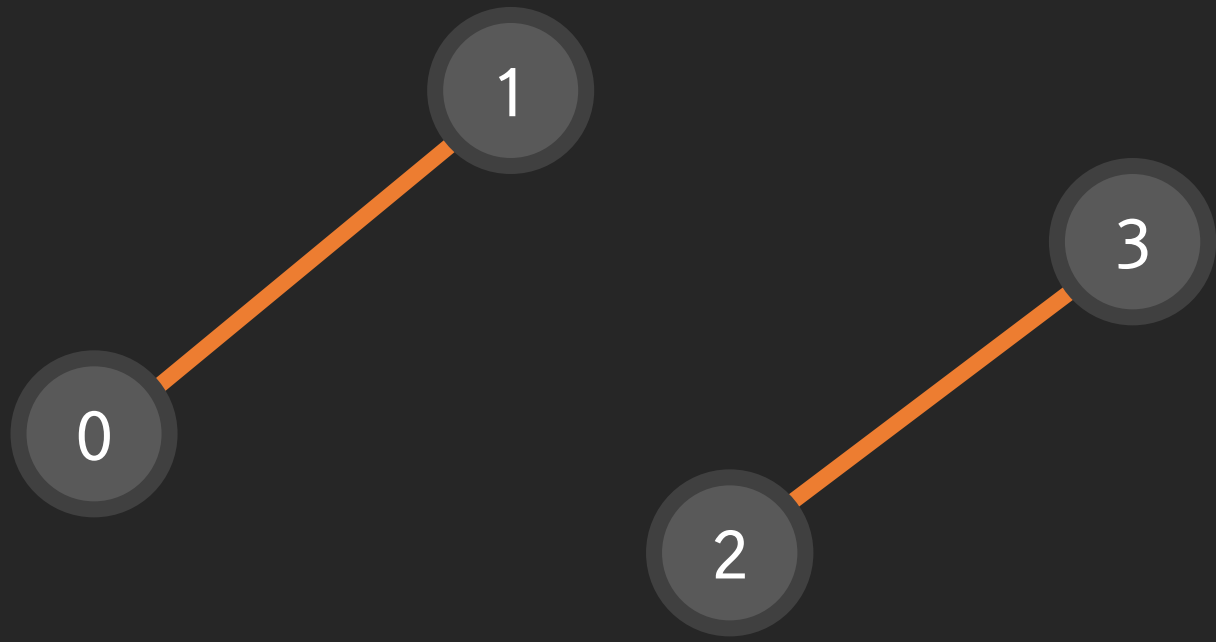
정점 개수 : 4개

간선 개수 : 2개

방향 : 무방향

순환 : 순환

연결 요소 : 2개



그래프 Graph

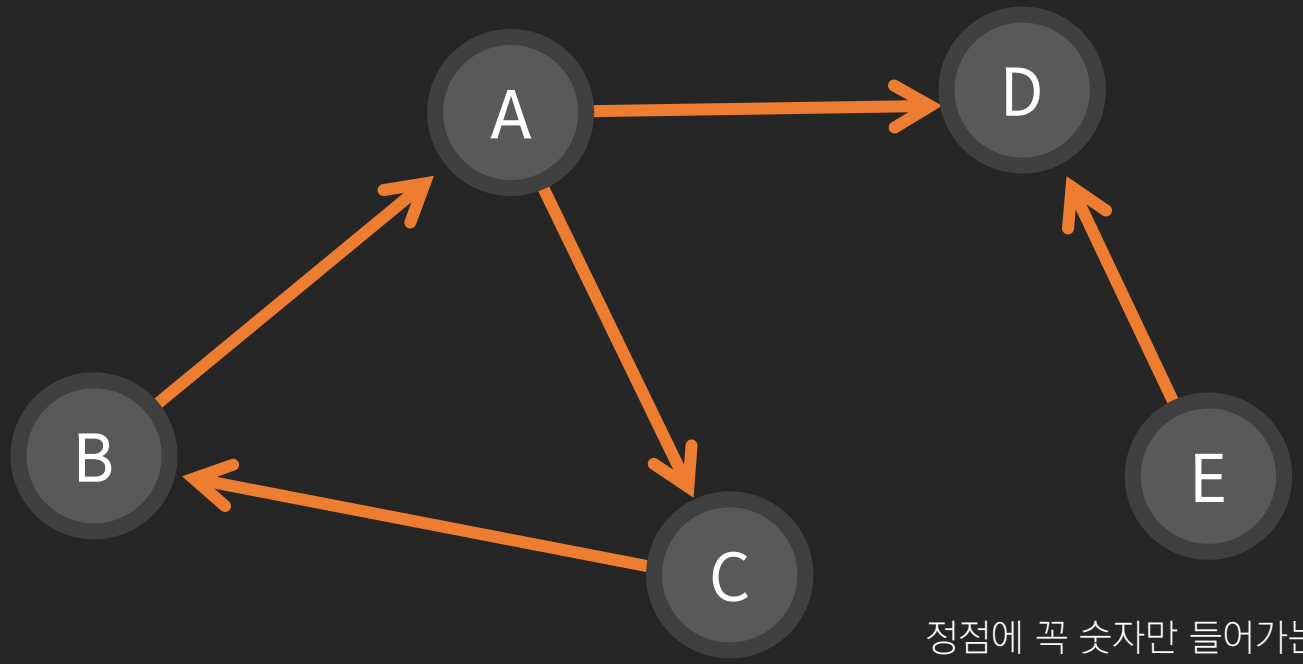
정점 개수 : 5개

간선 개수 : 5개

방향 : 단방향

순환 : 순환

연결 요소 : 1개



정점에 꼭 숫자만 들어가는 건 아니다!

그래프 Graph

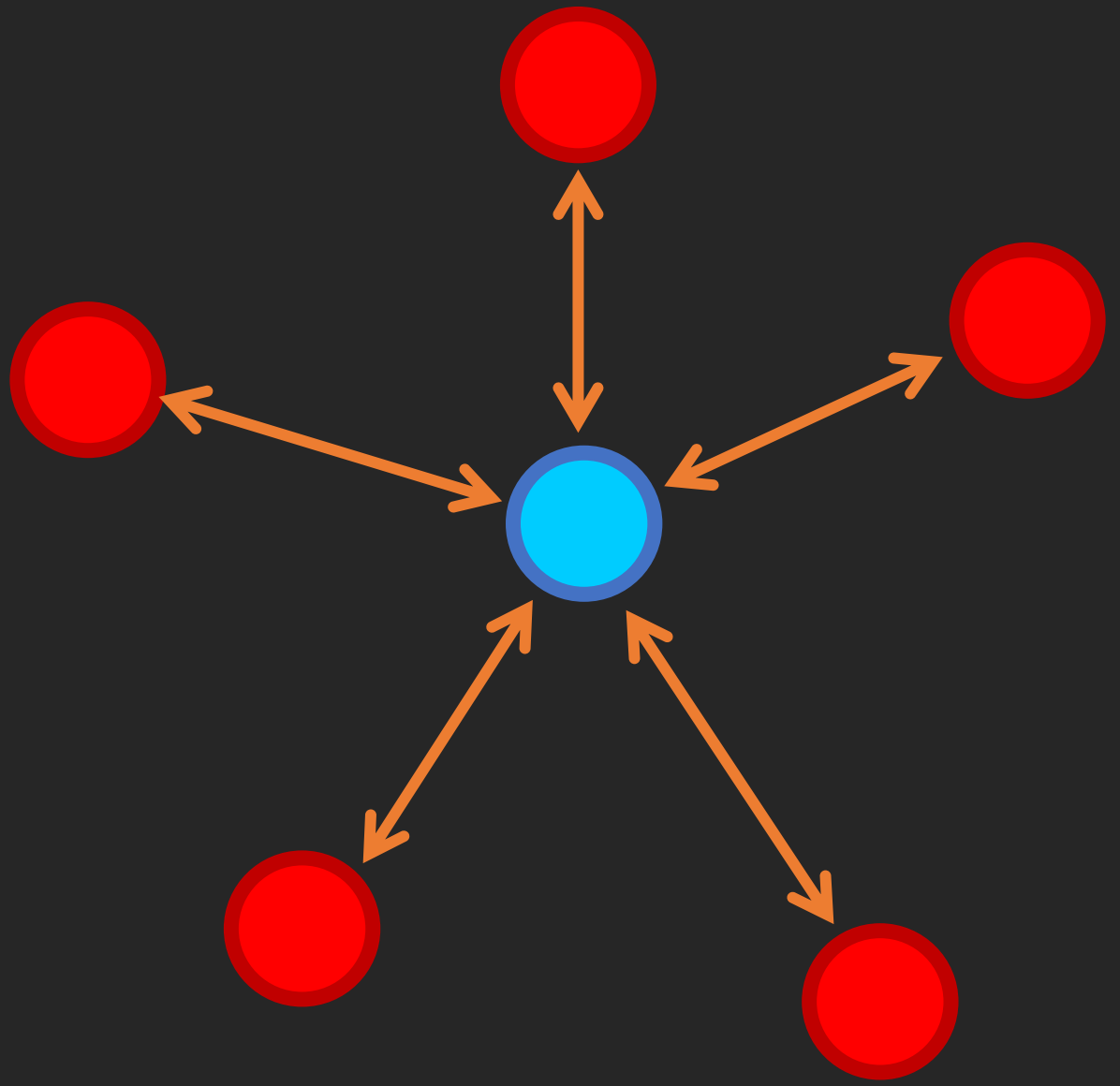
정점 개수 : 6개

간선 개수 : 5개

방향 : 양방향

순환 : 순환

연결 요소 : 1개



정점에 값 대신 색깔이 들어 갈 수도 있다.

