10주차 DFS / BFS

강사 : 한경수

Graph(그래프)

3가지를 사용하는데 필요한 그래프에 대해 알아보자!

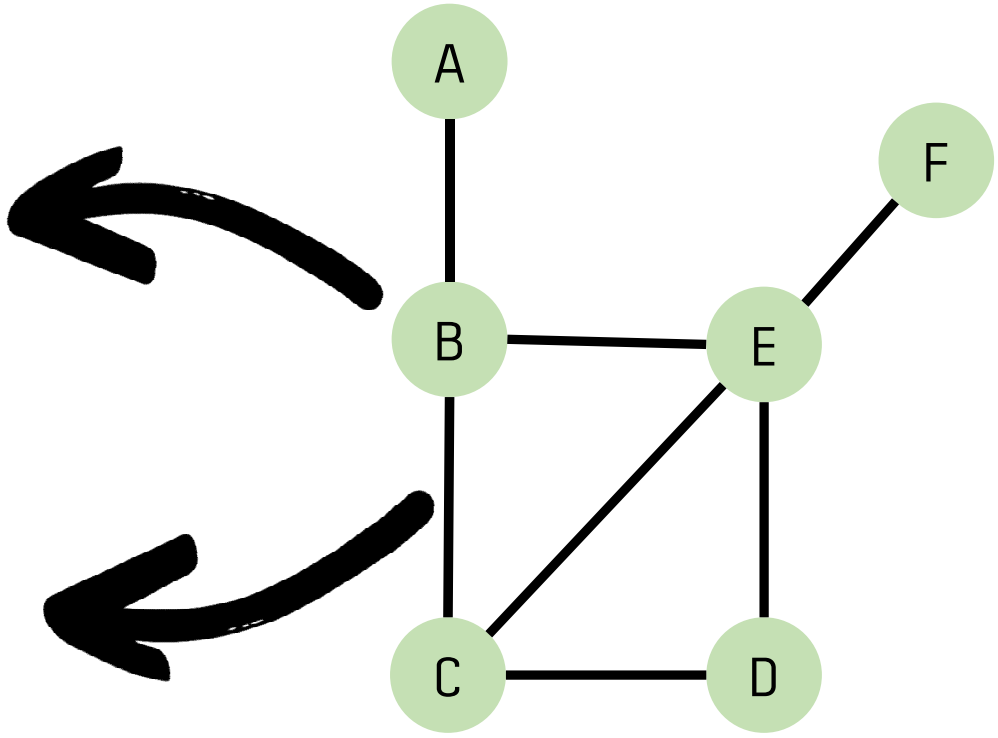
Graph(그래프)

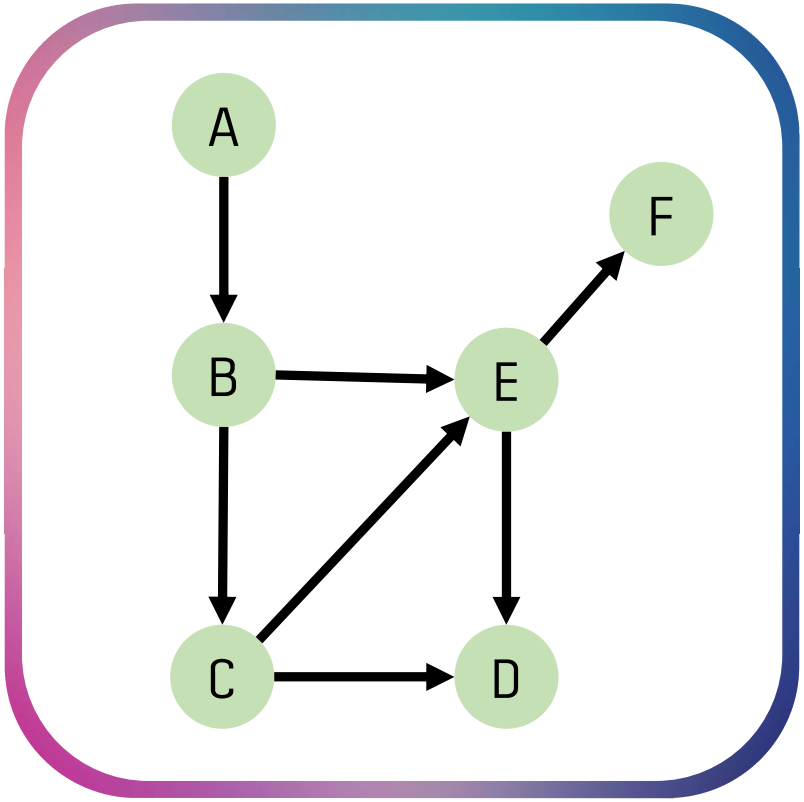
노드(N,node)

= 위치
알고리즘에서는 정점(vertex)

간선(E,edge)

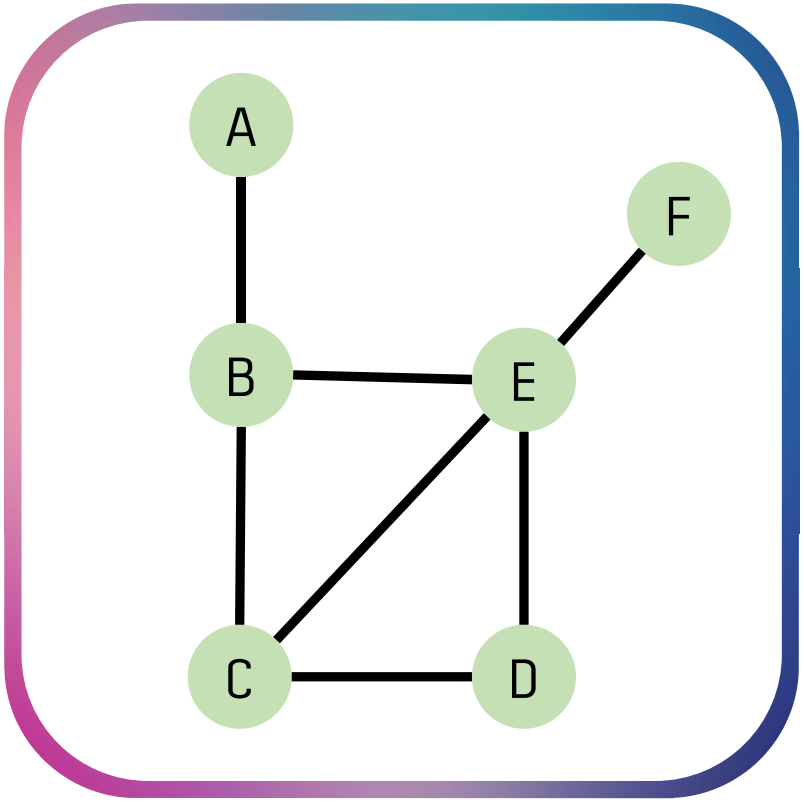
노드들을 연결하는 선





Directed Graph

방향성 O



Undirected Graph

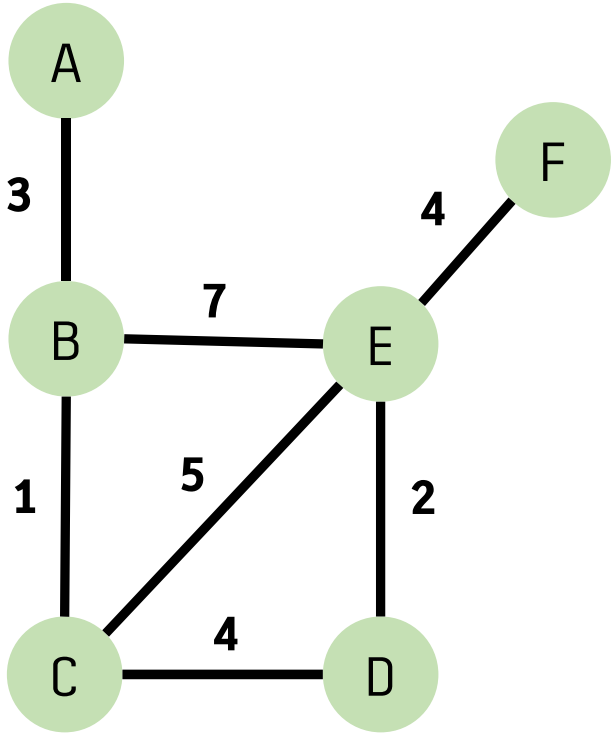
방향성 X

Graph(그래프)

