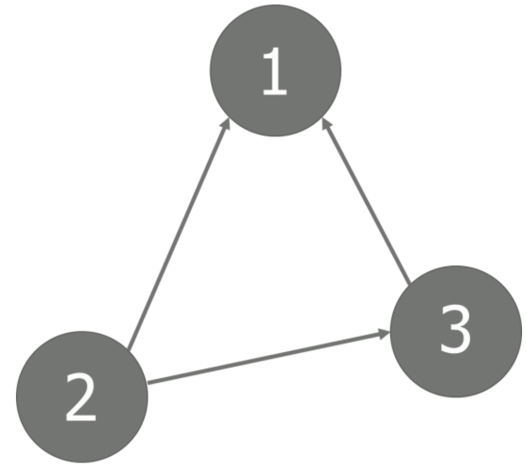
그래프 (Graph)

- 자료 구조의 일종
- 정점 (Node, Vertex), 간선(Edge) : 정점 간의 관계
- G = (V, E)



▣ 경로

정점 A → B로 가는 경로

A → C →D →E →B

A → B

A → C → B

A → C → E → B

▣ 사이클

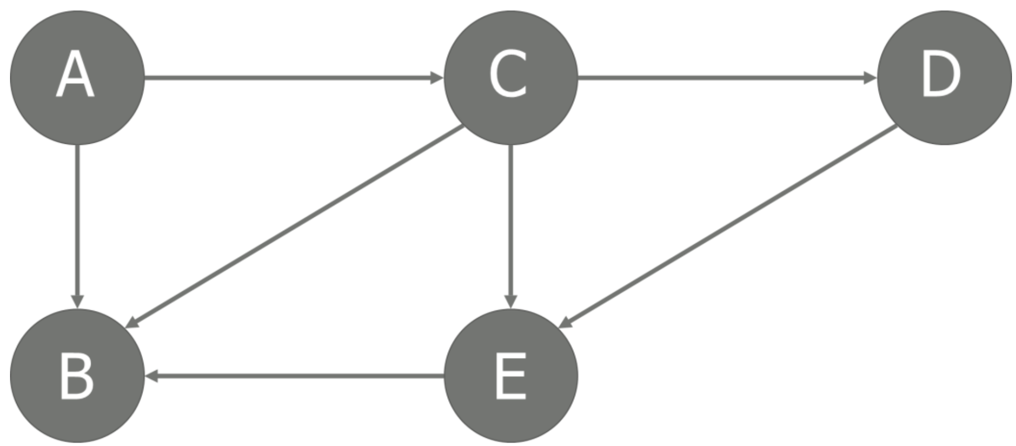
정점 A → A로 가는 경로

A → C → D → E → B

A → B

A → C → B

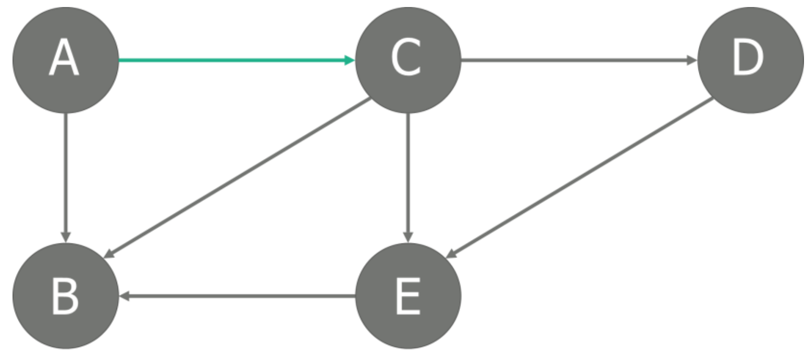
A → C → E → B



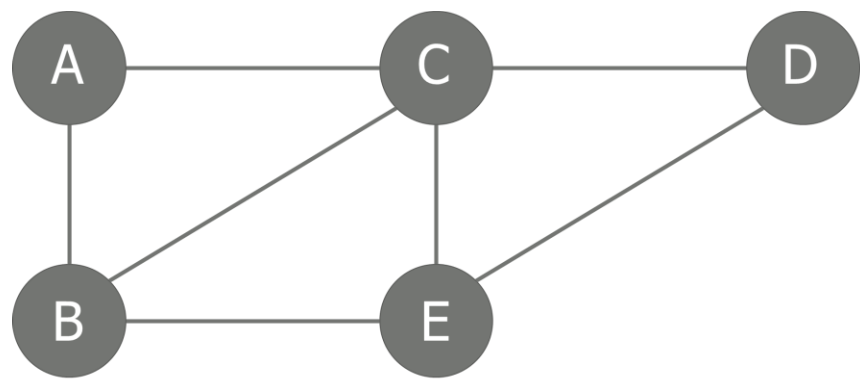
그래프 (Graph)

▣ 단순 경로와 단순 사이클
- 경로/사이클에서 같은 정점을 두 번 이상 방문하지 않는 경로/사이클
- 특별한 말이 없으면, 일반적으로 사용하는 경로와 사이클은 단순 경로/사이클을 말한다.