그래프 Graph

정점 개수 : 1개

간선 개수 : 0개

방향 : ?

순환 : 비순환

연결 요소 : 1개

그래프 Graph

정점 개수 : 0개

간선 개수 : 0개

방향 : ?

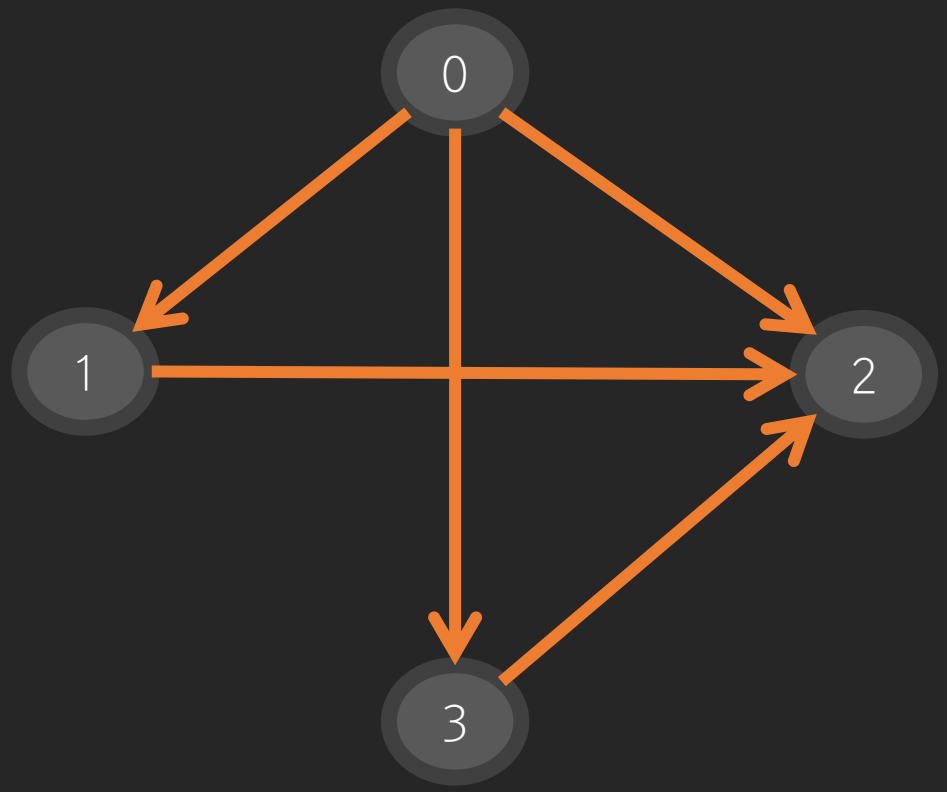
순환 : 비순환

연결 요소 : 0개

= 빈 그래프

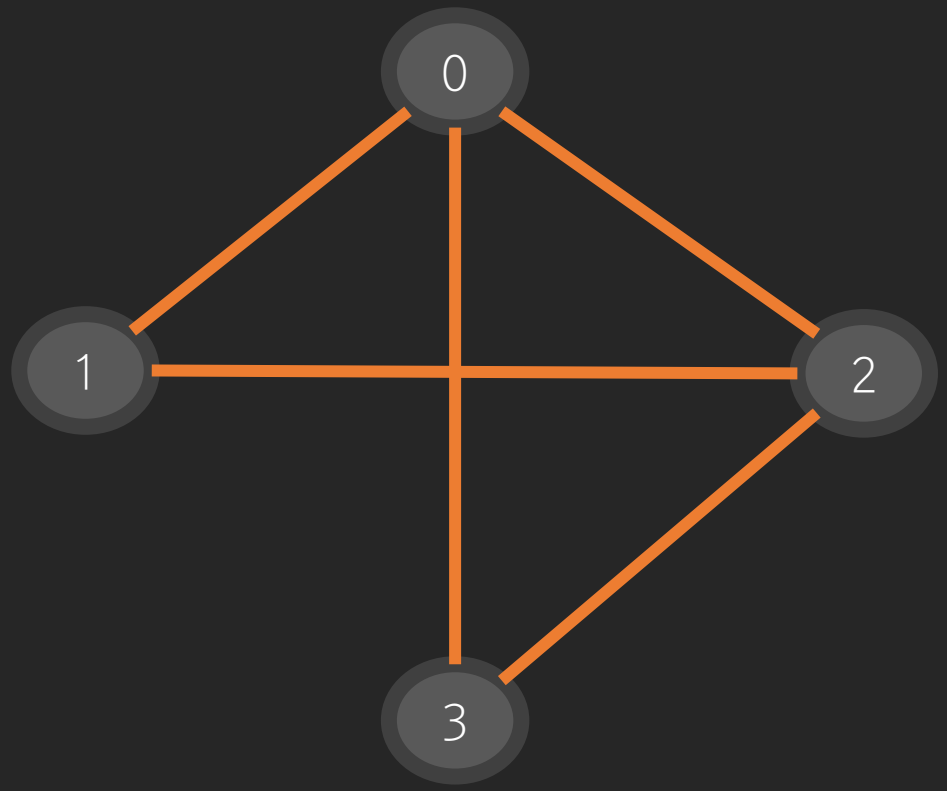
코드로 그래프를 나타내는 방법

1. 인접행렬



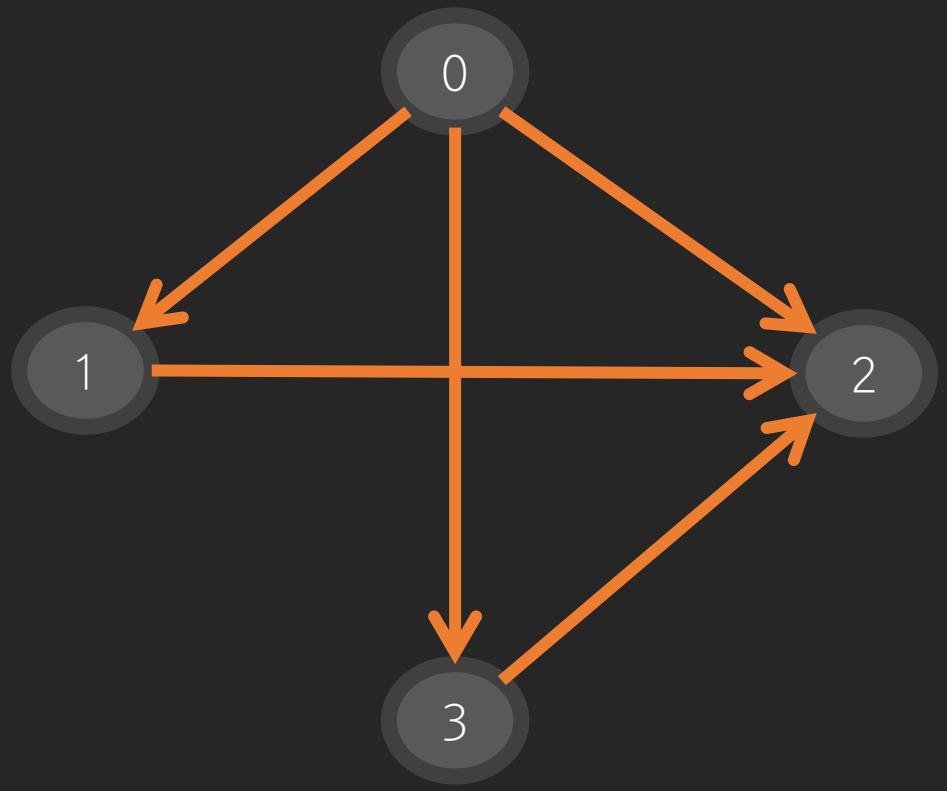
	0	1	2	3
0	0	1	1	1
1	0	0	1	0
2	0	0	0	0
3	0	0	1	0