가중치 그래프_Network

그래프의 간선에 비용이나 가중치가 할당된 그래프

Ex) 도로를 지나가는데 드는 비용, 도로의 길이 …



Graph(그래프)

그래프를 표현할 수 있는 2가지 방법

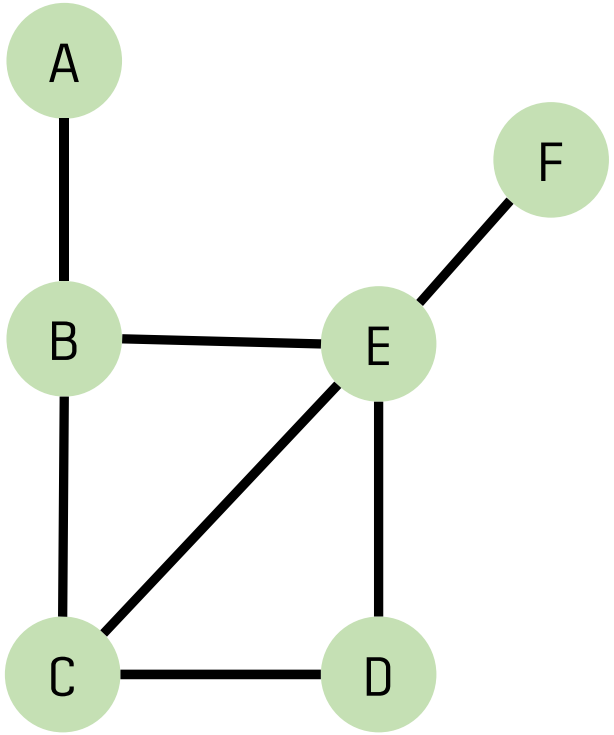
인접 행렬 (배열)

인접 리스트 (연결리스트, 벡터)

Graph_인접행렬 (배열)

간선이 존재하면 “1” 그렇지 않으면 “0”

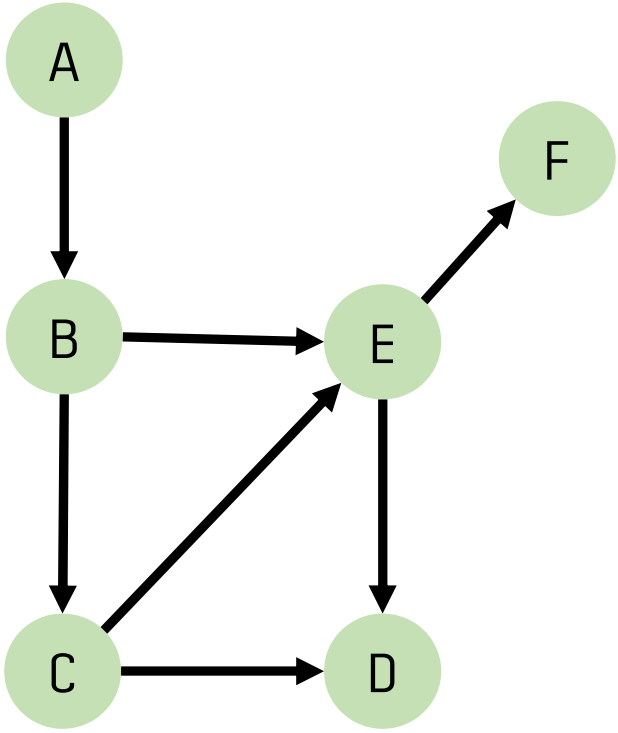
	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0



Graph_인접행렬 (배열)

간선이 존재하면 “1” 그렇지 않으면 “0”

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

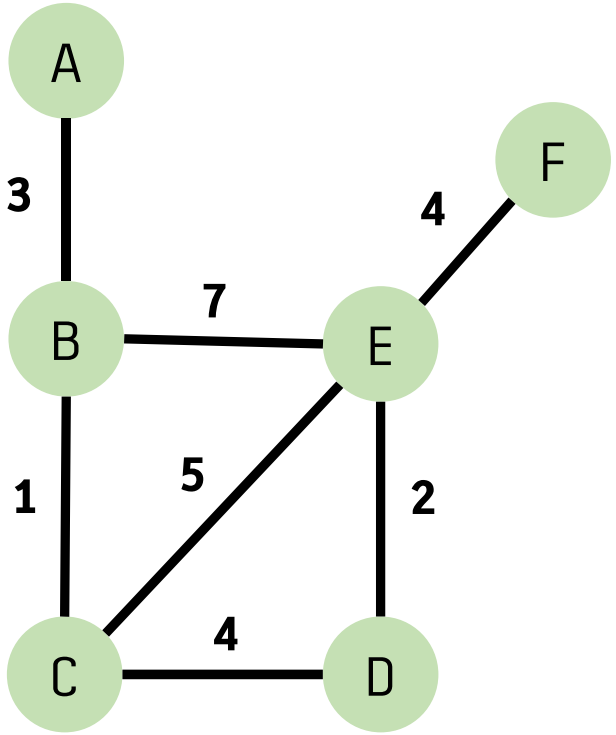


Graph_인접행렬 (배열)

가중치가 존재한다면..

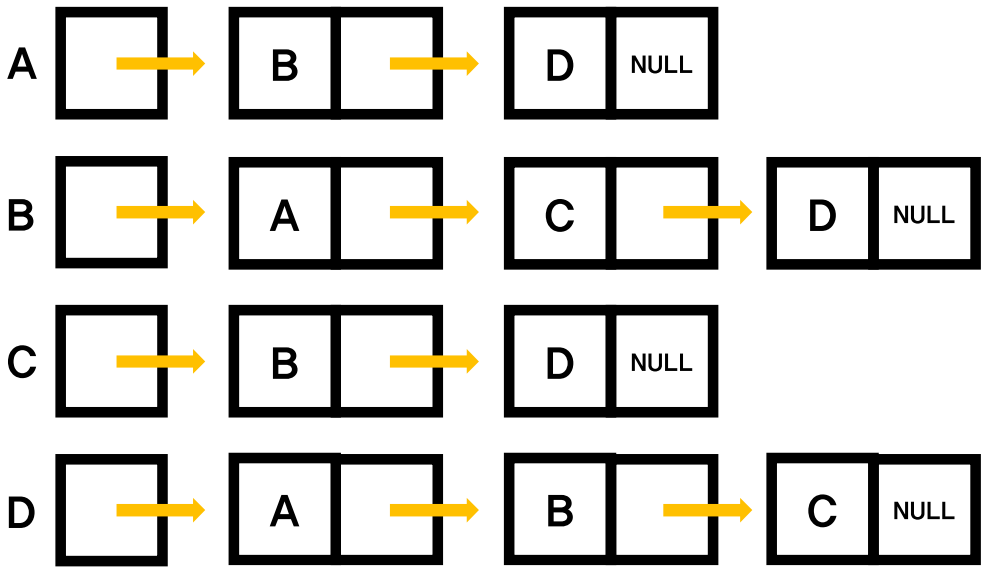
간선이 존재하면 "가중치" 그렇지 않으면 "0"

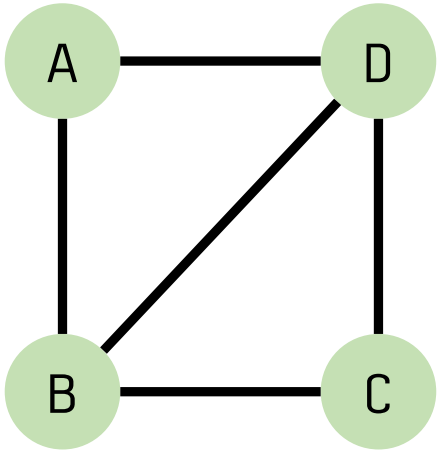
	A	B	C	D	E	F
A	0	3	0	0	0	0
B	3	0	1	0	7	0
C	0	1	0	4	5	0
D	0	0	4	0	2	0
E	0	7	5	2	0	4
F	0	0	0	0	4	0



Graph_인접리스트 (연결리스트)

