서울시립대학교 소모임 알

컴퓨터과학부 김민규, 김희중

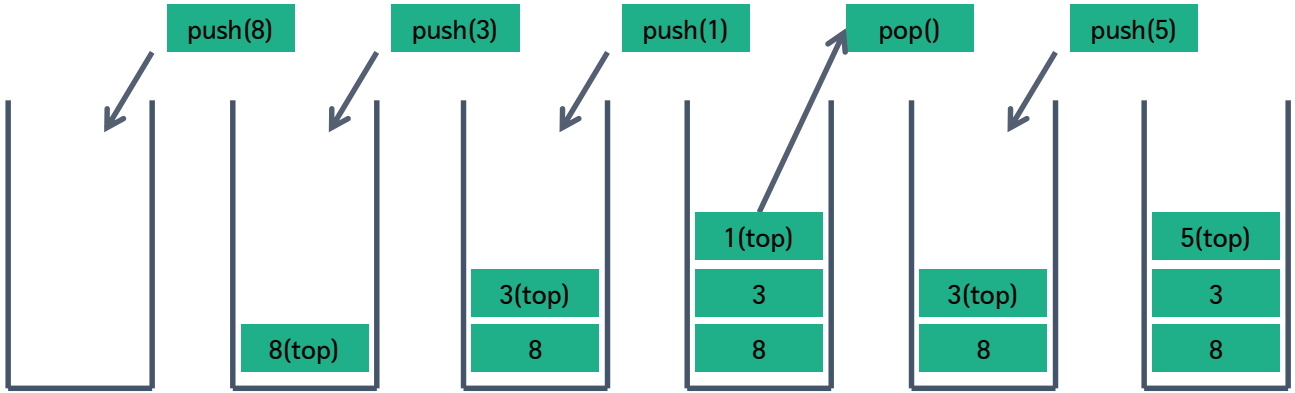
스택, 큐, DFS, BFS

스택

스택

Stack : 쌓다! 종이를 쌓는 것을 떠올리세요.

- 한쪽 끝에서만 자료를 넣고 뺄 수 있는 자료구조



스택

Stack의 특징

- 한쪽 끝에서만 자료를 넣고 뺄 수 있는 자료구조
- 마지막으로 넣은 것이 가장 먼저 나오기 때문에 Last In First Out(LIFO) 라고 한다.
- 재귀 함수, 백 트래킹, DFS(Depth First Search)등과 큰 관련이 있다.

스택

Stack의 연산

- Push : 스택에 자료를 넣는 연산
- Pop : 스택에 자료를 빼는 연산
- Top : 스택의 가장 위에 있는 자료를 보는 연산
- Empty : 스택이 비어있는지 아닌지를 알아보는 연산
- Size : 스택에 저장되어 있는 자료의 개수를 알아보는 연산

스택

문제를 통한 스택의 기본 연산 연습

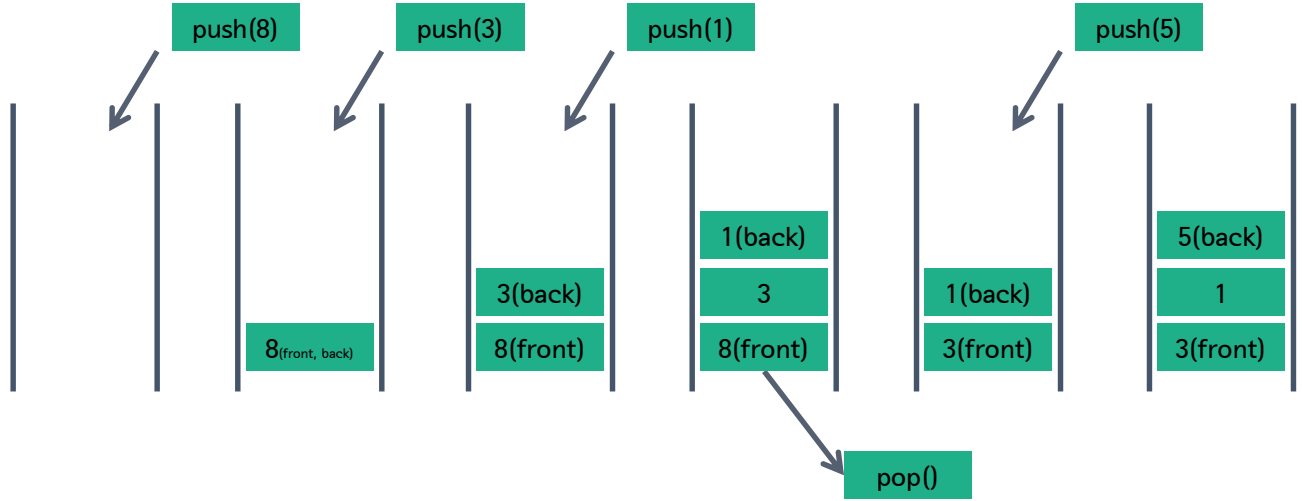
- C++에는 이미 구현되어있는 STL의 스택을 이용할 수 있다.
- 스택의 기본 연산 연습문제 : https://www.acmicpc.net/problem/10828
- C++ Code : https://github.com/OfficialDominyellow/AlgorithmByDominyellow/blob/master/BackjoonOnlineJudge/10828.cc

큐

큐

Queue : 표를 사러 일렬로 늘어선 사람들의 줄!

- 한쪽 끝에서만 자료를 넣고 다른 한쪽 끝에서만 뺄 수 있는 자료구조



큐

Queue의 특징

- 한쪽 끝에서만 자료를 넣고 다른 한쪽 끝에서만 뺄 수 있는 자료구조
- 먼저 넣은 것이 가장 먼저 나오기 때문에 First In First Out(FIFO)라고 한다.
- BFS(Breadth First Search), 최단 거리 알고리즘, OS 스케줄링(작업의 순서를 결정하는 작업)과 관련이 있다.

큐

Queue의 연산

- Push : 큐에 자료를 넣는 연산
- Pop : 큐에서 자료를 빼는 연산
- Front : 큐의 가장 앞에 있는 자료를 보는 연산
- Back : 큐의 가장 뒤에 있는 자료를 보는 연산
- Empty : 큐가 비어있는지 아닌지를 알아보는 연산
- Size : 큐에 저장되어 있는 자료의 개수를 알아보는 연산

큐

문제를 통한 큐의 기본 연산 연습

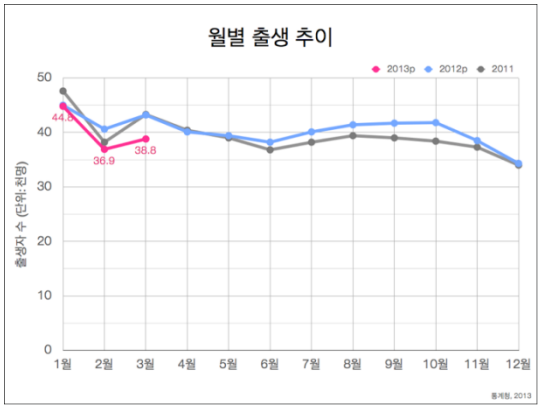
- C++에는 이미 구현되어있는 STL의 큐를 이용할 수 있다.
- 큐의 기본 연산 연습문제 :
  https://www.acmicpc.net/problem/10845
- C++ Code :
  https://github.com/OfficialDominyellow/AlgorithmByDominyellow/blob/master/BackjoonOnlineJudge/10854.cc

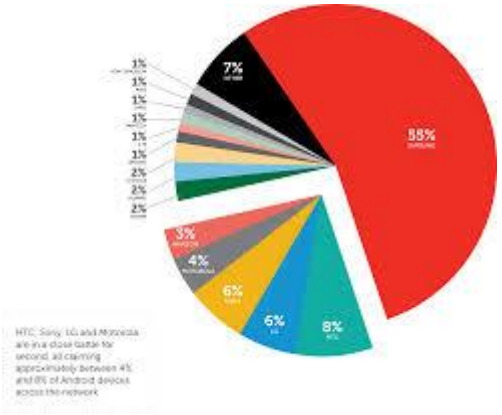
그래프

그래프

컴퓨터 과학부에게 Graph란?

- 컴퓨터 과학부 학생이 Graph라는 단어를 들었을 때 아래와 같은 그림을 떠올렸다면 벽돌로 맞아도 정당방위이다.