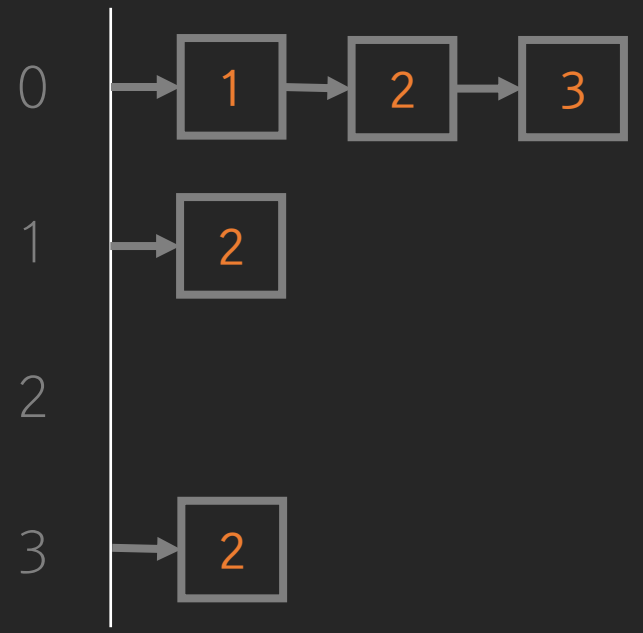
1. 인접행렬



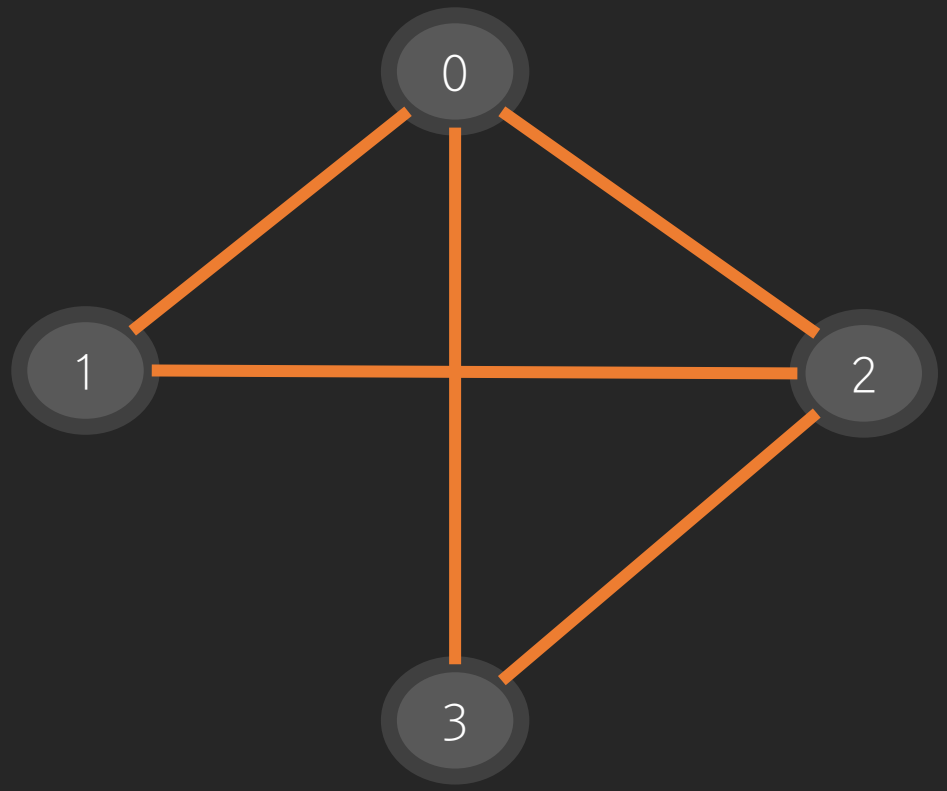
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

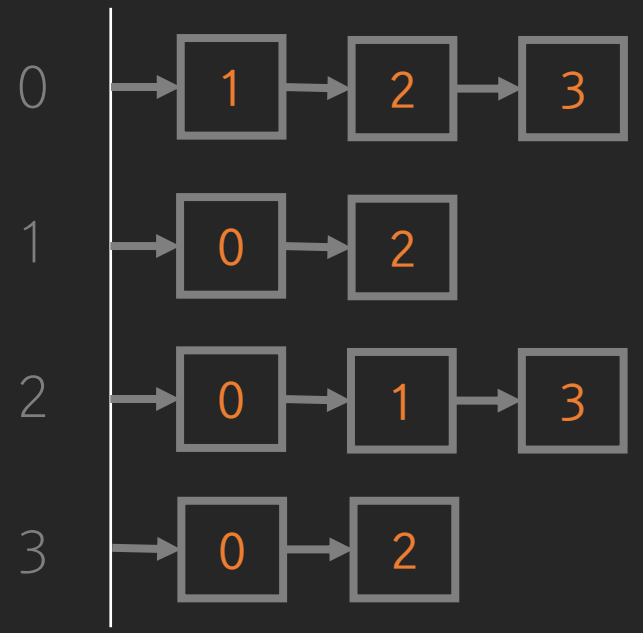
2. 인접리스트





2. 인접리스트





인접행렬 VS 인접리스트

- 인접행렬
  - 시간 복잡도 : $O(1)$
  - 공간 복잡도 : $O(V^2)$
- 인접리스트
  - 시간 복잡도 : $O(V)$
  - 공간 복잡도 : $O(V+E)$

간선이 많은 밀집 그래프일 때 쓰는게 좋다

정점이 아주 많으면 사용할 수 없다

백준 실버 수준 문제에서는 인접행렬,

인접리스트 2개중 아무거나 사용해도 OK

간선이 적은 희소 그래프일 때 쓰는게 좋다

정점이 아주 많을 때도 사용할 수 있다

완전탐색

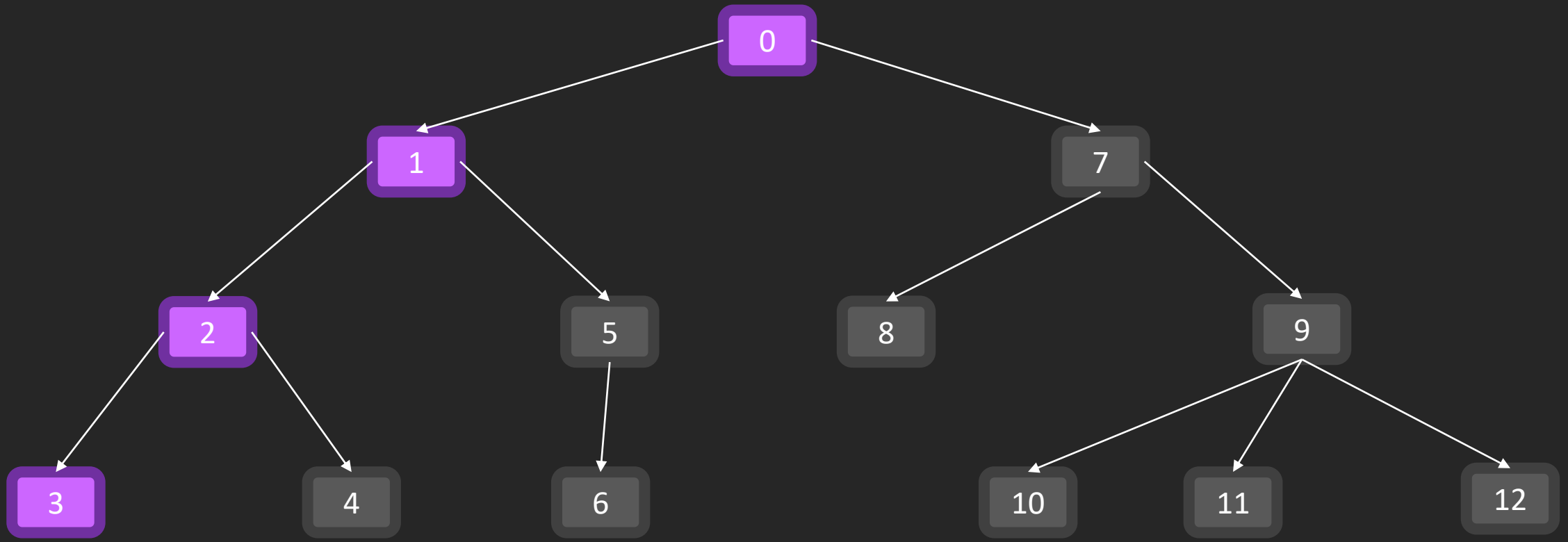
DFS

Depth First Search

깊이 우선 탐색

DFS Depth First Search 깊이 우선 탐색

- 스택 or 재귀를 사용해서 구현한다



DFS 의사코드

```
방문_배열 선언
방문_배열[시작_지점] = true

DFS(시작_지점)

함수 DFS(현재_지점) {
    // 필요한 연산 처리 후

    for 다음_지점 in 1 ~ N {
        if 정점_연결_확인(다음_지점) == false {
            continue
        }

        if 방문_배열[다음_지점] {
            continue
        }

        방문_배열[다음_지점] = true
        DFS(다음_지점)
    }
}
```