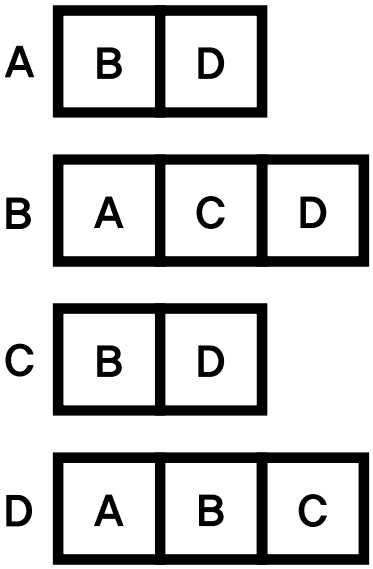
연결리스트를 사용하지만,
알고리즘에서는 잘 사용하지 않는다. PASS

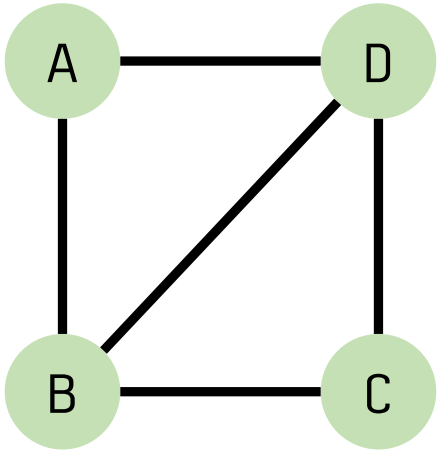




Graph_인접리스트 (Vector)

각각의 정점 벡터에 연결된 정점을 push_back

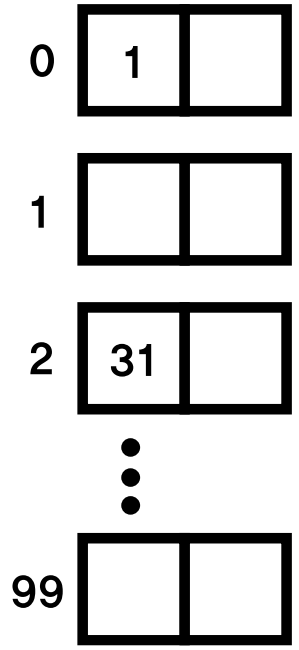




Graph_인접리스트 (Vector)

벡터 이용법?

```cpp
#include<iostream>
#include<vector> // 벡터 include
using namespace std;
int main() {
    vector<int> vec[100];
    vec[0].push_back(1); // 0번째 배열의 벡터에 1을 push_back;
    vec[2].push_back(31); // 2번째 배열의 벡터에 31을 push_back;
}
```



Graph(그래프)

그래프의 탐색

그래프의 모든 정점을 방문하는 것으로
DFS, BFS등이 존재한다

Depth-First Search (깊이 우선 탐색)

‘DFS’ 에 대해 알아보자!

DFS_개념

- **현재 정점에서 갈 수 있는 정점들까지 들어가며 탐색**

  → 한 방향으로 계속 갈 수 있을 때까지 탐색하다 막히면, 가장 가까운 갈림길로 돌아와서 다른 방향으로 탐색을 진행하는 방법

- **모든 정점을 방문하고 싶을 때 사용**

- **BFS와 비교하면 코드는 간단하다**

DFS_특징

1. 한번 방문한 정점은 다시 방문하지 않는다

→ 정점에 방문하였는지 확인하는 visit 배열을 만들어 구현

2. 재귀함수 또는 스택을 이용하여 구현

3. 표현 방식에 따른 시간복잡도

→ 인접 행렬 : O(V^2)

→ 인접 리스트 : O(V+E)

DFS_특징

간선(E)이 적을 경우 인접 리스트를 사용하는 것이 유리하다!

1. 한번 방문한 정점은 다시 방문하지 않는다

2. 재귀함수 또는 스택을 이용하여 구현

3. 표현 방식에 따른 시간복잡도

$V^2 > V+E$

→ 인접 행렬 : O(V^2)

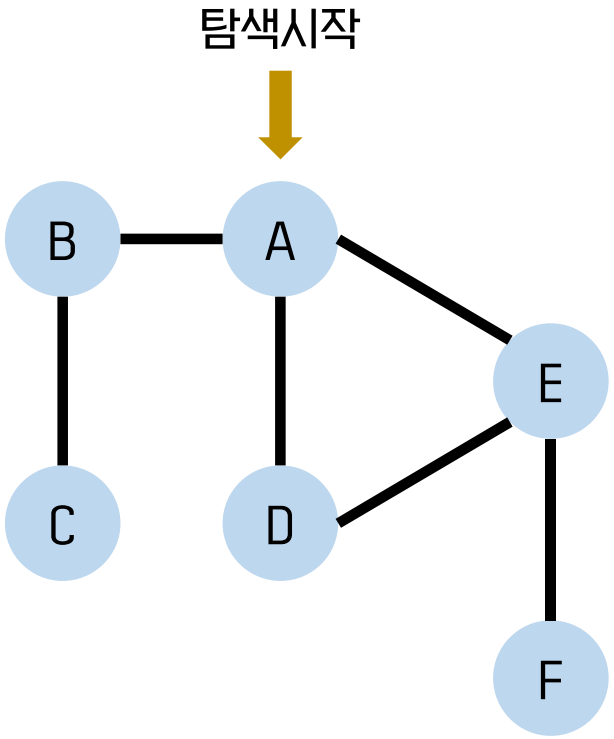
→ 인접 리스트 : O(V+E)

DFS_원리

- A ~ F 의 방문을 확인하는 visit배열 생성

  → 정점의 개수 ≤ 배열크기

- 그래프를 입력 받고, 탐색 시작할 정점 결정



	A	B	C	D	E	F
visit	0	0	0	0	0	0

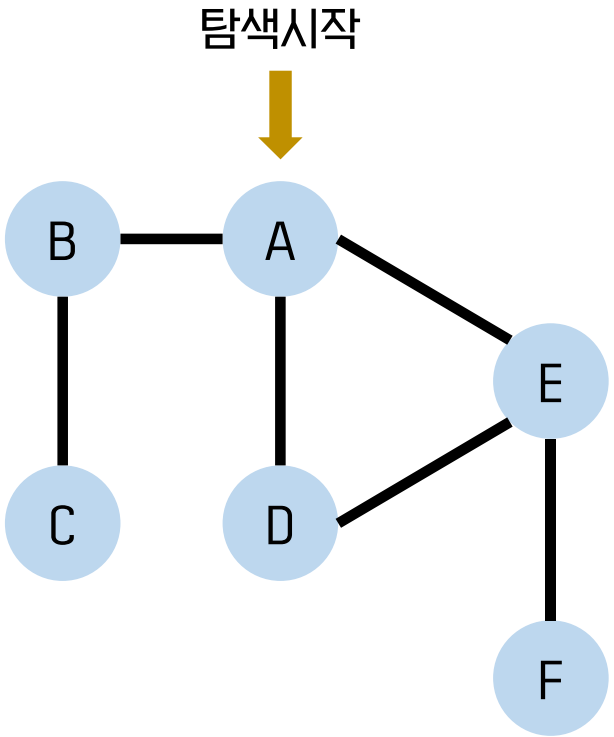
now	

DFS_원리

- **A부터 탐색시작**

  → 다른 정점부터 시작해도 상관없다.

- **그 이후 번호가 작은 정점부터 탐색**



방문 순서

	A	B	C	D	E	F
visit	0	0	0	0	0	0

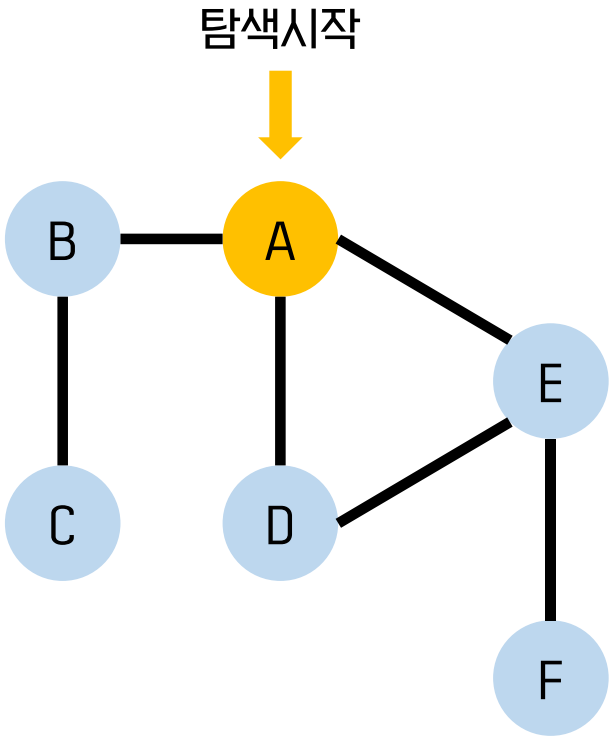
now	

DFS_원리

- **A부터 탐색시작**

  → 다른 정점부터 시작해도 상관없다.