▣ 유향 그래프 (Directed Graph)
- A → C와 같이 간선에 방향이 있다.
- A → C는 있지만, C → A는 없다.



▣ 무향 그래프 (Undirected Graph)
- A – C와 같이 간선에 방향이 없다.
- A – C는 A → C와 C → A를 나타낸다.

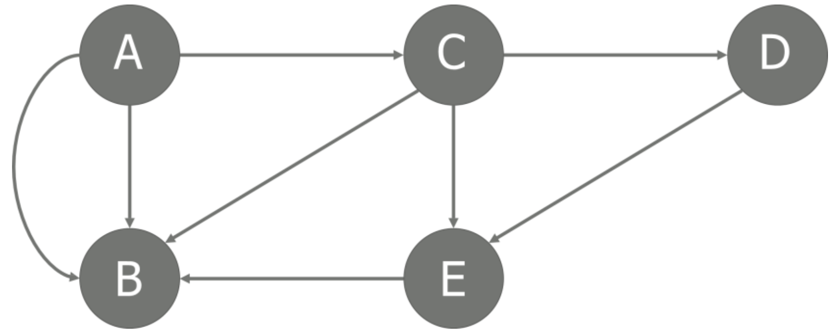


Gaussian

그래프 (Graph)

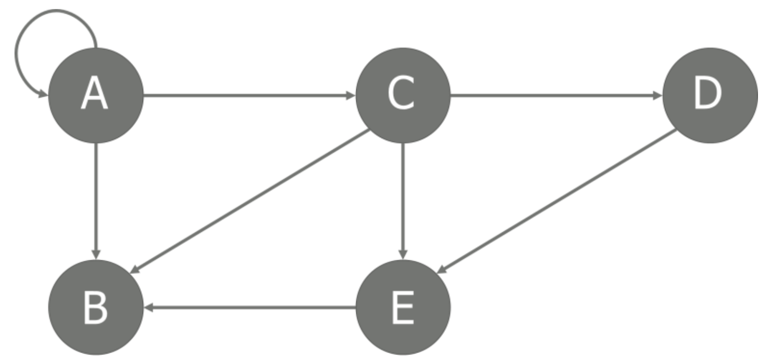
▣ 간선 여러 개 (Multiple Edge)

- 두 정점 사이에 간선이 여러 개일 수도 있다.
- A – B는 연결하는 간선이 2개이다.
- 두 간선은 서로 다른 간선이다.



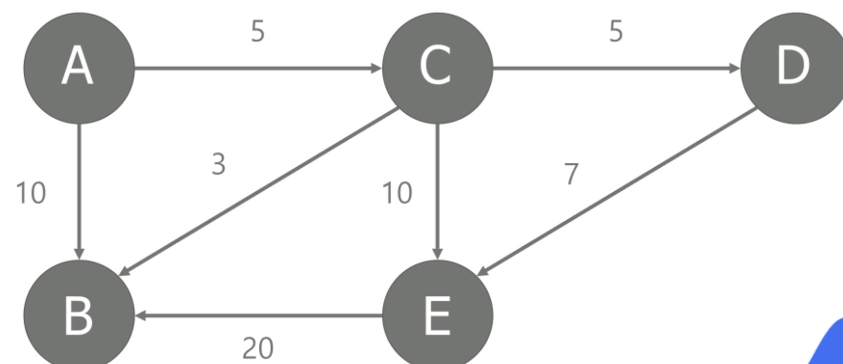
▣ 루프 (Loop)

- 간선의 양 끝점이 같은 경우
- A → A



▣ 가중치

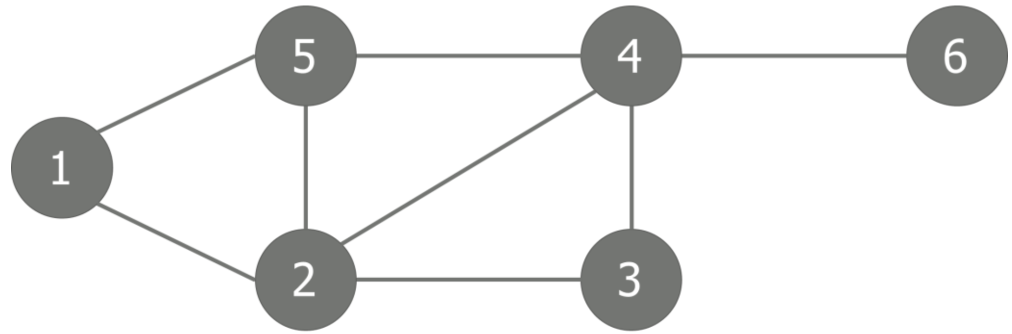
- 간선에 가중치가 있는 경우
- A → B로 이동하는 거리, 시간, 비용 등 표현
- 가중치가 없으면 가중치 1로 생각



그래프 (Graph)