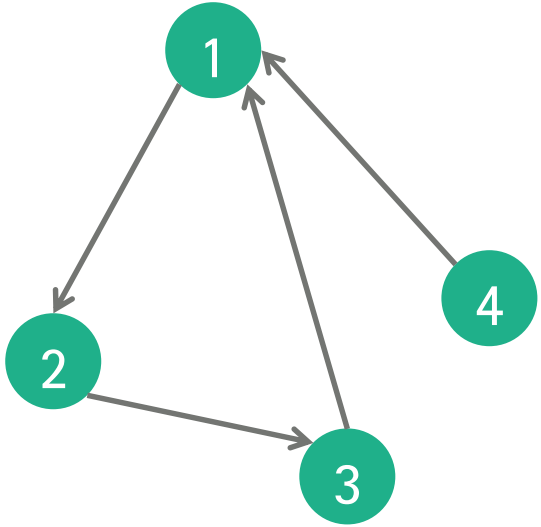
그래프

Graph 자료구조의 일종

- 자료구조의 일종

- 정점(Node, Vertex)
- 간선 (Edge) : 정점간의 관계를 나타냄

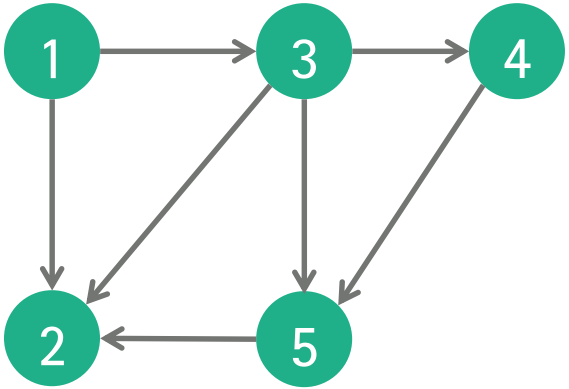
- G = (V, E)로 표현



경로

Path : 정점과 정점 사이의 경로

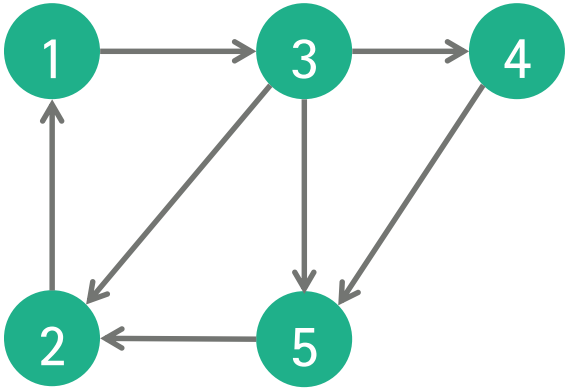
- 정점 A에서 B로 가는 경로
- 1 -> 3 -> 4 -> 5 -> 2
- 1 -> 2
- 1 -> 3 -> 2
- 1 -> 3 -> 5 -> 2



사이클

Cycle : 다시 돌아오는 경로

- 정점 A에서 A로 돌아오는 경로
- 1 -> 3 -> 2 -> 1
- 1 -> 3 -> 5 -> 2 -> 1
- 1 -> 3 -> 4 -> 5 -> 2 -> 1



단순 경로, 단순 사이클

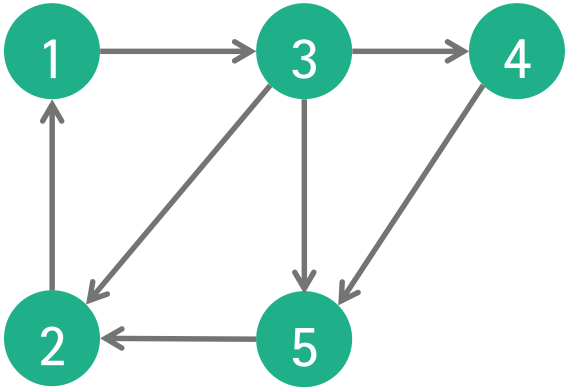
Simple Path, Simple Cycle

- 경로, 사이클에서 같은 정점을 다시 방문하지 않는 경로,사이클
- 문제를 푸는 과정에서 경로, 사이클 이라는 단어가 나오면 항상 단순 경로, 단순 사이클이라고 생각해도 무방하다.

방향 그래프

Directed Graph

- 간선에 방향이 있다.
- 1 -> 3 가능, 3 -> 1 불가능

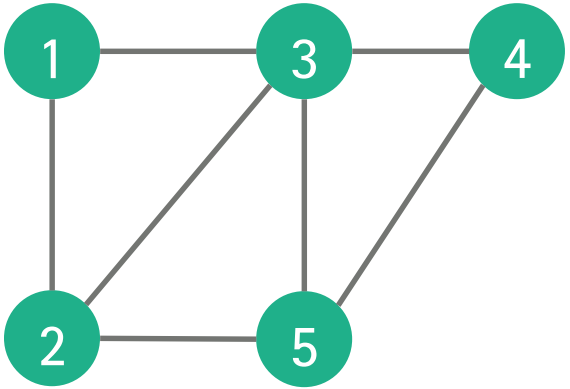


무 방향 그래프

Undirected Graph

- 간선에 방향이 없다.

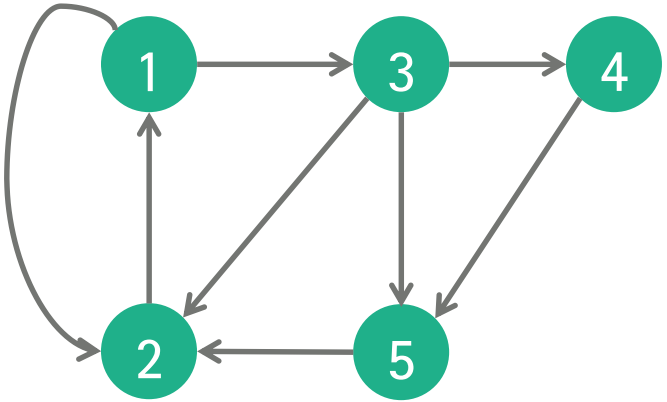
- 즉, 양 방향성을 뜻하기도 한다.
- Ex) 1 - 3 = (1 -> 3), (3 -> 1)



간선이 여러개

Multiple Edge

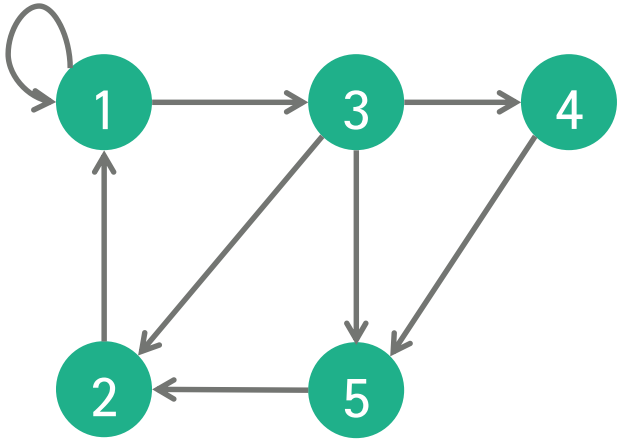
- 정점간 간선이 하나만 있을 이유가 없다.



루프

Loop

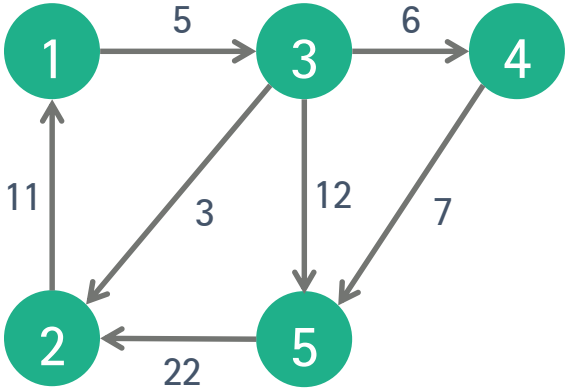
- 간선의 시작점과 도착점이 같을 수도 있다.
- 1 -> 1



가중치

Weight, Value

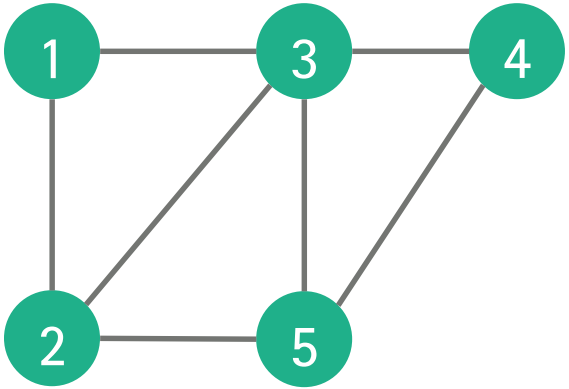
- 간선의 가중치가 의미하는 것은 문제에 따라서 달라진다.
- 이동 거리, 이동 시간, 비용….
- 가중치에 대한 언급이 없으면 가중치는 1이라고 생각하면 된다.