- **그 이후 번호가 작은 정점부터 탐색**



방문 순서

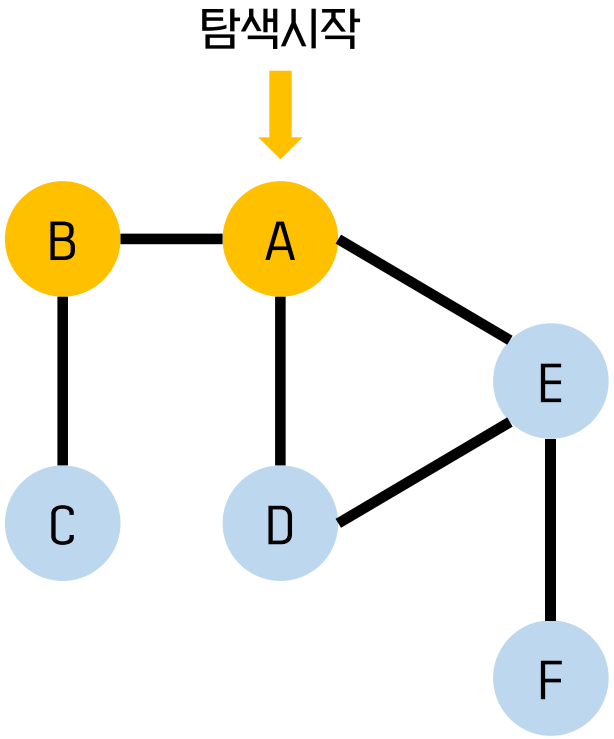
A

	A	B	C	D	E	F
visit	1	0	0	0	0	0

now	**A**

DFS_원리

- 그 이후 번호가 작은 정점부터 탐색



방문 순서

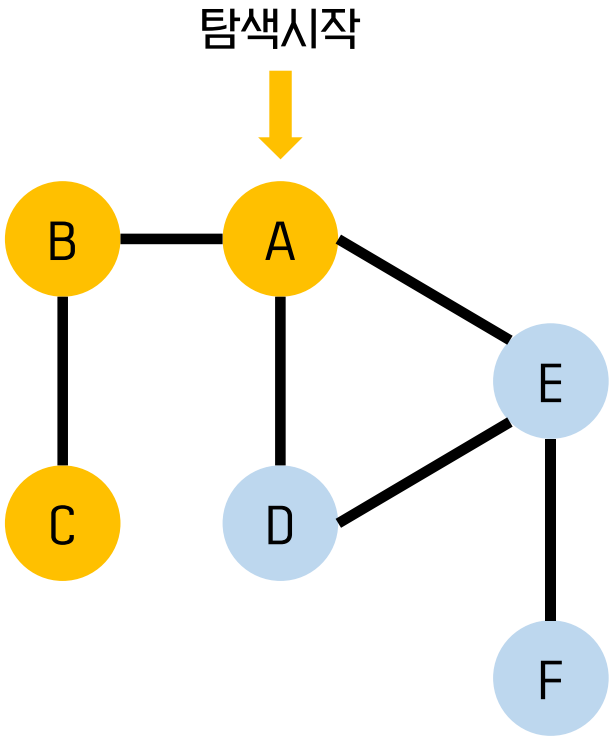
A → B

	A	B	C	D	E	F
visit	1	1	0	0	0	0

now	B

DFS_원리

- 그 이후 번호가 작은 정점부터 탐색



방문 순서

A → B → C

	A	B	C	D	E	F
visit	1	1	1	0	0	0

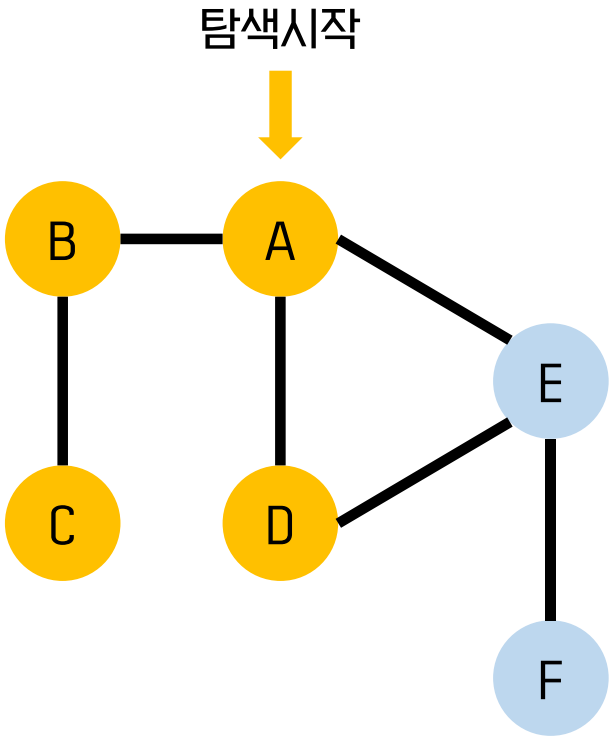
now	C

DFS_원리

- C에서 더 갈수 있는 정점이 없으므로 돌아오기

  → 연결된 노드 중 방문하지 않은 정점이 있나 확인

- 그 이후 번호가 작은 정점부터 탐색



방문 순서

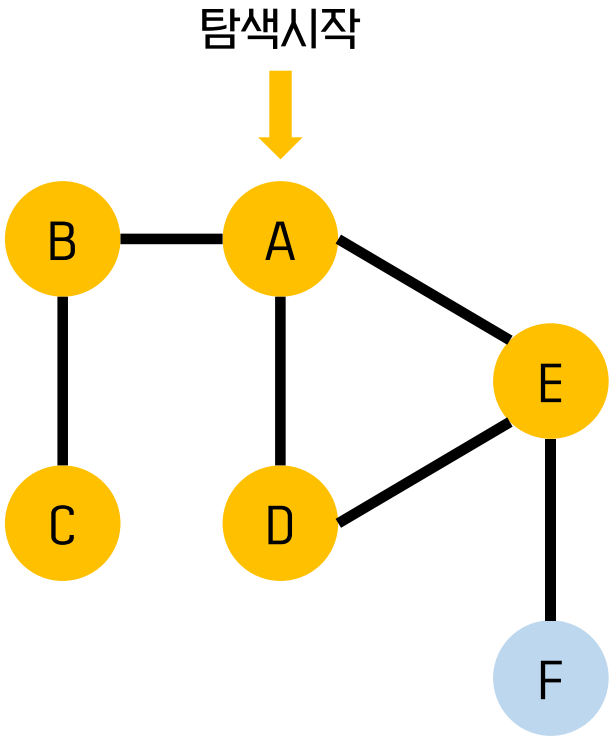
A → B → C → D

	A	B	C	D	E	F
visit	1	1	1	1	0	0

now	D

DFS_원리

- 그 이후 번호가 작은 정점부터 탐색



방문 순서

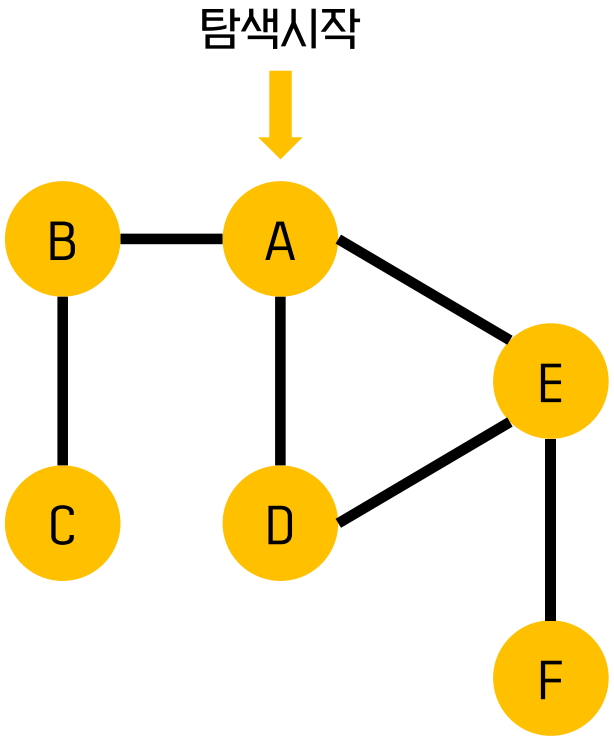
A → B → C → D → E

	A	B	C	D	E	F
visit	1	1	1	1	1	0

now	E

DFS_원리

- 그 이후 번호가 작은 정점부터 탐색
- 방문하지 않은 정점이 없으므로 탐색 종료



방문 순서

A → B → C → D → E → F

	A	B	C	D	E	F
visit	1	1	1	1	1	1

now	F

Breadth-First Search (너비 우선 탐색)

'BFS' 에 대해 알아보자!