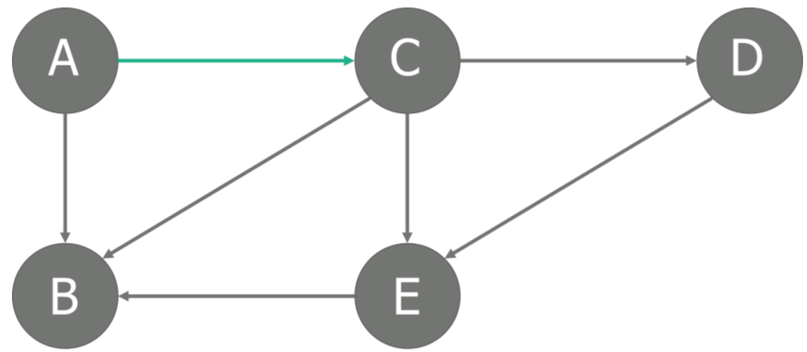
▣ 차수 (Degree)

- 정점과 연결되어 있는 간선의 개수
- 5의 차수 : 3
- 4의 차수 : 4



유향 그래프에서는 indgree, outdgree로 나뉨

- E의 차수 : indegree : 2
  outdegree : 1

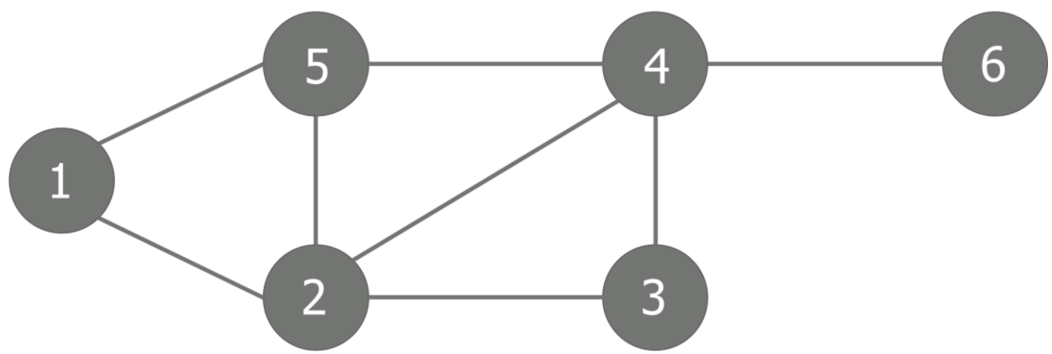


그래프 (Graph)

▣ 그래프의 표현

- 정점이 V = 6개, 간선이 E = 8개
- 간선에 방향이 없기 때문에, 방향이 없는 그래프이다.
- 정점 : {1, 2, 3, 4, 5, 6}
- 간선 : {(1, 2), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (4, 5), (4, 6)}
- 문제를 풀 때에는 아래와 같이 입력 받는다.

1	2
1	5
2	3
2	4
2	5
3	4
4	5
4	6

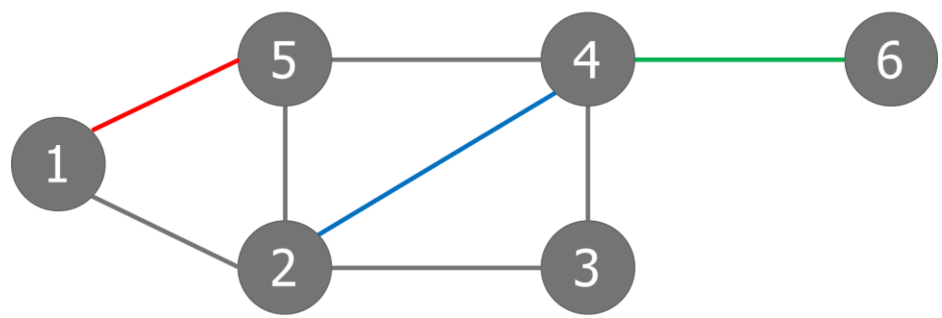


그래프 (Graph)

▣ 인접 행렬

- 정점의 개수를 V라고 했을 때 V X V 크기의 이차원 배열을 이용한다.
- A[i][j] = 1 (i → j 간선이 있을 때), 0 (간선이 없을 때)
- 아래와 같은 양방향 그래프 일 때 인접 행렬은 대칭 값을 가진다.
- 없는 간선도 저장해야 하므로 공간이 많이 필요 하다. O(V^2) ($V^2 \geq E$)
- 없는 간선도 탐색해야 하므로 시간도 오래 걸려 잘 사용하지 않는다.