차수

Degree

- 정점과 연결된 간선의 개수
- 1의 차수 : 2
- 3의 차수 : 4
- 5의 차수 : 3

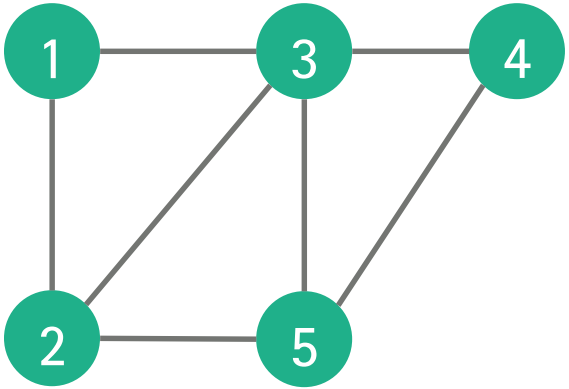


그래프의 표현

그래프의 표현

그래프를 코드로, 추상화라고도 한다.

- 정점이 5개, 간선이 7개 있다.
- 무 방향 그래프이다.
- 정점 : {1, 2, 3, 4, 5}



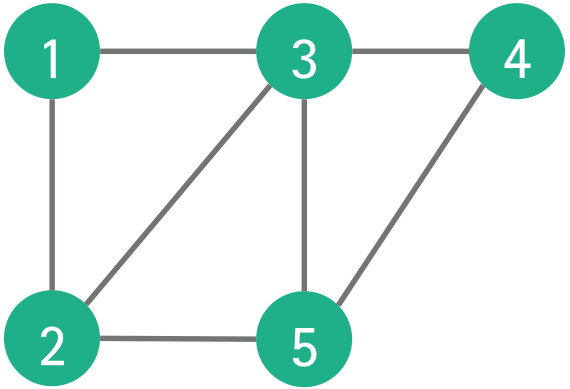
- 간선 : {(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5)}

그래프의 표현

문제에서 인풋이 들어오는 방식

- 정점의 개수 N, 간선의 개수 M 일 때

```
5 7 // N, M
1 2
1 3
2 3
2 5
3 4
3 5
4 5
```

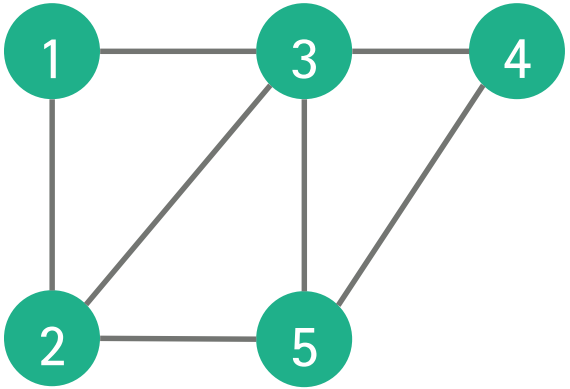


- 문제에서 주로 위와 같은 방식으로 input이 들어온다.
- 첫 째 줄에 정점과 간선의 개수, 둘 째 줄부터 간선의 정보

인접 행렬

Adjacency Matrix

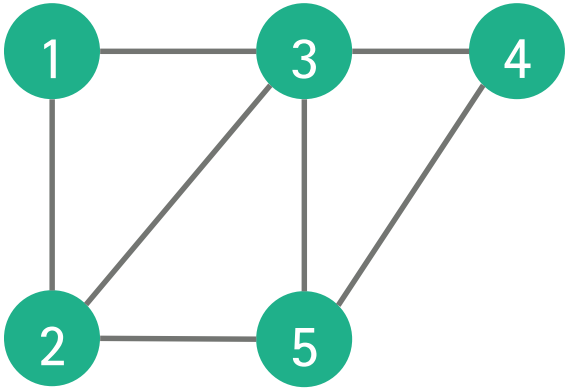
- 정점의 개수가 N개 일 때
- N X N 크기의 이차원 배열을 이용!
- G[i][j] =
  1 (i -> j 간선이 있을 때)
  0 (간선이 없을 때)



인접 행렬

Adjacency Matrix

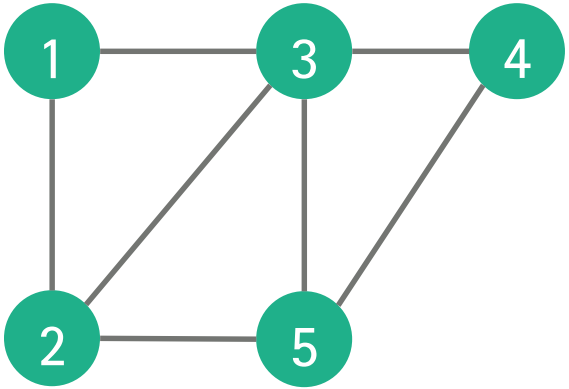
	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	0	1
3	1	1	0	1	1
4	0	0	1	0	1
5	0	1	1	1	0



인접 행렬

Input을 받아서 graph 2차원 배열에 추상화 하는 코드

```
#include <stdio.h>

//그래프 인접행렬은 전역변수가 좋다
int graph[6][6];

int main(void) {
    int n, m;
    scanf("%d %d",&n,&m);
    for (int i=0; i<m; i++) {
        int f, t;
        scanf("%d %d",&f,&t);
        //무 방향(양 방향) 그래프이기 때문에
        graph[f][t] = graph[t][f] = 1;
    }
    return 0
}
```

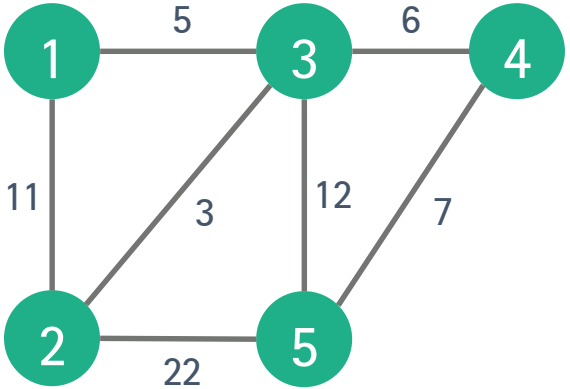


그래프의 표현

가중치가 있는 그래프의 표현

- 정점의 개수 N, 간선의 개수 M 일 때

```
5 7 // N, M
1 2 11
1 3 5
2 3 3
2 5 22
3 4 6
3 5 12
4 5 7
```

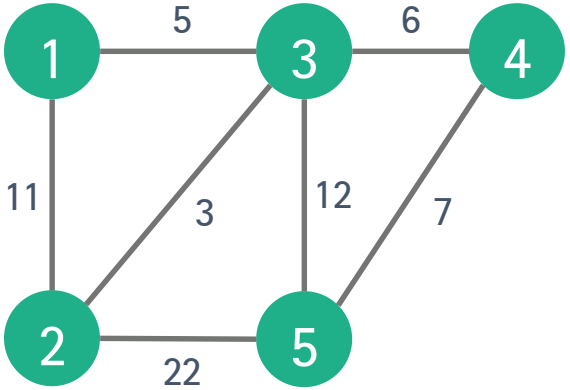


- 문제에서 주로 위와 같은 방식으로 input이 들어온다.
- 첫 째 줄에 정점과 간선의 개수, 둘 째 줄부터 간선의 정보

인접 행렬

Adjacency Matrix

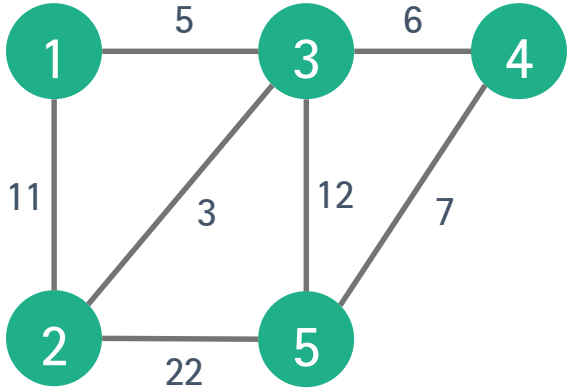
- 정점의 개수가 N개 일 때
- N X N 크기의 이차원 배열을 이용!
- G[i][j] =
  w (i -> j 간선이 있을 때 그 가중치)
  0 (간선이 없을 때)