DFS_개념

- **현재 정점에서 인접한 연결된 정점부터 탐색**

  → 시작 정점에 가까운 정점부터 넓게 탐색하여 먼 정점은 가장 나중에 방문

- **정점간의 최단경로 또는 임의의 경로를 찾을 때 사용**

- **BFS와 비교하면 코드는 복잡하다.**

DFS_특징

1. 한번 방문한 정점은 다시 방문하지 않는다

→ 정점에 방문하였는지 확인하는 visit 배열을 만들어 구현

2. 큐를 이용하여 구현

3. 표현 방식에 따른 시간복잡도

→ 인접 행렬 : O(V^2)

→ 인접 리스트 : O(V+E)

+ Dijkstra 알고리즘과 유사

DFS_특징

간선(E)이 적을 경우 인접 리스트를 사용하는 것이 유리하다!

1. 한번 방문한 정점은 다시 방문하지 않는다

→ 정점에 방문했는지 안했는지 여부를 1차원 배열을 만들어 구현

2. 큐를 이용하여 구현

3. 표현 방식에 따른 시간복잡도

→ 인접 행렬 : O(V^2)

→ 인접 리스트 : O(V+E)

V^2 > V+E

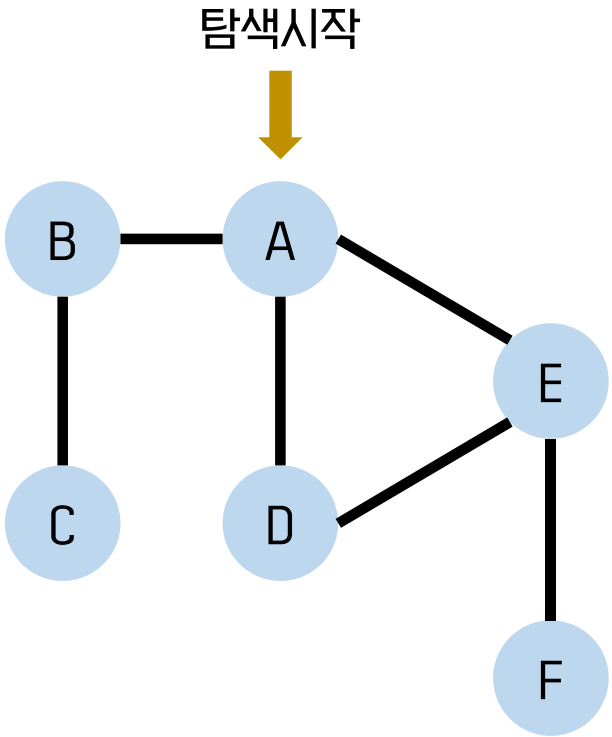
+ Dijkstra 알고리즘과 유사

BFS_원리

- A ~ F 의 방문을 확인하는 visit배열 생성

  → 정점의 개수 ≤ 배열크기

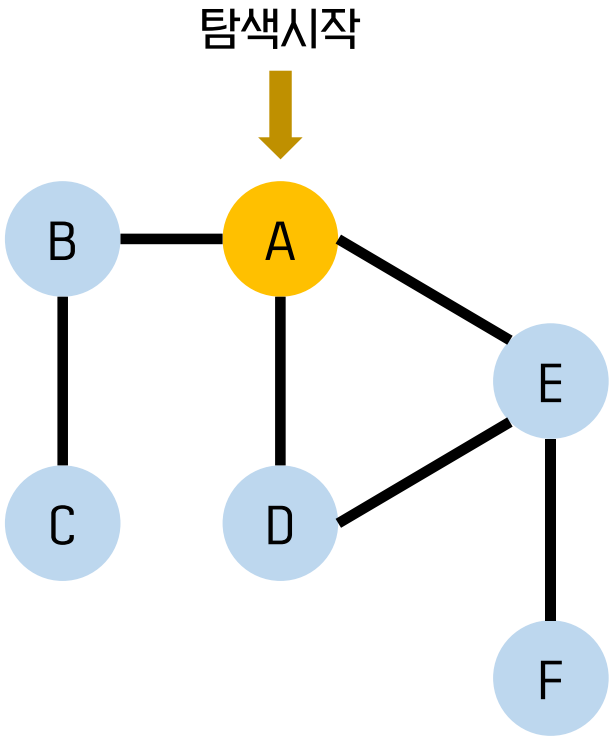
- 그래프를 입력 받고, 탐색 시작할 정점 결정



	A	B	C	D	E	F
visit	0	0	0	0	0	0
Queue						

BFS_원리

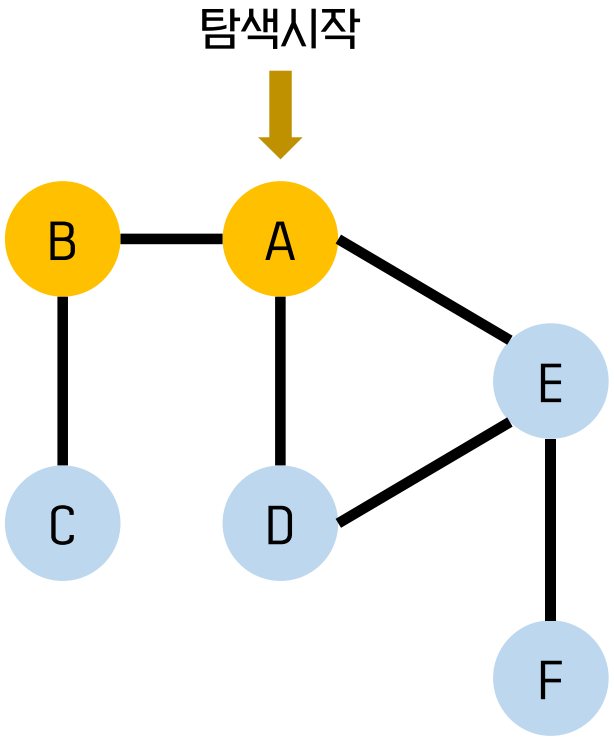
- 큐에 처음 탐색 시작하는 정점 입력



	A	B	C	D	E	F
visit	1	0	0	0	0	0
Queue	A					

BFS_원리

- **A와 인접한 방문하지 않은 정점을 큐에 삽입**
  → 큐의 순서대로 탐색한다
- **방문한 정점은 큐에서 POP**



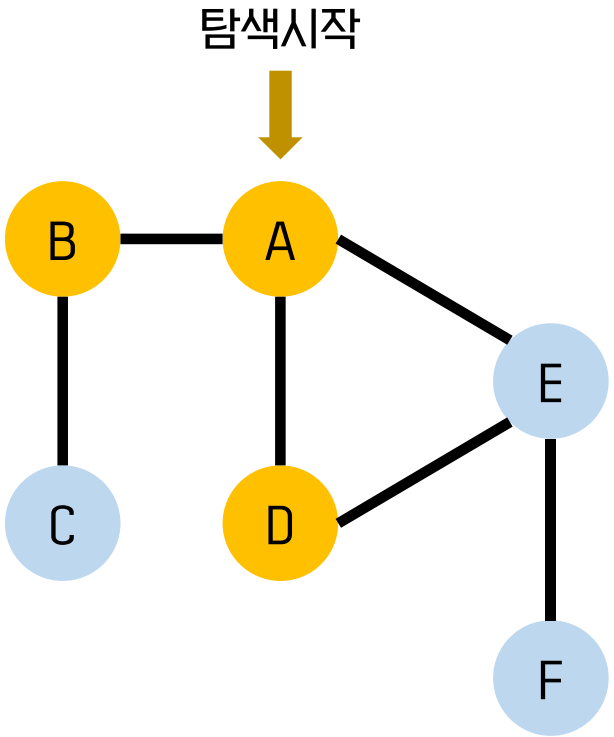
	A	B	C	D	E	F
visit	1	0	0	0	0	0
Queue	B	D	E			