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



그래프 (Graph)

▣ 인접 행렬

- 가중치가 없는 그래프 : 간선이 있는 경우 1, 간선이 없는 경우 0을 저장한다.
- 가중치가 있는 그래프 :
  ① 가중치 >= 0 인 경우 간선이 있으면 가중치 w, 없으면 -1 저장
  ② 가중치의 범위가 정수 전체 : 간선의 유무 행렬과 가중치 행렬을 따로 만든다.
  간선의 유무 행렬 : 간선이 있으면 1, 없으면 0
  가중치 행렬 : 가중치를 입력하여 간선이 있는 경우 값을 참조

```
int N, M;
scanf("%d %d", &N, &M);
vector<vector<int>> A(N, vector<int>(M));
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
        int u, v;
        scanf("%d %d", &u, &v);
        // Undirected
        A[u][v] = A[v][i] = 1;
        // Directed
        A[u][v] = 1;
    }
}
```

[가중치가 없는 경우]

```
int N, M;
scanf("%d %d", &N, &M);
vector<vector<int>> A(N, vector<int>(M));
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
        int u, v, w;
        scanf("%d %d", &u, &v, &w);
        // Undirected
        A[u][v] = A[v][i] = 1;
        // Directed
        A[u][v] = 1;
    }
}
```

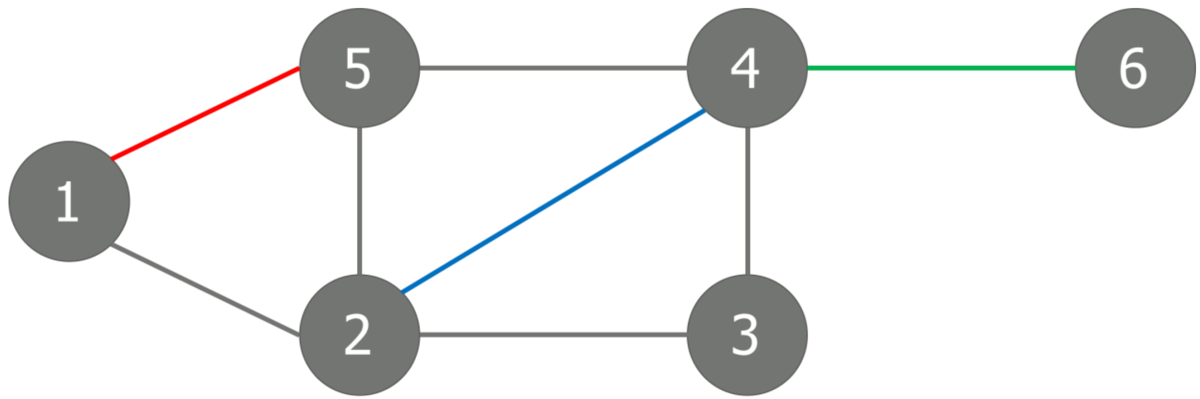
[가중치가 있는 경우]

그래프 (Graph)

▣ 인접 리스트

- 링크드 리스트를 이용해서 구현한다.
- A[i] = i와 연결된 정점을 링크드 리스트로 포함하고 있다.
- 효율적으로 공간을 사용하기 위해서 필요한 간선만 저장한다. O(E)
- 인접 리스트를 생성하는 데 시간이 걸리고, 인접 행렬보다 복잡하지만 문제를 풀 때는 공간의 효율성을 위해 인접 리스트를 우선적으로 사용해야 한다.

A[1]	2, 5
A[2]	1, 3, 4, 5
A[3]	2, 4
A[4]	3, 5, 2, 6
A[5]	1, 2, 4
A[6]	4



그래프 (Graph)

```
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
        int u, v, w;
        scanf("%d %d", &u, &v, &w);

        //Undirected
        A[u].push_back(v);
        A[v].push_back(u);

        //Directed
        A[u].push_back(v);
    }
}
```

[가중치가 없는 경우]

```
typedef pair<int, int> ii;
vector<ii> a[10];

int main() {

    int N, M;
    scanf("%d %d", &N, &M);
    for (int i = 0; i < M; ++i) {
        int u, v, w;
        scanf("%d %d %d", &u, &v, &w);
        a[u].push_back(ii(v, w));
        a[v].push_back(ii(u, w));
    }
```

정점과 가중치 입력 시 pair 사용
→ pair는 typedef로 재정의 사용하면 편함

[가중치가 있는 경우]

▣ 인접 행렬 및 인접 리스트 공간 복잡도

- 인접 행렬 : O(V^2)
- 인접 리스트 : O(E)

그래프 (Graph)

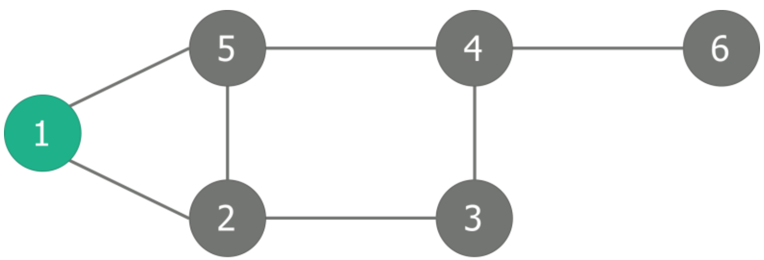
▣ 그래프의 탐색 : 모든 정점을 1번씩 방문하는 것이 목적

- DFS : 깊이 우선 탐색 → 최대한 깊숙이 탐색 → stack 자료구조 사용 (재귀 사용)
- BFS : 너비 우선 탐색 → 최대한 넓게 탐색 → Queue 자료구조 사용

  모든 가중치가 1인 경우 BFS는 최단 경로 탐색 알고리즘으로 사용할 수 있다.