인접 행렬

Adjacency Matrix

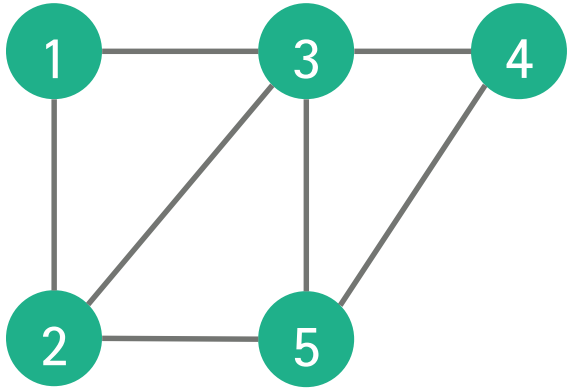
	1	2	3	4	5
1	0	11	5	0	0
2	11	0	3	0	22
3	5	3	0	6	12
4	0	0	6	0	7
5	0	22	12	7	0



인접 행렬

Input을 받아서 graph 2차원 배열에 추상화 하는 코드

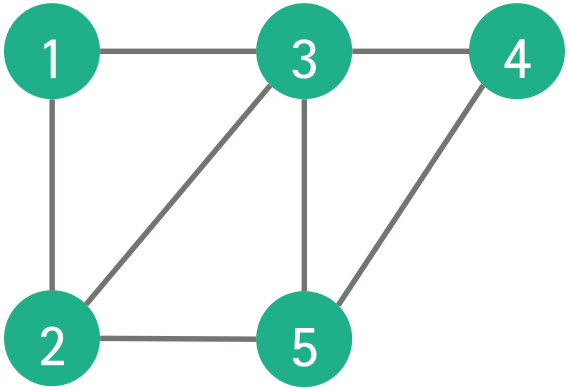
```
#include <stdio.h>

//그래프 인접행렬은 전역변수가 좋다
int graph[6][6];

int main(void) {
    int n, m;
    scanf("%d %d",&n,&m);
    for (int i=0; i<m; i++) {
        int f, t, w;
        scanf("%d %d %d",&f,&t,&w);
        //무 방향(양 방향) 그래프이기 때문에
        graph[f][t] = graph[t][f] = w;
    }
    return 0
}
```



인접 리스트

Adjacency List

- 링크드 리스트를 이용하여 구현 한다.
- G[i] : i가 시작점으로 연결된 간선의 도착 정점을 리스트로 포함



인접 리스트

Adjacency List

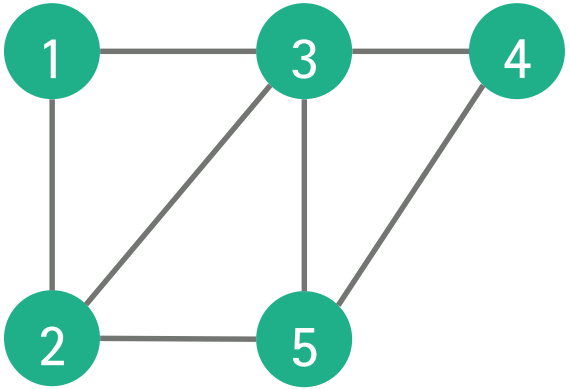
G[1] 2 3

G[2] 1 3 5

G[3] 1 2 4 5

G[4] 3 5

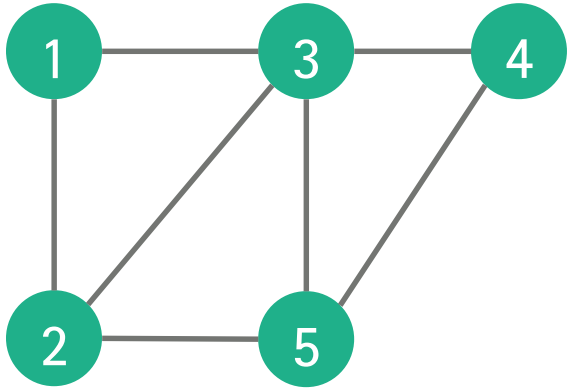
G[5] 2 3 4



인접 리스트

Adjacency List

- 링크드 리스트는 구현이 어렵다
- STL의 vector를 이용한다.
- Vector : 길이를 변경할 수 있는 배열
- 문제를 푸는 과정에서 링크드 리스트를 구현할 일은 없다.



인접 리스트

Vector의 간단한 사용법

- 앞에서 말했듯 vector는 길이를 변경할 수 있는 배열이다. 배열이기 때문에 당연히 2차원 배열을 만들 수 있다.
- Vector<vector<int> > vec(N+1); 과 같이 선언한다.
- (N+1)의 의미 : 2차원 벡터는 몇 개의 행을 사용할 것 인지 선언과 함께 결정해야 한다. (0 ~ N행 까지 N+1개의 행)
- Vector는 []연산자로 접근이 가능하다. 어려워 보이지만 코드를 보면 빠르게 이해가 가능하다.

인접 리스트

Vector의 간단한 사용법

- push_back() 함수를 사용할 수 있어야 한다. push_back의 의미는 배열의 마지막에 원소를 하나 추가하고 길이를 늘리는 연산이다.

- vector = {1, 2, 3, 4, 5}인 길이 5의 벡터가 있다.

  vector.push_back(10);

- vector = {1, 2, 3, 4, 5, 10}인 길이가 6인 벡터가 되었다.

인접 리스트

Vector의 간단한 사용법

- vector = {1, 2, 3, 4, 5, 10}인 길이가 6인 벡터가 되었다.
- 배열과 마찬가지로 []연산자를 통해서 원소에 접근한다.
- Ex) vector[0] = 1, vector[5] = 10이 된다.

인접 리스트

Vector의 간단한 사용법

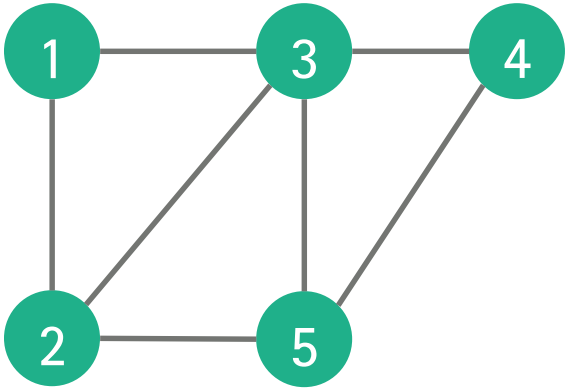
- vector<int> vector;는 int형 원소를 담는 벡터를 선언한다.
- 그렇다면 vector< vector<int> > vector;과 같이 선언하면?
- 벡터가 들어있는 벡터를 선언! = 배열이 들어있는 배열을 선언 = 리스트가 들어있는 배열을 선언! = 인접 리스트
- 2차원 벡터는 2차원 배열과 같이 [][]연산자로 접근할 수 있다.
- vector[i][j] = 2차원 벡터의 i행 벡터에서 j번째 원소를 뜻한다.

그래프의 표현

문제에서 인풋이 들어오는 방식

- 정점의 개수 N, 간선의 개수 M 일 때

```
5 7 // N, M
1 2
1 3
2 3
2 5
3 4
3 5
4 5
```



- 문제에서 주로 위와 같은 방식으로 input이 들어온다.
- 첫 째 줄에 정점과 간선의 개수, 둘 째 줄부터 간선의 정보

인접 리스트

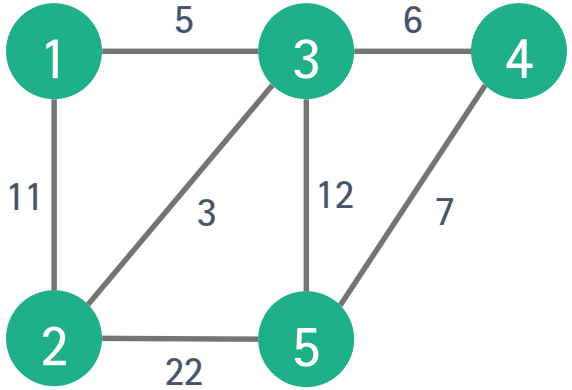
Input을 받아서 graph 2차원 벡터에 추상화 하는 코드

```
#include <cstdio>
#include <vector>
using namespace std;
int main() {
    int n, m;
    scanf("%d %d",&n,&m);
    //n+1개의 행을 갖는 2차원 벡터 선언
    vector<vector<int>> graph(n+1);
    for (int i=0; i<m; i++) {
        int u,v;
        scanf("%d %d",&u,&v);
        //u번 벡터에 v를 추가
        graph[u].push_back(v);
        //v번 벡터에 u를 추가
        graph[v].push_back(u);
    }
}
```

인접 리스트

Adjacency List

- 가중치가 있는 그래프도 마찬가지…
- 링크드 리스트를 이용하여 구현 한다!
- G[i] : i가 시작점으로 연결된 간선의 도착 정점과 가중치를 리스트로 포함



- 도착 정점과 가중치를 저장해야 한다. 1개가 넘는 데이터를 동시에 저장해야 한다! = 구조체!!!