BFS_원리

- **B와 인접한 방문하지 않은 정점을 큐에 삽입**

  → 큐의 순서대로 탐색한다

- **방문한 정점은 큐에서 POP**



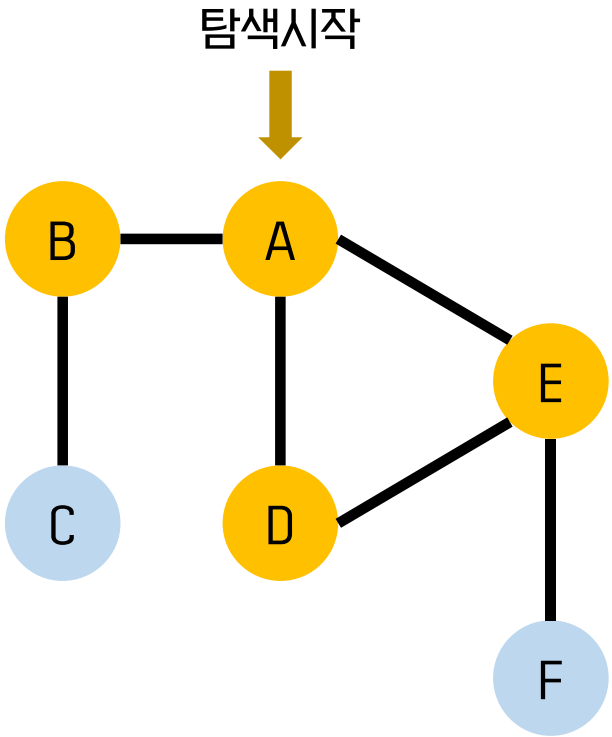
	A	B	C	D	E	F
visit	1	1	0	0	0	0
Queue	D	E	C			

BFS_원리

- D와 인접한 방문하지 않은 정점을 큐에 삽입

  → 큐의 순서대로 탐색한다

- 방문한 정점은 큐에서 POP



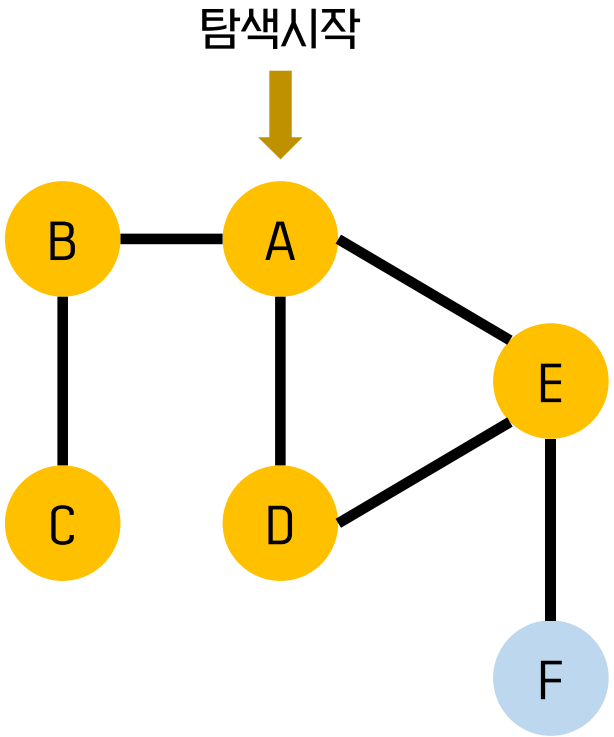
	A	B	C	D	E	F
visit	1	1	0	1	0	0
Queue	E	C	F			

BFS_원리

- E와 인접한 방문하지 않은 정점을 큐에 삽입

  → 큐의 순서대로 탐색한다

- 방문한 정점은 큐에서 POP



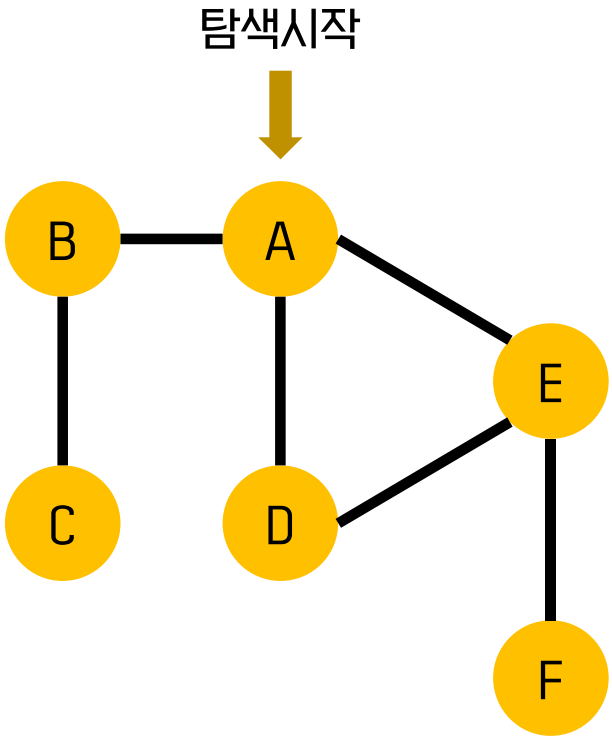
	A	B	C	D	E	F
visit	1	1	0	1	1	0
Queue	C	F				