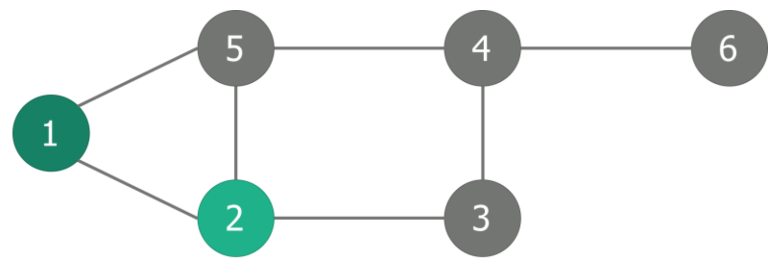
① 깊이 우선 탐색

탐색 : 1 / 스택 : 1



i	1	2	3	4	5	6
check[i]	1	0	0	0	0	0

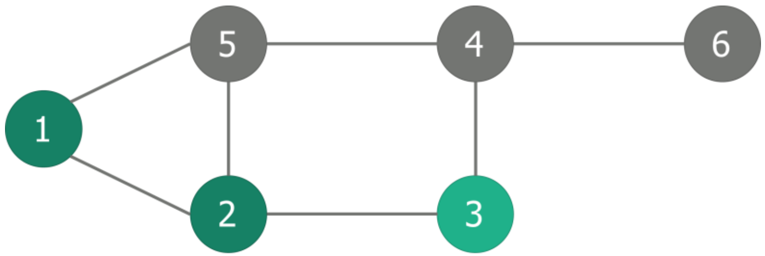
탐색 : 1, 2 / 스택 : 1, 2



i	1	2	3	4	5	6
check[i]	1	1	0	0	0	0

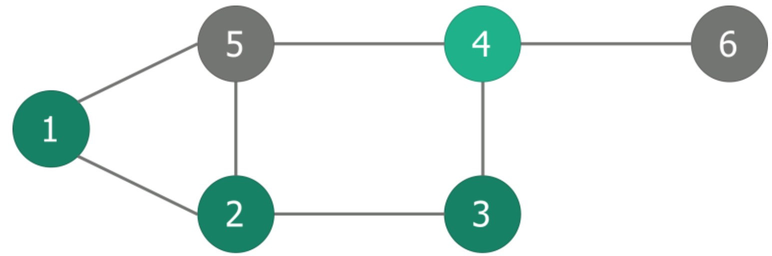
그래프 (Graph)