인접 리스트

Adjacency List

- C언어 에서는 사용하는 구조체를 통해서 구현해도 무방하다.
- 구조체에 대한 설명은 생략하고 C++ STL의 pair를 이용한 구현을 알아보도록 한다.
- pair는 이름에서 알 수 있듯이 자료 2개를 묶어서 저장할 수 있는 자료구조 이다.

인접 리스트

Pair의 간단한 사용법

- pair<자료형, 자료형> 변수명; 과 같이 선언한다.
- pair<int, char> p;로 선언했다면 int형 변수와 char형 변수를 묶어서 하나의 pair로 저장 할 수 있다.
- pair에 값을 집어 넣을 때는 make_pair()함수를 이용한다.
- Ex) pair<int, char> p = make_pair(1, 'A');

인접 리스트

Pair의 간단한 사용법

- Ex) pair<int, char> p = make_pair(1, ‘A’);
- 2개의 자료에 접근할 때는 .메서드를 이용한다.
- .first : 1번 자료에 접근, .second : 2번 자료에 접근
- Ex) printf(“pair : %d %c\n”, p.first, p.second);
- 결과
  pair : 1 A

인접 리스트

Adjacency List

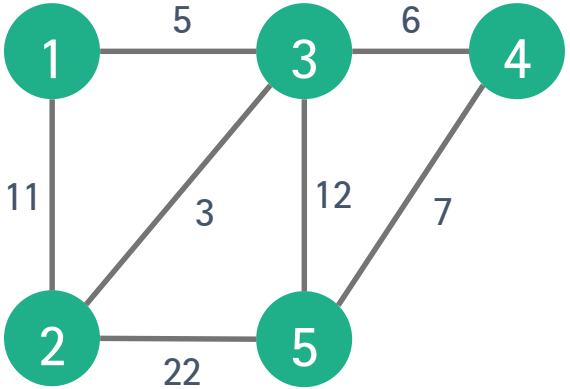
G[1] (2, 11) (3, 5)

G[2] (1, 11) (3, 3) (5, 22)

G[3] (1, 5) (2, 3) (4, 6) (5, 12)

G[4] (3, 6) (5, 7)

G[5] (2, 22) (3, 12) (4, 7)

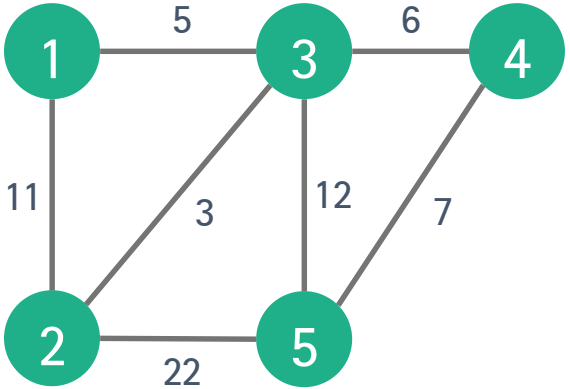


그래프의 표현

가중치가 있는 그래프의 표현

- 정점의 개수 N, 간선의 개수 M 일 때

```
5 7 // N, M
1 2 11
1 3 5
2 3 3
2 5 22
3 4 6
3 5 12
4 5 7
```



- 문제에서 주로 위와 같은 방식으로 input이 들어온다.
- 첫 째 줄에 정점과 간선의 개수, 둘 째 줄부터 간선의 정보

인접 리스트

Input을 받아서 graph 2차원 벡터에 추상화 하는 코드

```
#include <cstdio>
#include <vector>
using namespace std;

//10의 행을 갖는 2차원 벡터 (행의 개수를 초기화 하는 방법이 여러 개가 있다.)
vector<pair<int,int>> graph[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);
        graph[u].push_back(make_pair(v,w));
        graph[v].push_back(make_pair(u,w));
    }
}
```

공간 복잡도

인접 배열과 인접 리스트의 공간 복잡도

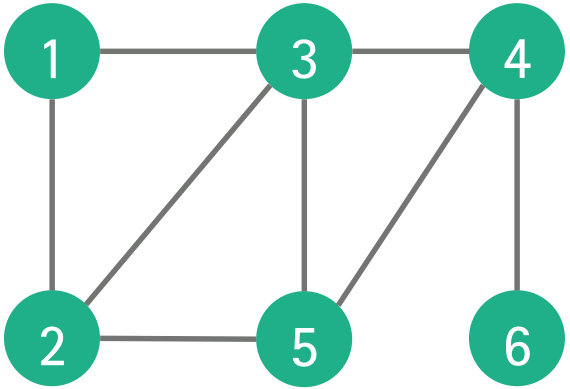
- 인접 행렬의 공간 복잡도 : V^2 (정점 개수의 제곱)
- 인접행렬의 구현이 더 편해서 많이 이용된다. 하지만 정점의 개수가 많아져서 인접행렬로는 구현이 불가능 할 때가 있다.(약 V >= 4000)
- 인접 리스트의 공간 복잡도 : E (간선의 개수)
- 다시 말해서 두 개의 방법 모두 다 숙련되어야 한다.

그래프의 탐색

그래프의 표현

DFS와 BFS

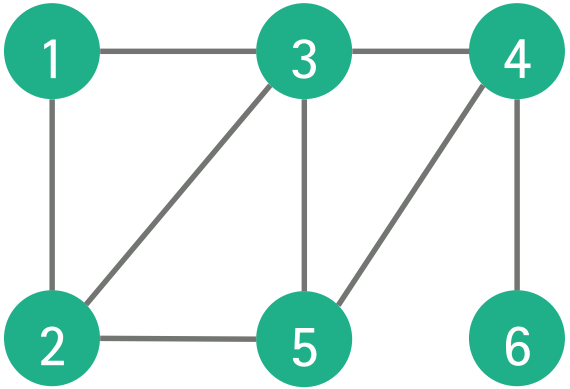
- DFS : 깊이 우선 탐색
- BFS : 너비 우선 탐색



깊이 우선 탐색

Depth First Search

- 스택을 이용하여 갈 수 있는 만큼 최대한 많이 간다.
- 더 이상 갈 수 있는 정점이 없으면 이전 정점으로 되돌아간다.

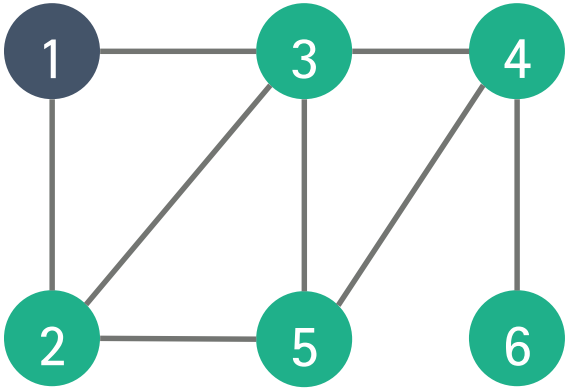


- 스택이 따로 필요 없다. 함수의 재귀 콜이 스택 구조를 띄기 때문이다.