BFS_원리

- F와 인접한 방문하지 않은 정점을 큐에 삽입

  → 큐의 순서대로 탐색한다

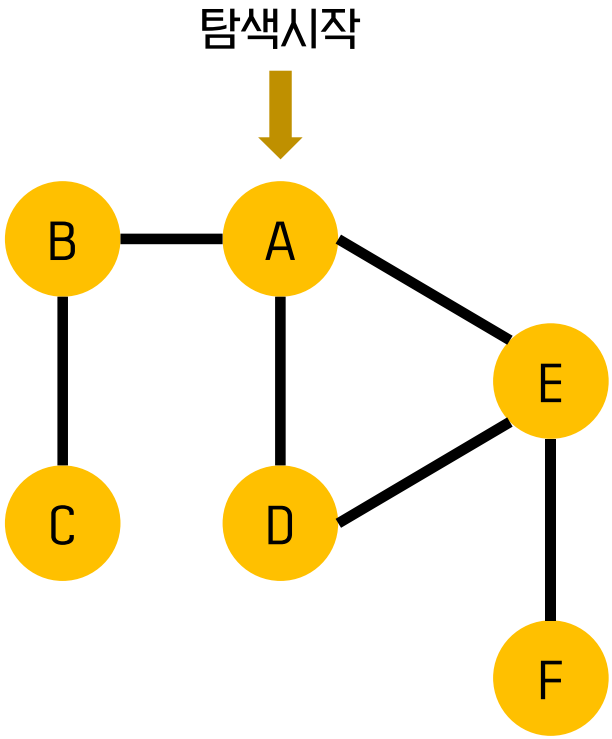
- 방문한 정점은 큐에서 POP



	A	B	C	D	E	F
visit	1	1	1	1	1	0
Queue	F					

BFS_원리

- 모든 정점을 방문했으면 탐색 종료



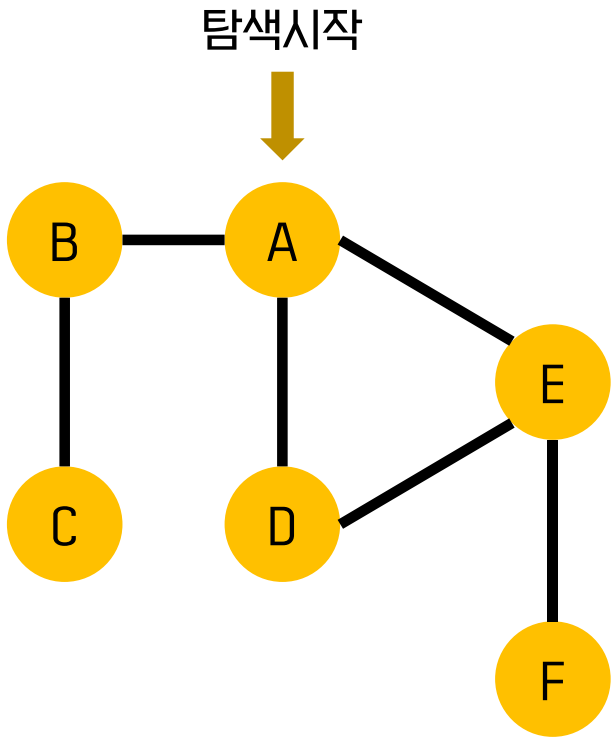
	A	B	C	D	E	F
visit	1	1	1	1	1	1
Queue						

BFS_원리

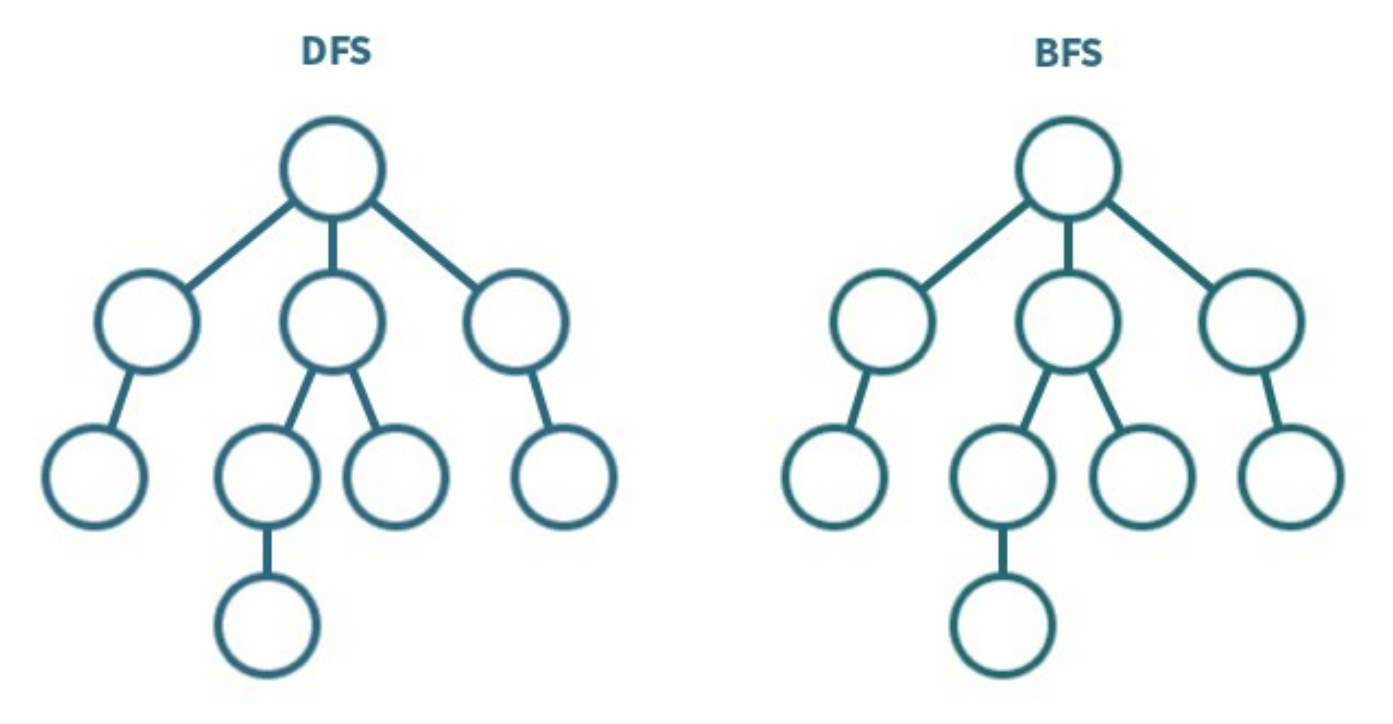
탐색 순서

A → B → D → E → C → F

	A	B	C	D	E	F
visit	1	1	1	1	1	1
Queue						



한눈에 보는 DFS, BFS의 탐색차이



https://twpower.github.io/images/20180114_73/dfs-bfs-example.gif

DFS, BFS 구현

DFS, BFS 하루 만에 뽀개기!

DFS_구현(벡터)

```
void dfs(int x) // main에서 처음 탐색할 정점 입력
{
    visit_dfs[x] = true; // 정점 x 를 방문했으므로 true
    printf("%d ", x); // 방문한 정점 x 를 출력
    for (int i = 0; i < vec[x].size(); i++) // 정점x에 연결된 정점만큼
        if (!visit_dfs[vec[x][i]]) // 방문하지 않은 정점이면
            dfs(vec[x][i]); // dfs 실행 (재귀함수이용)
}
```

BFS_구현(벡터)

```
void bfs()
{
    q.push(v); // v = 처음 시작할 정점
    visit_bfs[v] = true; // 방문했다고 표시
    while (!q.empty()) { // q 에 아무것도 없을때까지 (연결된 정점 X)
        int now = q.front(); // q의 첫번째 요소  = now
        q.pop(); // 방문한 정점은 pop
        printf("%d ", now); // 방문한 정점 출력
        for (int i = 0; i < vec[now].size(); i++) { // 현재 정점에 연결된 정점 개수만큼
            if (!visit_bfs[vec[now][i]]) { //방문하지 않았으면
                q.push(vec[now][i]); // 큐에 그 정점 push
                visit_bfs[vec[now][i]] = true; // 그 정점을 방문했다고 표시
            }
        }

    }
}
```

만약 visit_bfs를 정점을 pop할 때 바꾼다면?

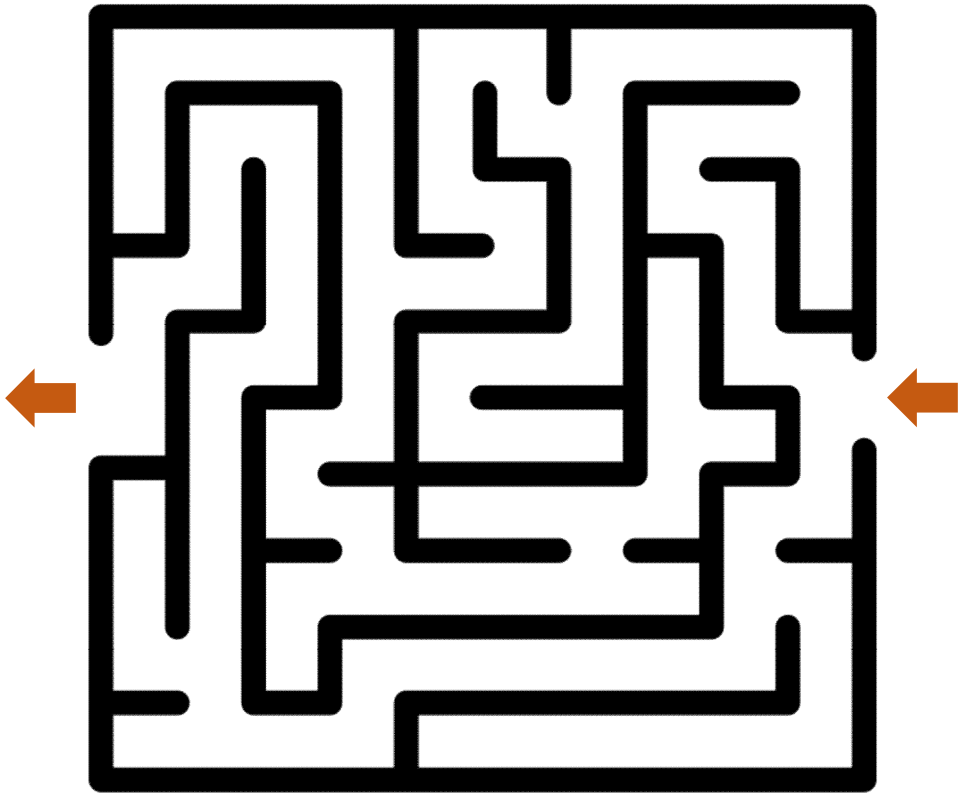
- Queue에 여러 번 입력되어 중복 방문이 될 수 있다!