탐색 : 1, 2, 3 / 스택 : 1, 2, 3



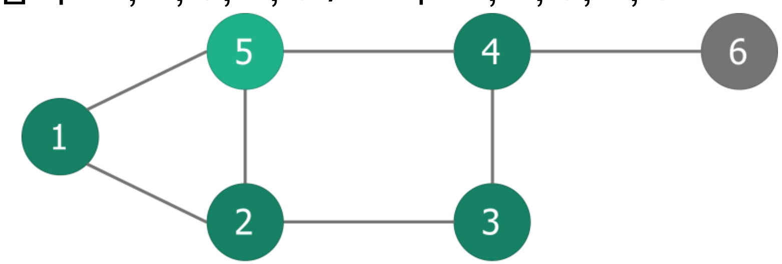
i	1	2	3	4	5	6
check[i]	1	1	1	0	0	0

탐색 : 1, 2, 3, 4 / 스택 : 1, 2, 3, 4



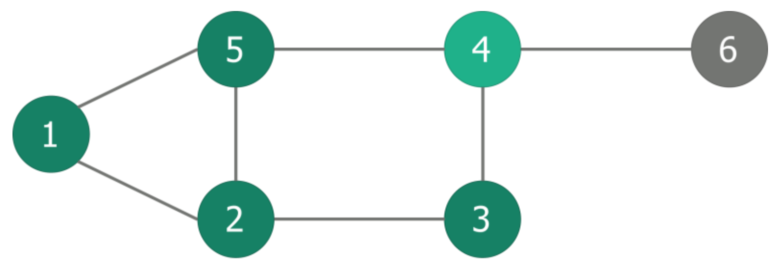
i	1	2	3	4	5	6
check[i]	1	1	1	1	0	0

탐색 : 1, 2, 3, 4, 5 / 스택 : 1, 2, 3, 4, 5



i	1	2	3	4	5	6
check[i]	1	1	1	1	1	0

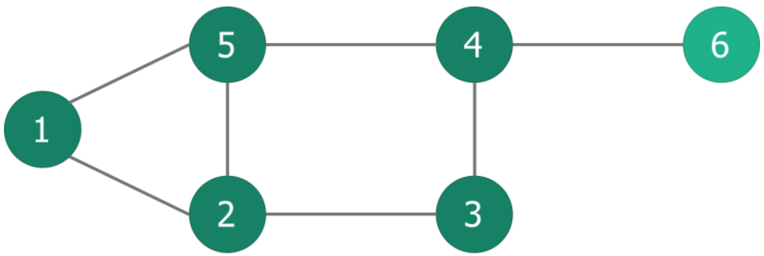
탐색 : 1, 2, 3, 4, 5 / 스택 : 1, 2, 3, 4



i	1	2	3	4	5	6
check[i]	1	1	1	1	1	0

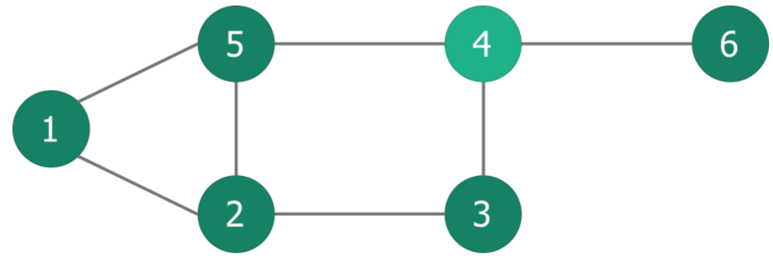
그래프 (Graph)

탐색 : 1, 2, 3, 4, 5, 6 / 스택 : 1, 2, 3, 4, 6



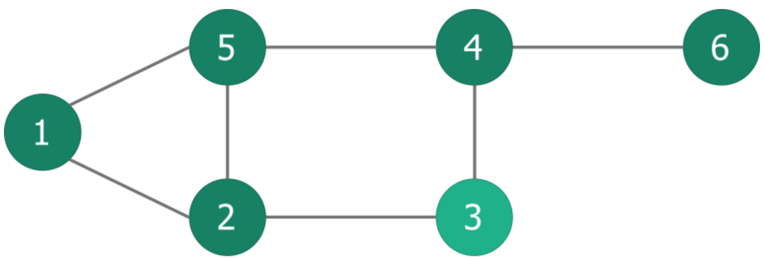
i	1	2	3	4	5	6
check[i]	1	1	1	1	1	1

탐색 : 1, 2, 3, 4, 5, 6 / 스택 : 1, 2, 3, 4



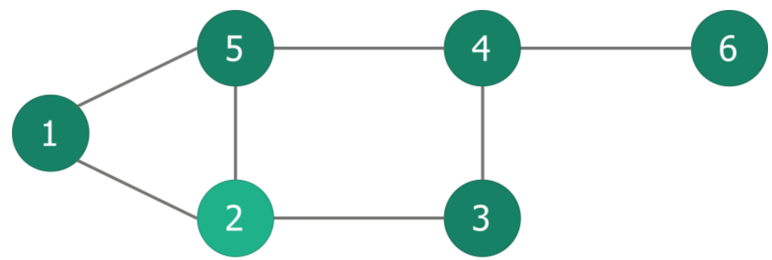
i	1	2	3	4	5	6
check[i]	1	1	1	1	1	1

탐색 : 1, 2, 3, 4, 5, 6 / 스택 : 1, 2, 3



i	1	2	3	4	5	6
check[i]	1	1	1	1	1	1

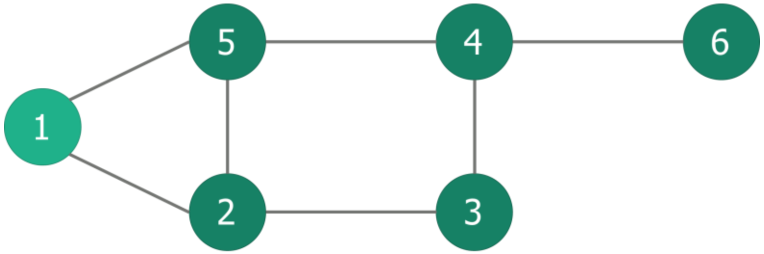
탐색 : 1, 2, 3, 4, 5, 6 / 스택 : 1, 2



i	1	2	3	4	5	6
check[i]	1	1	1	1	1	1

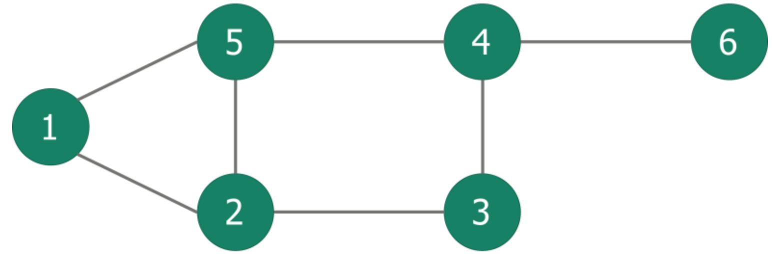
그래프 (Graph)