깊이 우선 탐색

Depth First Search

- 현재 정점 : 1
- 순서 : 1
- 스택 : 1

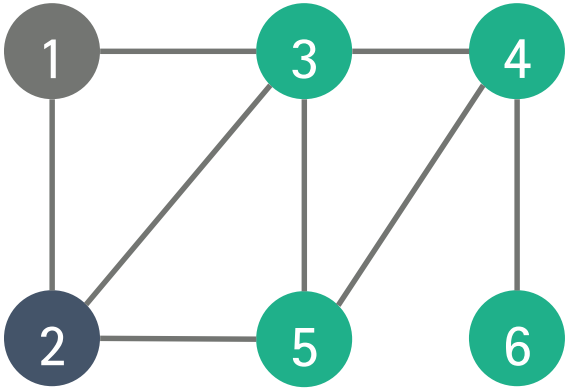


i	1	2	3	4	5	6
Visited[i]	1	0	0	0	0	0

깊이 우선 탐색

Depth First Search

- 현재 정점 : 2
- 순서 : 1 2
- 스택 : 1 2

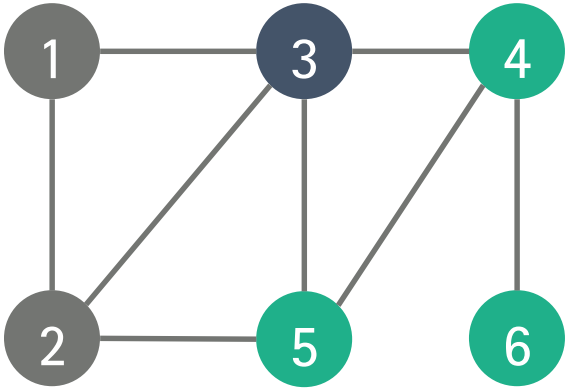


i	1	2	3	4	5	6
Visited[i]	1	1	0	0	0	0

깊이 우선 탐색

Depth First Search

- 현재 정점 : 3
- 순서 : 1 2 3
- 스택 : 1 2 3

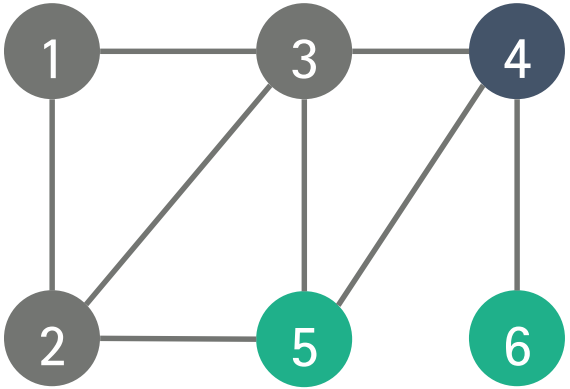


i	1	2	3	4	5	6
Visited[i]	1	1	1	0	0	0

깊이 우선 탐색

Depth First Search

- 현재 정점 : 4
- 순서 : 1 2 3 4
- 스택 : 1 2 3 4

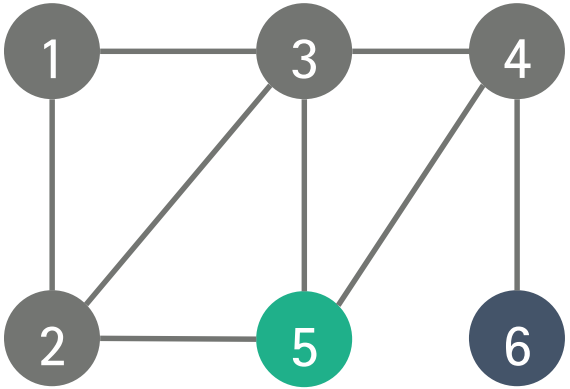


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	0	0

깊이 우선 탐색

Depth First Search

- 현재 정점 : 6
- 순서 : 1 2 3 4 6
- 스택 : 1 2 3 4 6

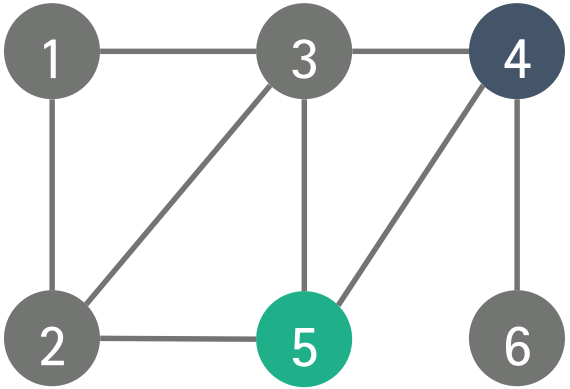


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	0	1

깊이 우선 탐색

Depth First Search

- 현재 정점 : 4
- 순서 : 1 2 3 4 6
- 스택 : 1 2 3 4
- 6에서 더 갈 수 있는 곳이 없기 때문에 4로 돌아간다

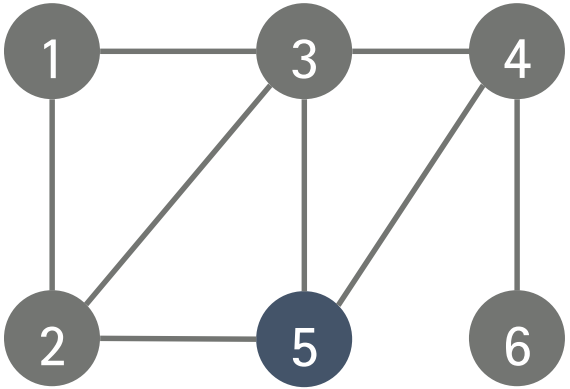


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	0	1

깊이 우선 탐색

Depth First Search

- 현재 정점 : 5
- 순서 : 1 2 3 4 6 5
- 스택 : 1 2 3 4 5

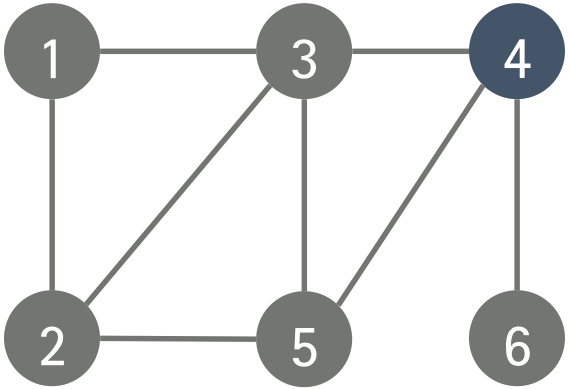


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	1	1

깊이 우선 탐색

Depth First Search

- 현재 정점 : 4
- 순서 : 1 2 3 4 6 5
- 스택 : 1 2 3 4
- 5에서 더 갈 수 있는 곳이 없기 때문에 4로 돌아간다

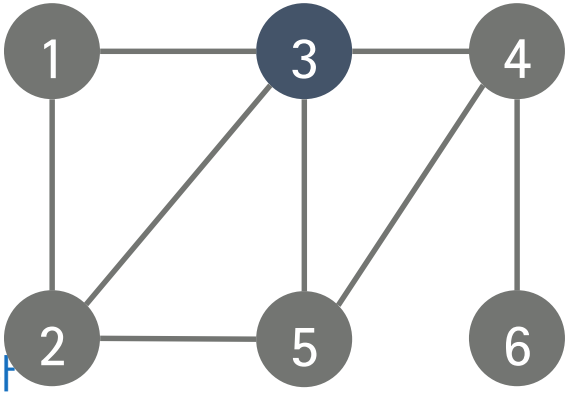


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	1	1

깊이 우선 탐색

Depth First Search

- 현재 정점 : 3
- 순서 : 1 2 3 4 6 5
- 스택 : 1 2 3
- 4에서 더 갈 수 있는 곳이 없기 때문에 3으로 돌아간다

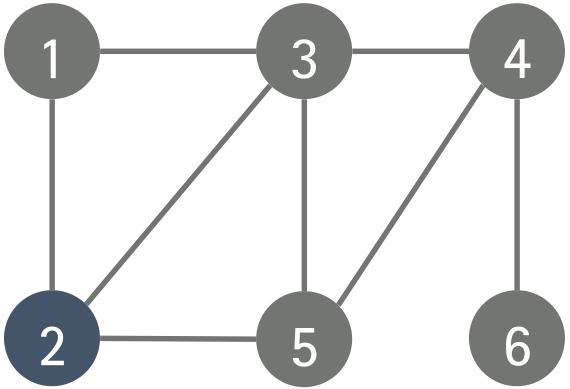


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	1	1

깊이 우선 탐색

Depth First Search

- 현재 정점 : 2
- 순서 : 1 2 3 4 6 5
- 스택 : 1 2
- 3에서 더 갈 수 있는 곳이 없기 때문에 2로 돌아간다

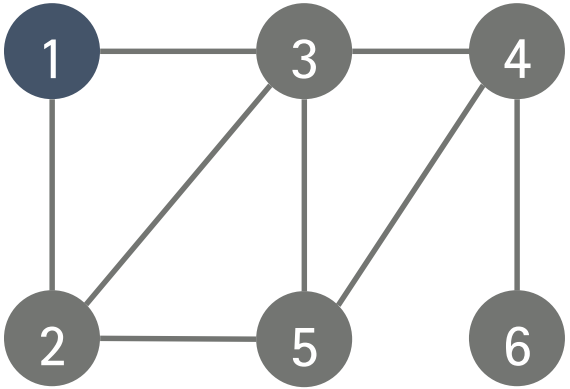


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	1	1

깊이 우선 탐색

Depth First Search

- 현재 정점 : 1
- 순서 : 1 2 3 4 6 5