연습

1260번 - DFS와 BFS
11724번 - 연결 요소의 개수

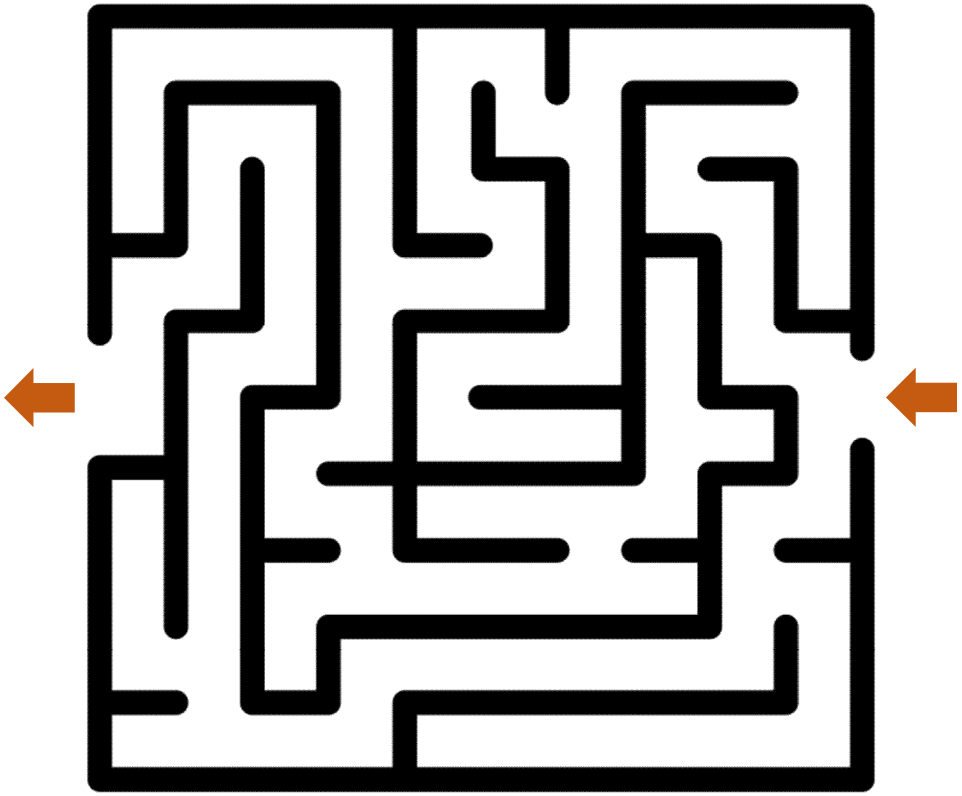
시간순삭 미로찾기

미로 찾기(2178번)



미로 찾기(2178번)_기본

- 정점 대신 (x,y) 좌표
- bfs는 pair<int, int> 사용 하여 queue 이용
- 지도이므로 배열로 받아서 탐색
- visit은 미로의 모든 좌표(2차원)



미로 찾기(2178번)_기본

- **한자리 수로 scanf 하기**

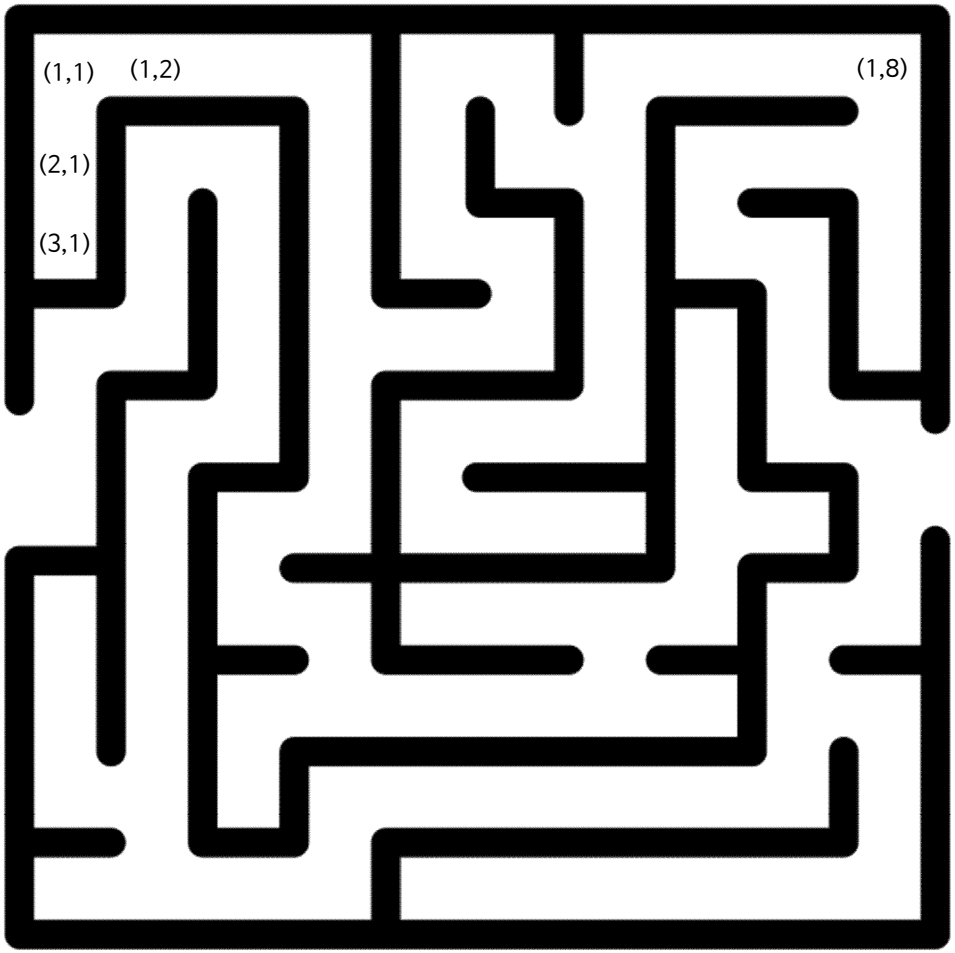
  → scanf("%1d", &MAP[i][j])

- **최단거리를 구하기 위한 식**

  → dis[y][x] 배열로 각 좌표까지
  최단거리 구하기

- **상,하,좌,우를 나타내기**

  → int dx[4] = {-1,1,0, 0}
  int dy[4] = {0,0,-1,1}



미로 찾기_DFS 이용

```
void dfs(int x, int y) { //x,y 좌표
    visit_dfs[y][x] = true; // 이차원 배열에서는 앞에가 y
    for (int i = 0; i < 4; i++) { // 상하좌우 확인
        int nx = x + dx[i]; // x값 변화
        int ny = y + dy[i]; // y값 변화
        if (visit_dfs[ny][nx] == false) { // 방문하지 않았다면 + 지도 안쪽인지 확인도 추가해야함!
            dfs(ny, nx); //dfs 실행
        }
    }
}
```

최단거리를 구하는 것이므로 bfs가 좋다

지도에서 벗어나는지 확인하는 법?

미로 찾기_BFS 이용

```
void bfs() {
    q.push({ 1,1 }); // 1,1 에서부터 미로탐색 시작
    while (!q.empty()) {
        int size = q.size(); // q의 값 = 현재 연결된 정점 개수
        int x = q.front().first; // x값
        int y = q.front().second; // y값
        q.pop();
        visit[x][y] = true; // 방문함 표시
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < 4; j++) { // 상하좌우
                int nx = x + dx[j];
                int ny = y + dy[j];
                if (MAP[ny][nx] == 1 && dis[ny][nx] > dis[y][x] + 1) { // map 안쪽이고, 갈수 있는 길이고, 거리가 더 적으면
                    q.push({ nx,ny });
                    dis[ny][nx] = dis[y][x] + 1; // 걸린 거리 입력
                }
            }
        }
    }
}
```

dis 배열은 INF값으로 초기화

지도에서 벗어나는지 확인하는 법?

미로 찾기_memset

- memset을 사용하여 배열을 -1로 초기화

```
#include<string.h>
main() {
    memset(MAP, -1, sizeof(MAP));
}
```

-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1

미로 찾기_memset

- **지도를 입력 받고 bfs 탐색 실행**
  - → if문에 -1, 0 이 아닌 조건을 추가
  - → MAP은 (1,1) 부터 입력받기

-1	-1	-1	-1	-1	-1	-1	-1
-1	1	0	0	0	0	0	-1
-1	1	0	0	1	1	1	-1
-1	1	0	0	1	0	1	-1
-1	1	0	0	1	0	1	-1
-1	1	1	1	1	0	1	-1
-1	1	0	1	1	0	1	-1
-1	-1	-1	-1	-1	-1	-1	-1

미로 찾기_지도의 범위

- 지도의 크기가 나와있다면

→ if문에 지도의 크기보다 x,y 값이 작다는 조건 추가

연습

2178번 - 미로탐색(최단거리)
1012번 - 유기농 배추
7576 - 토마토

연습+++

2644번 - 촌수계산
2583번 - 영역 구하기
7562번 - 나이트의 이동

다 보느라 수고하셨어요

감사합니다