탐색 : 1, 2, 3, 4, 5, 6 / 스택 : 1



i	1	2	3	4	5	6
check[i]	1	1	1	1	1	1

탐색 : 1, 2, 3, 4, 5, 6 / 스택 : empty



i	1	2	3	4	5	6
check[i]	1	1	1	1	1	1

- 재귀 호출을 이용하여 구현할 수 있다.
- 인접 행렬을 이용한 경우 각 정점에서 모든 정점을 탐색하게 되므로 V x O(V) = O(V^2)

```
int N;
vi visited;
vii AdjMat;
void dfs(int x){
    visited[x] = true;
    printf("%d ", x);

    for (int i = 1; i <= N; ++i) {
        if (AdjMat[x][i] == 1 && !visited[i]) {
            dfs(i);
        }
    }
}
```

다음 정점 확인

그래프 (Graph)

- 인접 리스트를 이용한 경우 모든 정점과 모든 간선을 탐색하게 되므로 O(V+E)가 된다.
- 인접 행렬에 비하여 공간 복잡도와 탐색 시간 복잡도를 비교해 보면 인접리스트가 효율적
  → 인접 리스트를 이용한 탐색을 이용하도록 한다.

```
int N;
vi visited;
vi AdjList[10];
void dfs(int x){
    visited[x] = true;
    printf("%d ", x);
                              존재하는 간선만 체크
    for (int i = 0; i <= AdjList[x].size(); ++i) {
        int y = AdjList[x][i];
        if(!visited[y]) {
            dfs(y);   다음 정점 확인
        }
    }
}
```

- 인접 행렬 vs 인접 리스트

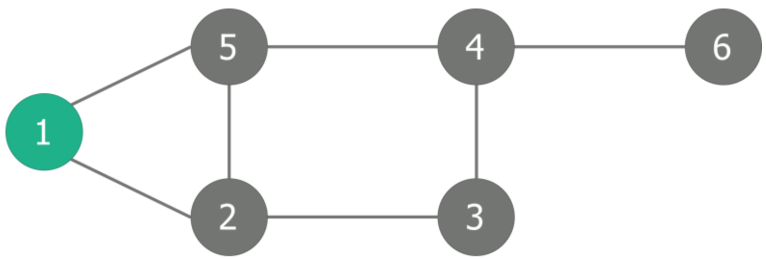
	인접 행렬	인접 리스트
공간 복잡도	O(V^2)	O(E)
시간 복잡도	O(V^2)	O(V+E)

그래프 (Graph)

② 너비 우선 탐색 (BFS)

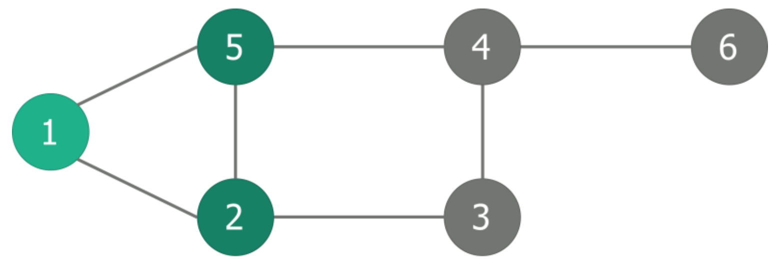
- Queue를 이용해서 지금 위치에서 갈 수 있는 것을 모두 큐에 넣는 방식
- 큐에 넣었을 때, 방문함을 체크 해야 한다.