완전탐색

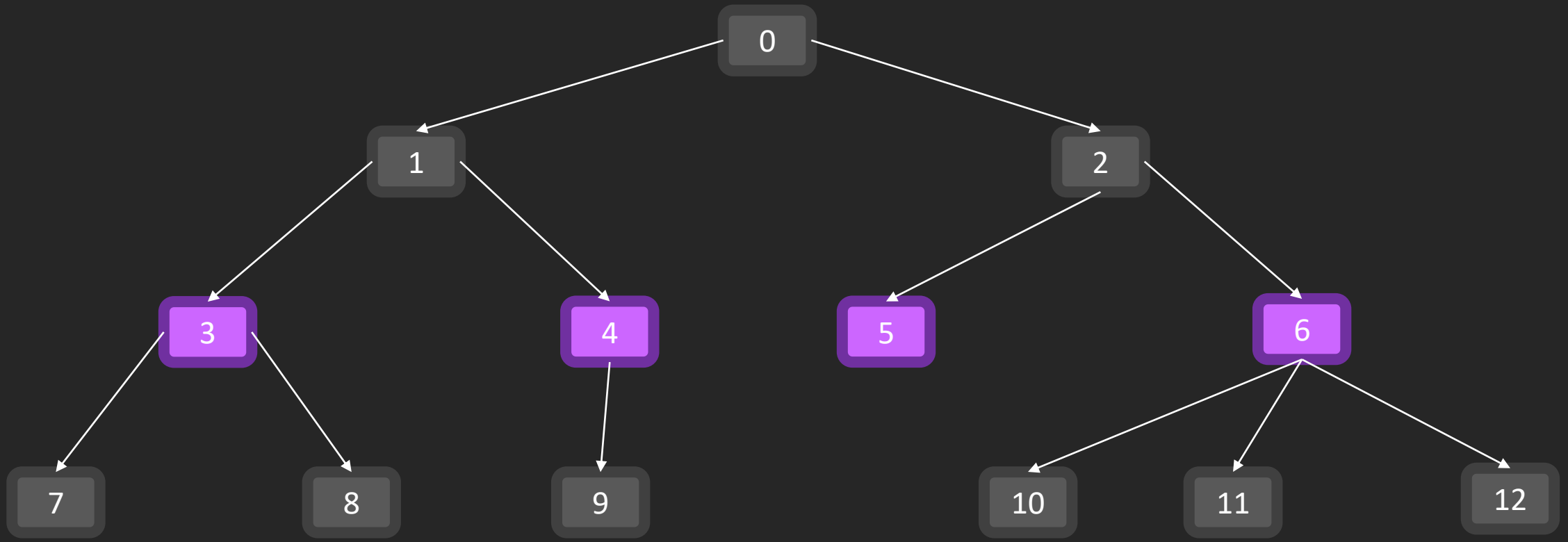
BFS

Breadth First Search

너비 우선 탐색

BFS Breadth First Search 너비 우선 탐색

- 큐를 사용해서 구현한다



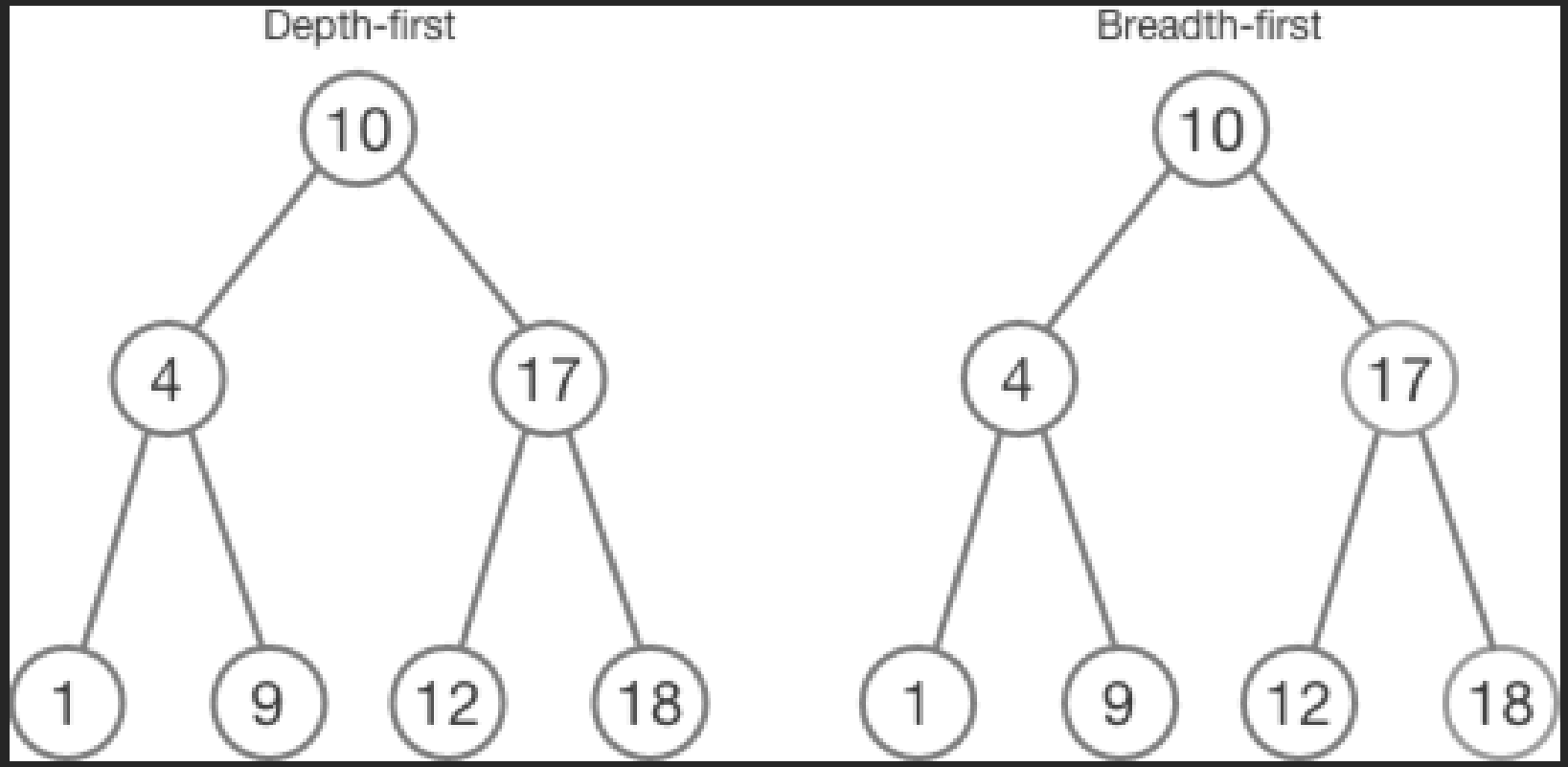
BFS 의사코드

```
함수 BFS(시작_지점) {
    방문_배열 선언
    방문_배열[시작_지점] = true
    큐 선언
    큐.push(시작_지점)

    while 큐가 비어있지 않으면 {
        현재_지점 = 큐.pop()

        // 필요한 연산 처리 후

        for 다음_지점 in 1 ~ N {
            if 정점_연결_확인(다음_지점) == false {
                continue
            }

            if 방문_배열[다음_지점] {
                continue
            }

            방문_배열[다음_지점] = true
            큐.push(다음_지점)
        }
    }
}
```