- 스택 : 1
- 2에서 더 갈 수 있는 곳이 없기 때문에 1로 돌아간다

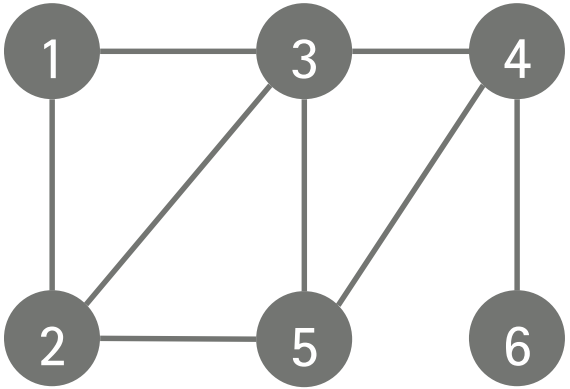


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	1	1

깊이 우선 탐색

Depth First Search

- 현재 정점 :
- 순서 : 1 2 3 4 6 5
- 스택 :
- 탐색종료

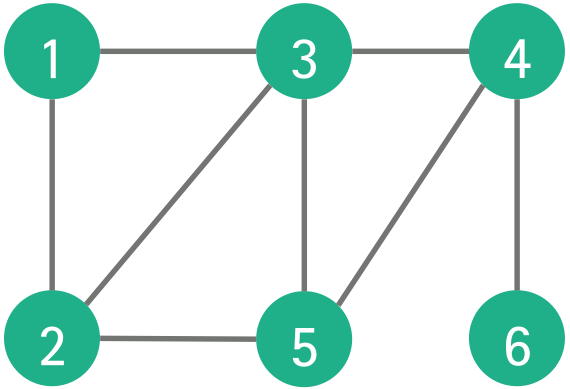


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	1	1

너비 우선 탐색

Breadth First Search

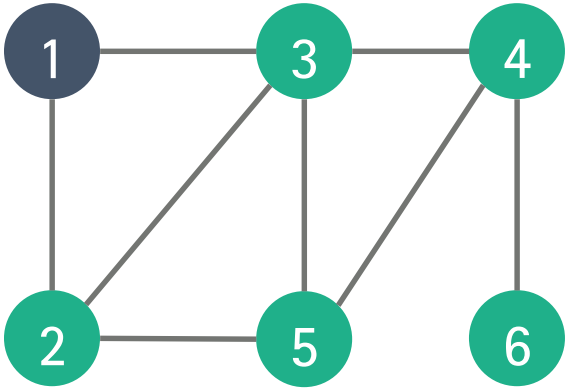
- 큐를 이용해서 지금 위치에서
  갈 수 있는 것을 모두 큐에 넣는 방식
- 큐에 넣을 때 방문 했다고 체크한다.
- 스택이 필요없는 DFS와 달리
  BFS는 큐가 필요하다. STL의 큐를 이용한다.



너비 우선 탐색

Breadth First Search

- 현재 정점 : 1
- 순서 : 1
- 큐 : 1

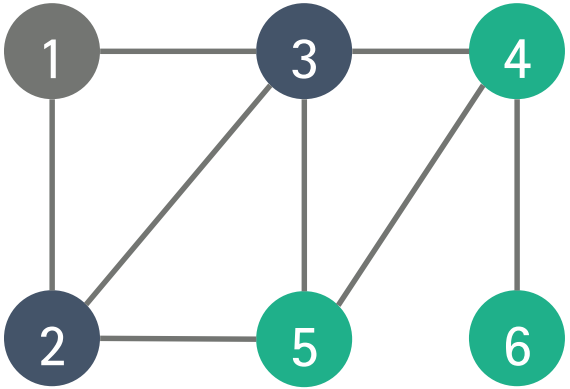


i	1	2	3	4	5	6
Visited[i]	1	0	0	0	0	0

너비 우선 탐색

Breadth First Search

- 현재 정점 : 1
- 순서 : 1 2 3
- 큐 : 2 3
- 큐에서 1을 pop. 1에서 갈 수 있는 2, 3을 push.

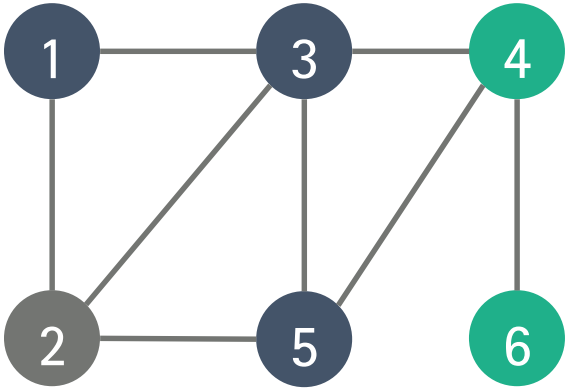


i	1	2	3	4	5	6
Visited[i]	1	1	1	0	0	0

너비 우선 탐색

Breadth First Search

- 현재 정점 : 2
- 순서 : 1 2 3 5
- 큐 : 3 5
- 큐에서 2을 pop. 1에서 갈 수 있는 5를 push.

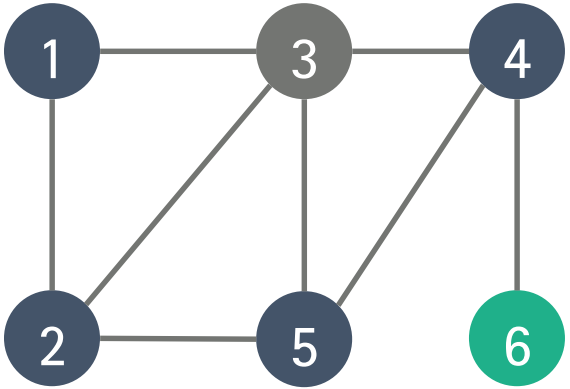


i	1	2	3	4	5	6
Visited[i]	1	1	1	0	1	0

너비 우선 탐색

Breadth First Search

- 현재 정점 : 3
- 순서 : 1 2 3 5 4
- 큐 : 5 4
- 큐에서 3을 pop. 1에서 갈 수 있는 4를 push.

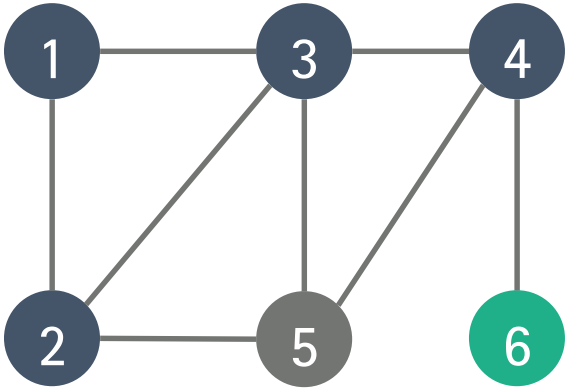


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	1	0

너비 우선 탐색

Breadth First Search

- 현재 정점 : 5
- 순서 : 1 2 3 5 4
- 큐 : 4
- 큐에서 5을 pop. 갈 곳이 없기 때문에 아무일도 안함

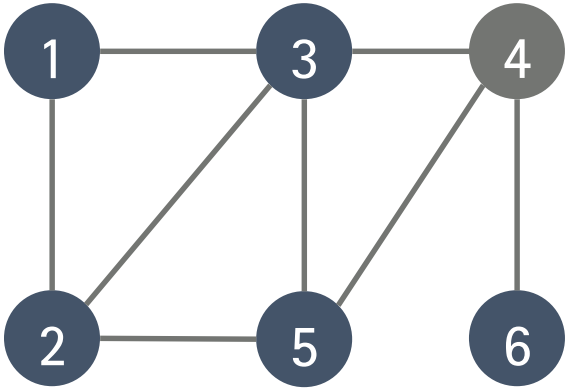


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	1	0

너비 우선 탐색

Breadth First Search

- 현재 정점 : 4
- 순서 : 1 2 3 5 4 6
- 큐 : 6
- 큐에서 4을 pop. 갈 곳이 없기 때문에 아무일도 안함

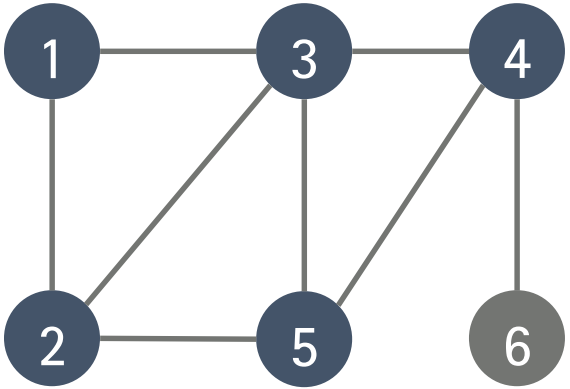


i	1	2	3	4	5	6
Visited[i]	1	1	1	1	1	1

너비 우선 탐색

Breadth First Search

- 현재 정점 : 6
- 순서 : 1 2 3 5 4 6
- 큐 :
- 큐에서 6을 pop. 탐색 종료



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

DFS와 BFS

DFS와 BFS

https://www.acmicpc.net/problem/1260

- 문제와 코드를 통해서 감각을 익히도록 하자. 문제와 코드로 이해하는 것이 가장 빠르다.