탐색 : 1 / 큐 : 1



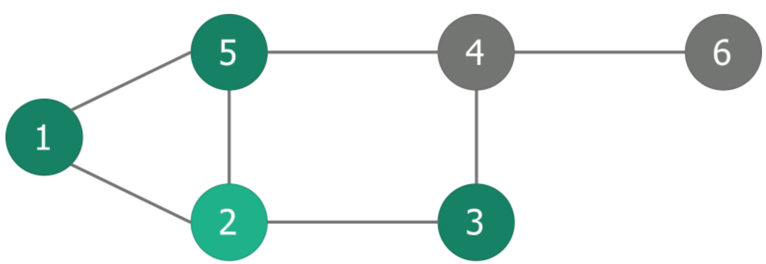
i	1	2	3	4	5	6
check[i]	1	0	0	0	0	0

탐색 : 1,2,5 / 큐 : 2, 5



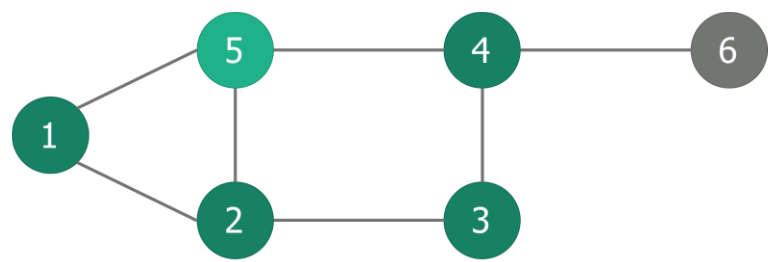
i	1	2	3	4	5	6
check[i]	1	1	0	0	1	0

탐색 : 1,2,5,3 / 큐 : 5, 3



i	1	2	3	4	5	6
check[i]	1	1	1	0	1	0

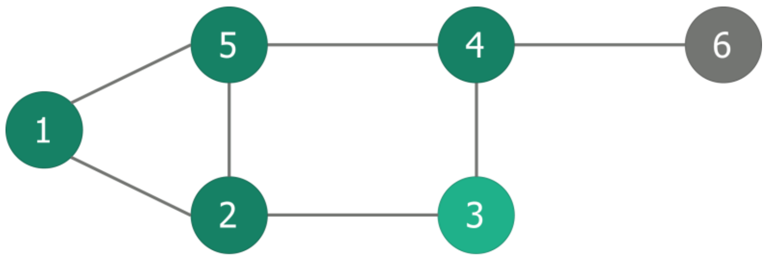
탐색 : 1,2,5,3,4 / 큐 : 3, 4



i	1	2	3	4	5	6
check[i]	1	1	1	1	1	0

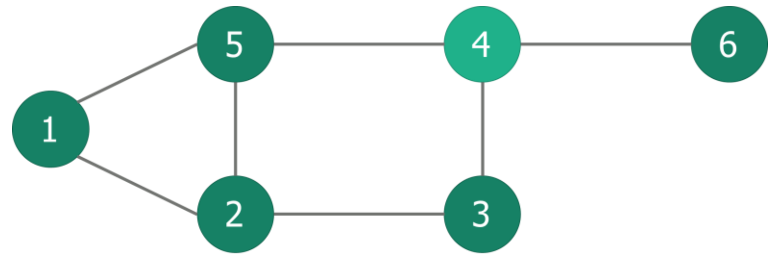
그래프 (Graph)

탐색 : 1,2,3,4,5 / 큐 : 4



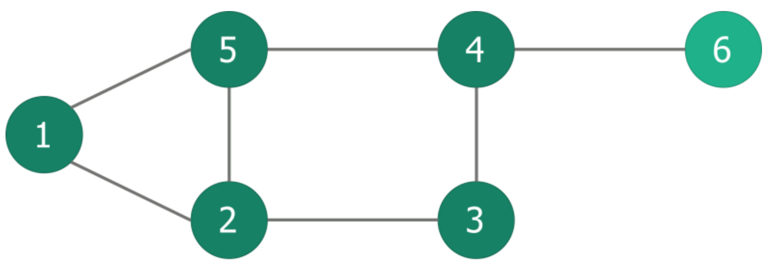
i	1	2	3	4	5	6
check[i]	1	1	1	1	1	0

탐색 : 1,2,3,4,5,6 / 큐 : 6



i	1	2	3	4	5	6
check[i]	1	1	1	1	1	0

탐색 : 1,2,3,4,5,6 / 큐 : empty



i	1	2	3	4	5	6
check[i]	1	1	1	1	1	1

그래프 (Graph)

- Queue를 이용하여 구현할 수 있다.
- 인접 행렬을 이용한 경우 각 정점에서 모든 정점을 탐색하게 되므로 V x O(V) = O(V^2)

```
queue<int> q;
visited[1] = true; //방문하기 전 체크
vii AdjMat;
q.push(1);
while (!q.empty()) {
    int x = q.front(); // 탐색 노드 확인
    q.pop(); //확인 후 pop
    printf("%d ", x);
    // 다음 노드 확인
    for (int i = 1; i <= N; ++i) {
        if (AdjMat[x][i] == 1 && !visited[i]) {
            visited[i] = true;
            q.push(i);
        }
    }
}
```

Gaussian

그래프 (Graph)

- Queue를 이용하여 구현할 수 있다.
- 인접 리스트를 이용한 경우 모든 정점과 간선을 탐색하므로 O(V+E)가 된다.

```
queue<int> q;
visited[1] = true; //방문하기 전 체크
vi AdjList[10];
q.push(1);
while (!q.empty()) {
    int x = q.front(); // 탐색 노드 확인
    q.pop(); //확인 후 pop
    printf("%d ", x);
    // 다음 노드 확인
    for (int i = 0; i < AdjList[i].size(); ++i) {
        int y = AdjList[x][i];
        if (!visited[y]) {
            visited[y] = true;
            q.push(y);
        }
    }
}
```

- 인접 행렬 vs 인접 리스트

	인접 행렬	인접 리스트
공간 복잡도	O(V^2)	O(E)
시간 복잡도	O(V^2)	O(V+E)

그래프 (Graph)

- DFS, BFS 모두 기본적인 그래프 탐색으로 사용할 수 있다.
- DFS, BFS 모두 인접 리스트를 사용하여 구현하는 것이 공간/시간 복잡도에 유리하다.

	인접 행렬	인접 리스트
공간 복잡도	O(V^2)	O(E)
시간 복잡도	O(V^2)	O(V+E)

- DFS Vs. BFS

	O(V+E) DFS	O(V+E) BFS
장점	- 메모리를 덜 사용 - 절단점 다리, SCC를 구할 수 있다.	- 가중치 없는 그래프에서 단일 시작점 최단 경로를 구할 수 있다.
단점	- 가중치 없는 그래프에서 단일 시작점 최단 경로를 구할 수 없다.	- 보통 메모리를 더 쓴다. (그래프가 큰 경우 불리)

Gaussian