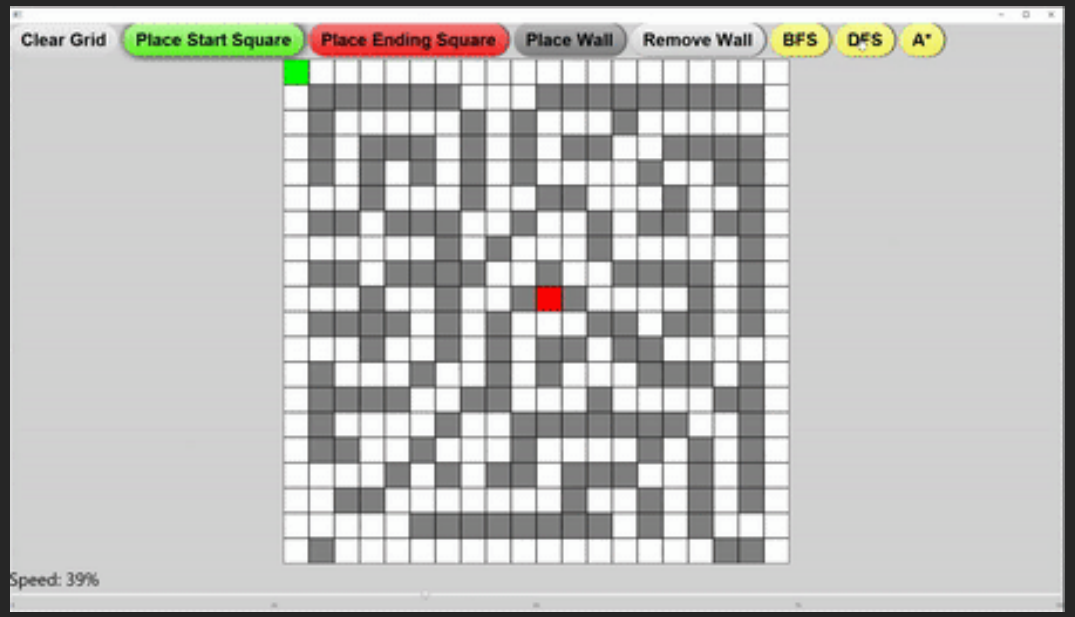
DFS & BFS

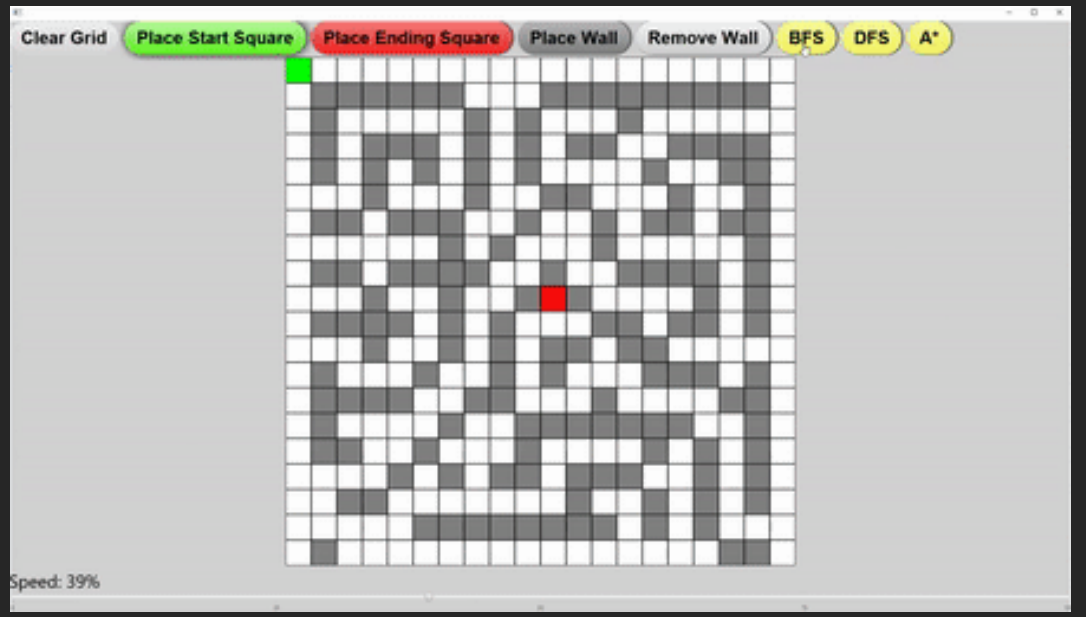
- 공통점
  - 완전 탐색이다
  - 모든 경우의 수를 살펴보는 것이면 DFS/BFS 아무거나 사용해도 괜찮다!
- 차이점
  - BFS에서 찾은 노드는 최단 거리의 노드임을 보장한다

DFS & BFS



길찾기 문제


Clear Grid
Place Start Square
Place Ending Square
Place Wall
Remove Wall
BFS
DFS
A*
Speed: 39%


Clear Grid
Place Start Square
Place Ending Square
Place Wall
Remove Wall
BFS
DFS
A*
Speed: 39%

길찾기 문제

- → ↓ ← ↑ 4방향
- 범위 체크 & 방문 체크 필요

```
dy = (0, 1, 0, -1)
dx = (1, 0, -1, 0)
chk = [[False] * 100 for _ in range(100)]

def is_valid_coord(y, x):
    return 0 <= y < 100 and 0 <= x < 100

def dfs(y, x):
    for i in range(4):
        ny = y + dy[i]  # next y, x
        nx = x + dx[i]
        if is_valid_coord(ny, nx) and not chk[ny][nx]:
            chk[ny][nx] = True
            dfs(ny, nx)

chk[sy][sx] = True  # start y, x
dfs(sy, sx)
```

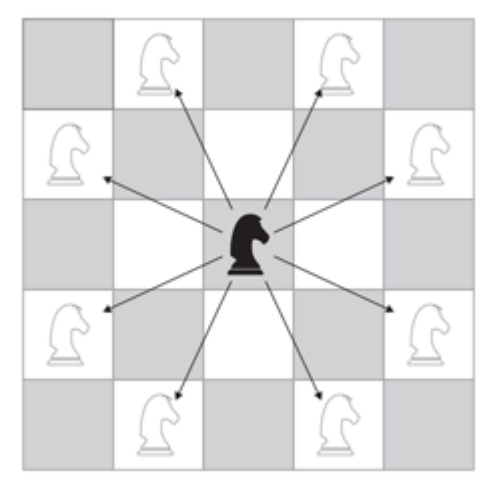
길찾기 문제

- → ↓ ← ↑ 4방향

```
dy = (0, 1, 0, -1)
dx = (1, 0, -1, 0)

def is_valid_coord(y, x):
    return 0 <= y < 100 and 0 <= x < 100

def bfs(sy, sx):  # start y, x
    chk = [[False] * 100 for _ in range(100)]
    chk[sy][sx] = True
    q = deque()
    q.append((sy, sx))
    while q:
        y, x = q.popleft()
        for i in range(4):
            ny = y + dy[i]  # next y, x
            nx = x + dx[i]
            if is_valid_coord(ny, nx) and not chk[ny][nx]:
                chk[ny][nx] = True
                q.append((ny, nx))
```

길찾기 문제

- dy, dx 배열 안쓰고도 구현은 가능, but
  - 코드가 길어진다
  - 코딩실수 날 가능성도 있다



dy = [-2, -1, 1, 2, 2, 1, -1, -2]
dx = [1, 2, 2, 1, -1, -2, -2, -1]

```python
chk = [[False] * 100 for _ in range(100)]

def is_valid_coord(y, x):
    return 0 <= y < 100 and 0 <= x < 100

def dfs(y, x):
    ny = y + 0  # →로 가본다
    nx = x + 1
    if is_valid_coord(ny, nx) and not chk[ny][nx]:
        chk[ny][nx] = True
        dfs(ny, nx)

    ny = y + 1  # ↓로 가본다
    nx = x + 0
    if is_valid_coord(ny, nx) and not chk[ny][nx]:
        chk[ny][nx] = True
        dfs(ny, nx)

    ny = y + 0  # ←로 가본다
    nx = x + -1
    if is_valid_coord(ny, nx) and not chk[ny][nx]:
        chk[ny][nx] = True
        dfs(ny, nx)

    ny = y + -1  # ↑로 가본다
    nx = x + 0
    if is_valid_coord(ny, nx) and not chk[ny][nx]:
        chk[ny][nx] = True
        dfs(ny, nx)

chk[sy][sx] = True  # start y, x
dfs(sy, sx)
```

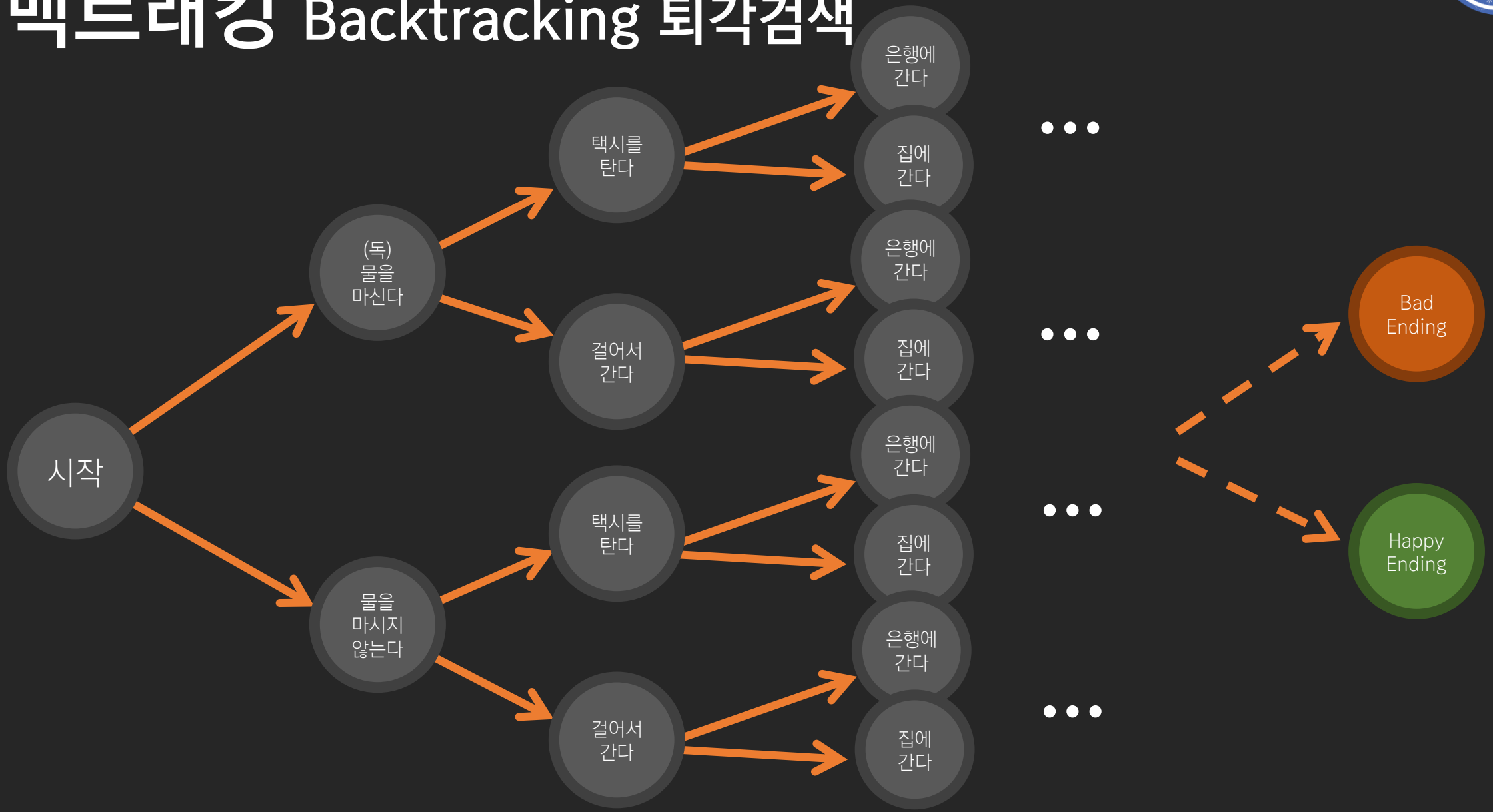
알고리즘 설계 기법

백트래킹

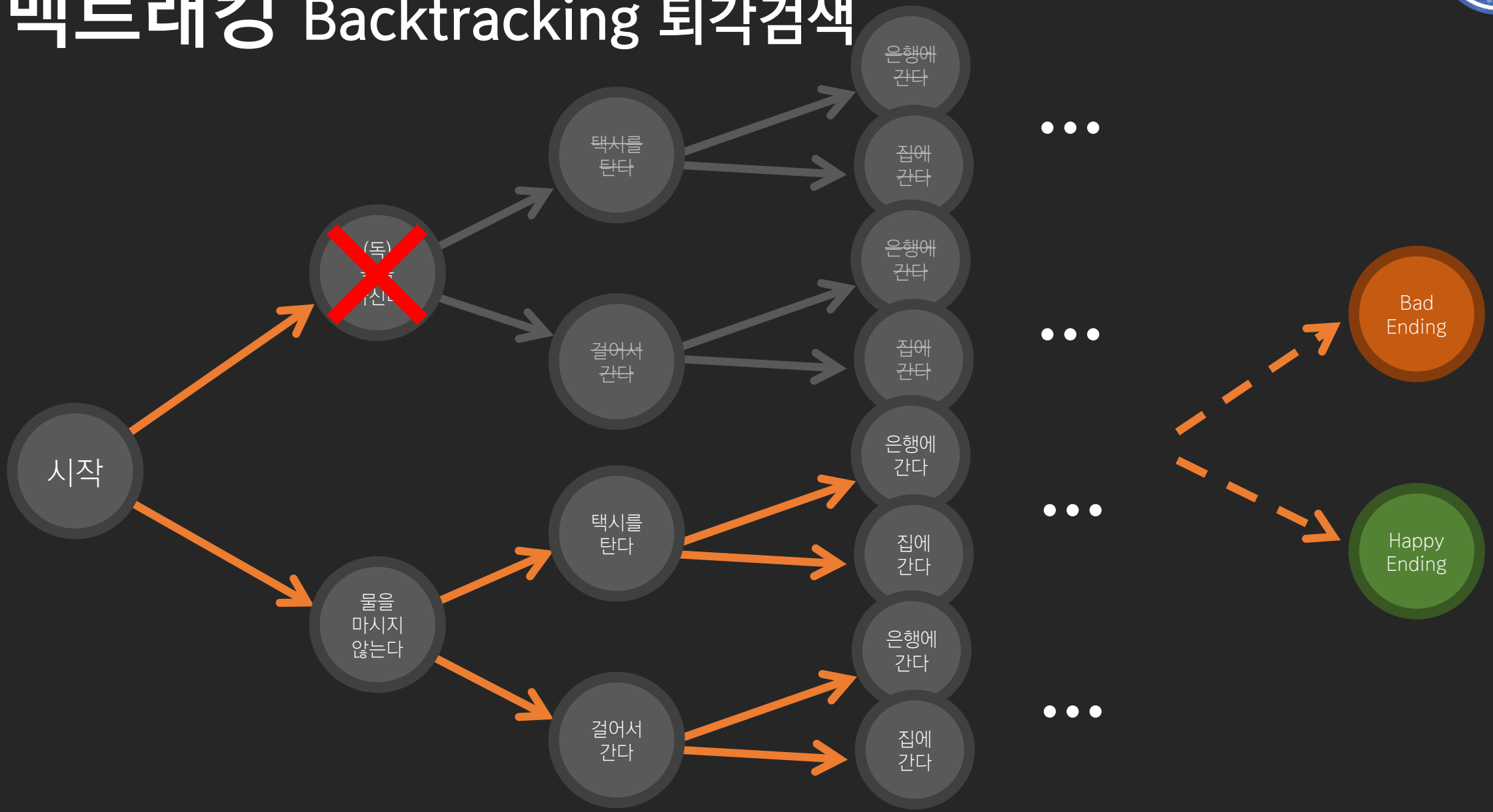
Backtracking

퇴각검색

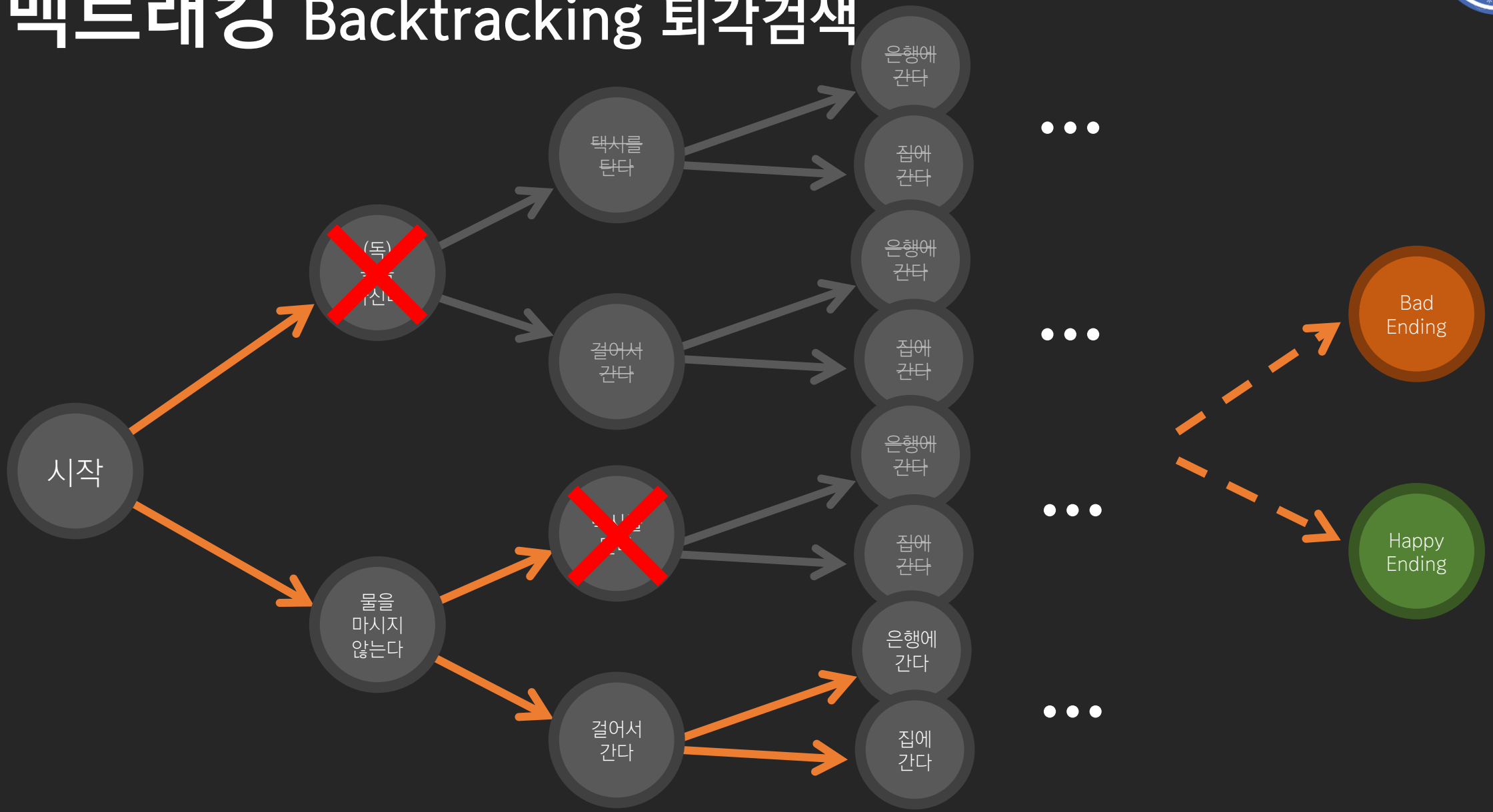
백트래킹 Backtracking 퇴각검색



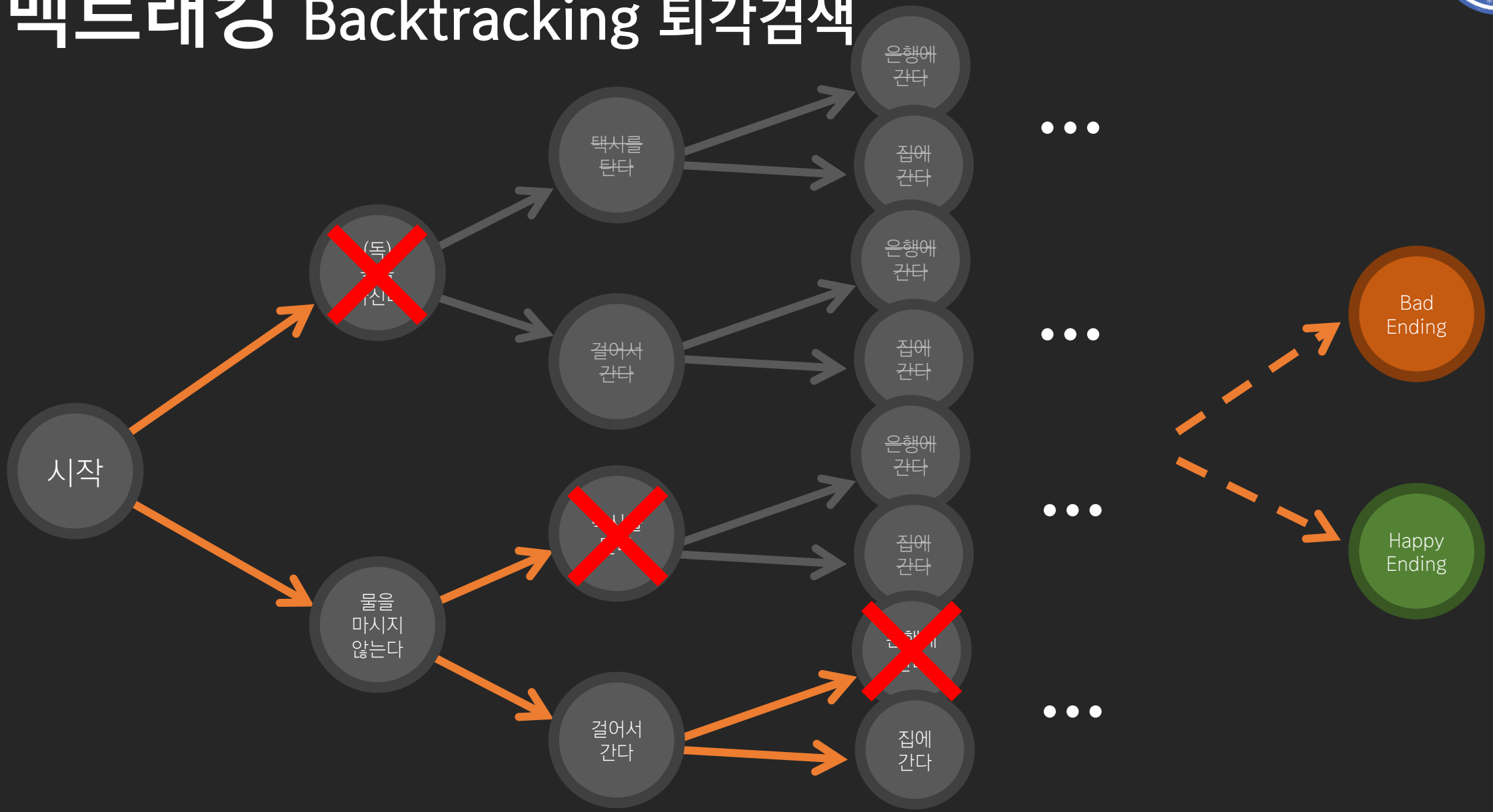
백트래킹 Backtracking 퇴각검색



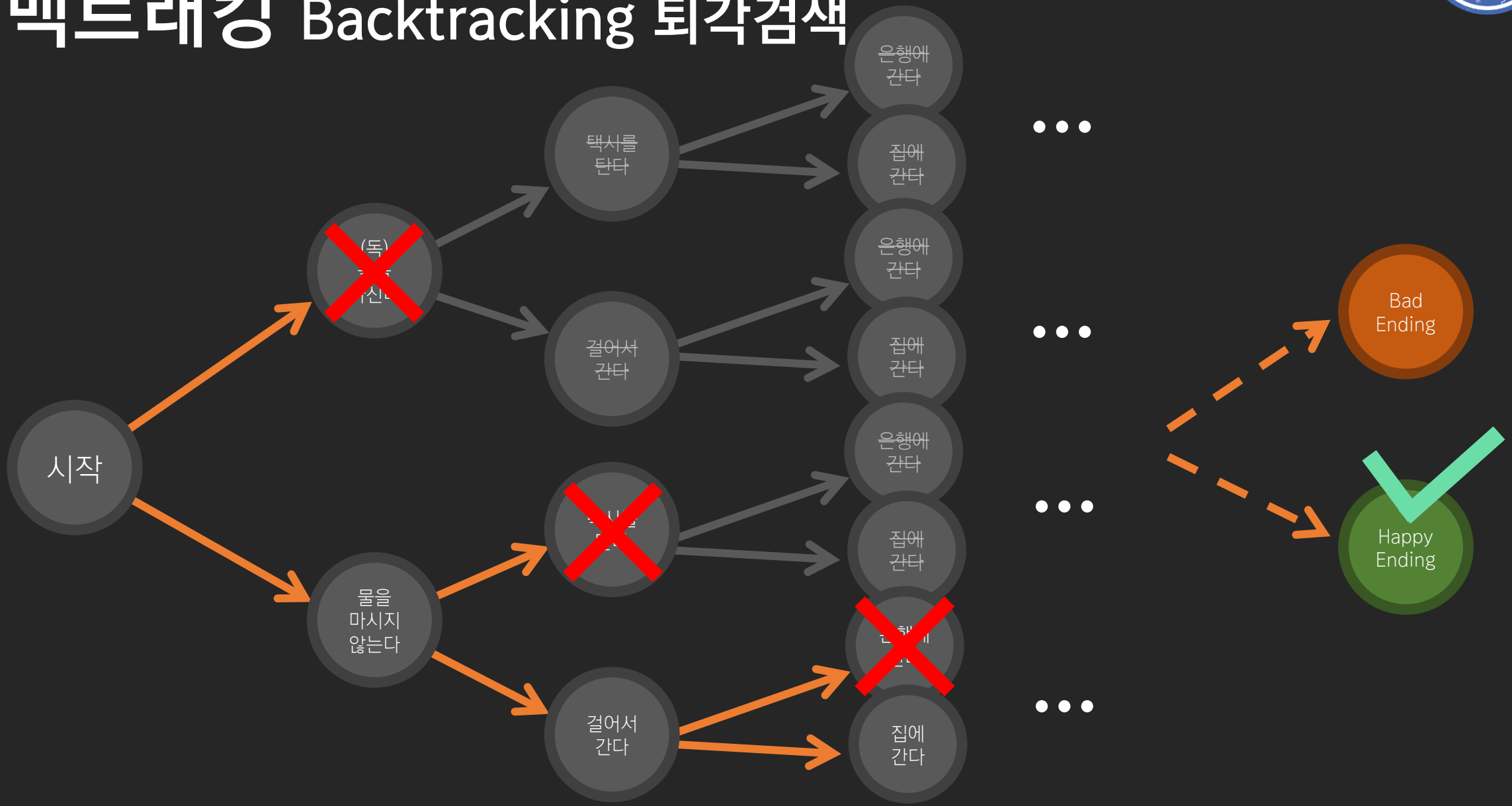
백트래킹 Backtracking 퇴각검색



백트래킹 Backtracking 퇴각검색



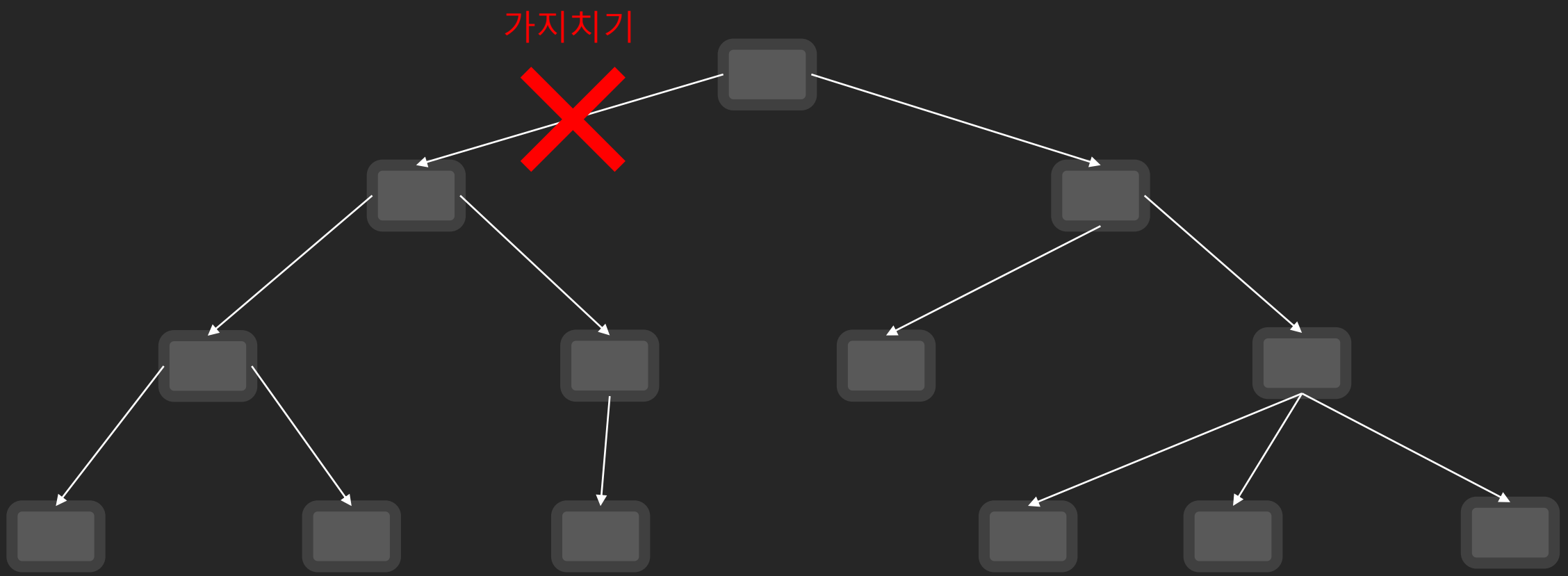
백트래킹 Backtracking 퇴각검색



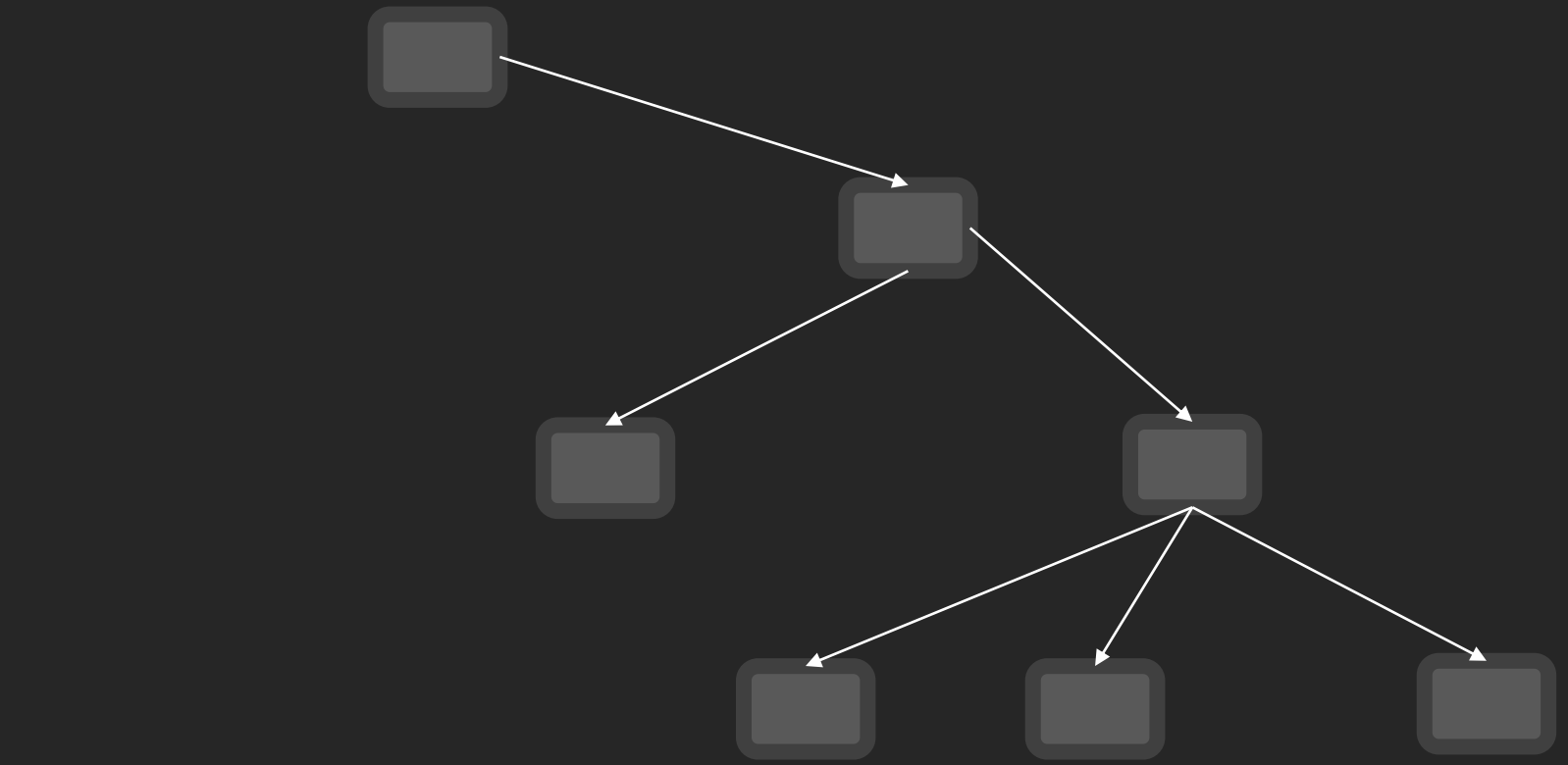
백트래킹 Backtracking 퇴각검색

- 완전 탐색시 볼 필요 없는 경우들을 줄여주는 알고리즘 기법
- 백트래킹은 가지치기를 통해 탐색 경우의 수를 줄인다
  - 최악의 경우, 모든 경우를 다 살펴보게 될 수도 있지만 가능한 덜 보겠다는 전략
  - '가망성이 없으면 가지 않는다'
- DFS & BFS 둘 다 백트래킹을 통해 경우의 수를 줄여 줄 수 있다.
  - 보통은 DFS 백트래킹을 자주 사용한다

백트래킹 Backtracking 퇴각검색



백트래킹 Backtracking 퇴각검색



지금까지 배운 것

- 알고리즘
  - 완전 탐색 & 조합 탐색
    - 브루트포스
    - 순열 & 조합
    - DFS & BFS
  - 백트래킹
- 자료구조
  - 배열, 연결 리스트
  - 스택, 큐, 덱, 우선순위 큐
  - 맵, 집합
  - 그래프 (인접행렬, 인접리스트)

라이브 코딩

Live Coding

라이브 코딩

11724. 연결 요소의 개수

라이브 코딩

2178. 미로 탐색

정리

- 그래프는 정점과 간선으로 이루어져 있다
- 방향성과 순환성이 각각 있거나 없거나
- 연결요소
- 수많은 그래프 종류와 알고리즘들이 존재한다

정리

- 그래프를 코드로 구현할 때는 2가지 방법이 있다
  1. 인접행렬
     - 간선이 많은 그래프일 때 쓰는게 좋다
     - 탐색이 빠르다
  2. 인접리스트
     - 간선이 적은 그래프일 때 쓰는게 좋다
     - 메모리를 적게 쓴다

정리

- DFS, BFS은 완전탐색 알고리즘
  - 최악의 경우 모든 노드를 탐색하는 건 동일
- 최단거리를 구할 때는 BFS 사용
- DFS는 재귀 (or 스택), BFS는 큐로 구현
- 완전 탐색에서 가지치기를 하면 백트래킹

과제

1. Prev Mission: 라이브 코딩 문제들 바로 다음날까지 풀기
2. Signature Mission: 문제들 풀이법 및 pseudo code 일지 작성
    - 실 구현은 할 필요 X