- C++ code : https://github.com/OfficialDominyellow/AlgorithmByDominyellow/blob/master/BackjoonOnlineJudge/1260.cc

숨바꼭질

숨바꼭질

https://www.acmicpc.net/problem/1697

- BFS의 가장 전형적인 문제! 이 문제를 푸는 것이 BFS를 이용한 문제풀이의 시작이다.
- 구조체와 pair까지 연습할 수 있는 좋은 문제

숨바꼭질

https://www.acmicpc.net/problem/1697

- 현재 위치를 N, 동생이 있는 위치를 K라고 한다.
- 수빈이의 위치가 X라고 한다면 1초 후에 X-1, X+1, 2X 3가지의 경우로 이동할 수 있다.
- 동생을 찾는 **가장 빠른 시간**을 구해라.

숨바꼭질

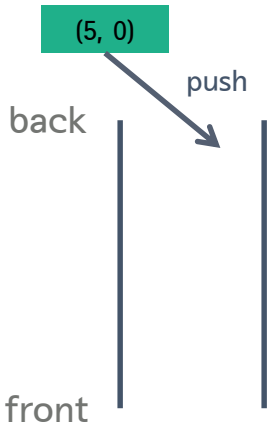
https://www.acmicpc.net/problem/1697

- BFS는 앞에서 설명했듯이 최단경로와 아주 깊은 관련이 있다.
- 큐를 이용해서 BFS 코드를 구현하는 것 보다 문제를 보고 BFS로 해결할 수 있다는 것을 파악하는 것이 매우 중요하다.
- 큐와 visited(방문 표시)배열이 필요하다. 큐에는 (현재위치, 시간)을 묶어서 집어넣어야 한다. 구조체 혹은 pair를 이용한다.

숨바꼭질

https://www.acmicpc.net/problem/1697

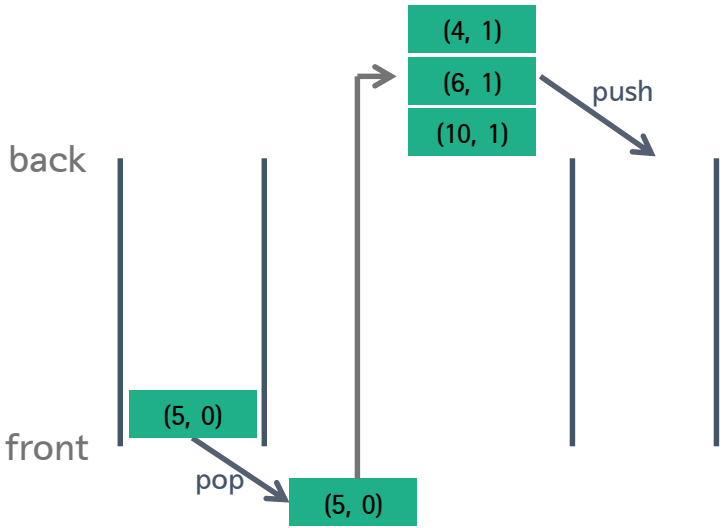
- 수빈이의 위치가 5에서 시작하기 때문에 (5, 0)을 큐에 push한다.



숨바꼭질

https://www.acmicpc.net/problem/1697

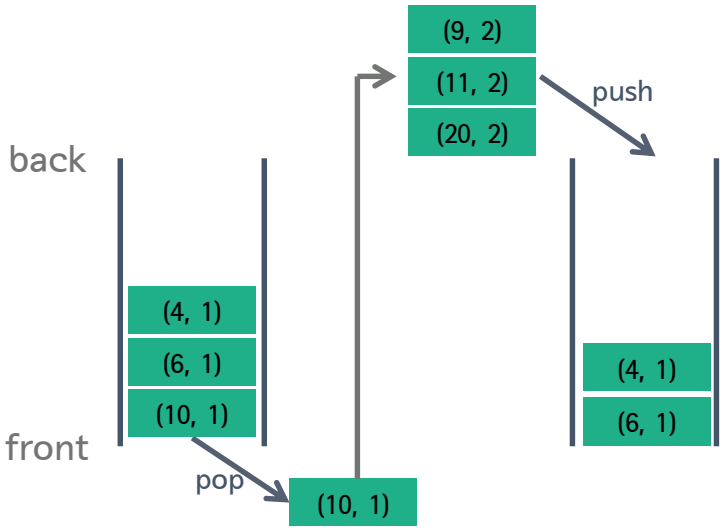
- 5에서 갈 수 있는 5-1, 5+1, 5*2와 시간을 +1해서 push한다.



숨바꼭질

https://www.acmicpc.net/problem/1697

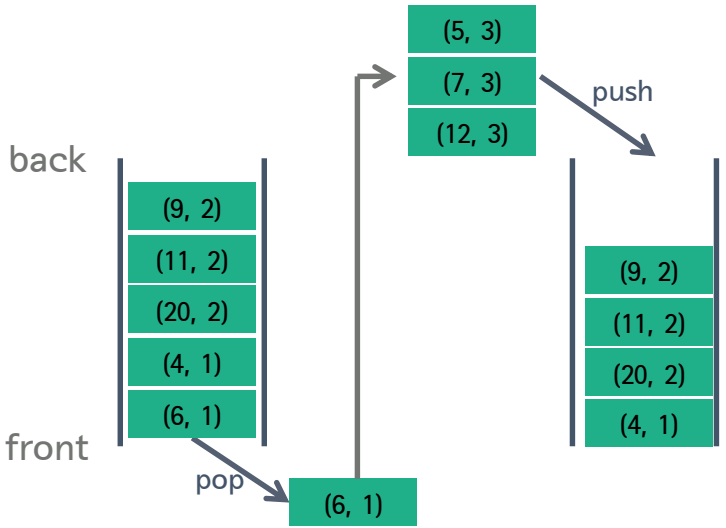
- 10에서 갈 수 있는 10-1, 10+1, 10*2와 시간을 +1해서 push한다.



숨바꼭질

https://www.acmicpc.net/problem/1697

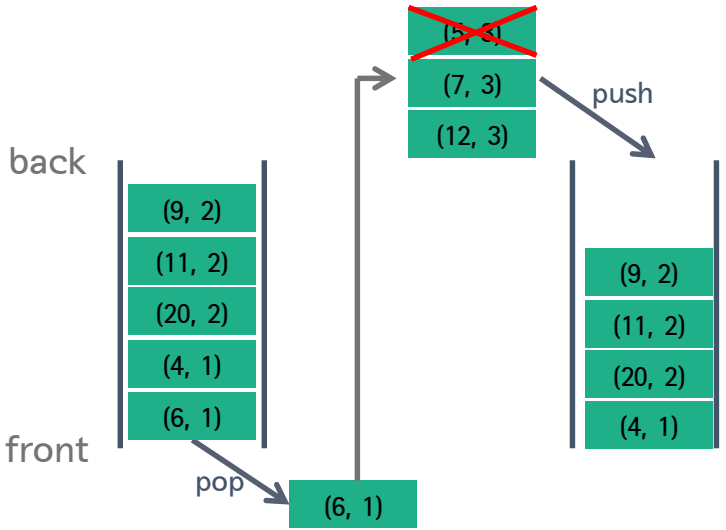
- 6에서 갈 수 있는 6-1, 6+1, 6*2와 시간을 +1해서 push한다.



숨바꼭질

https://www.acmicpc.net/problem/1697

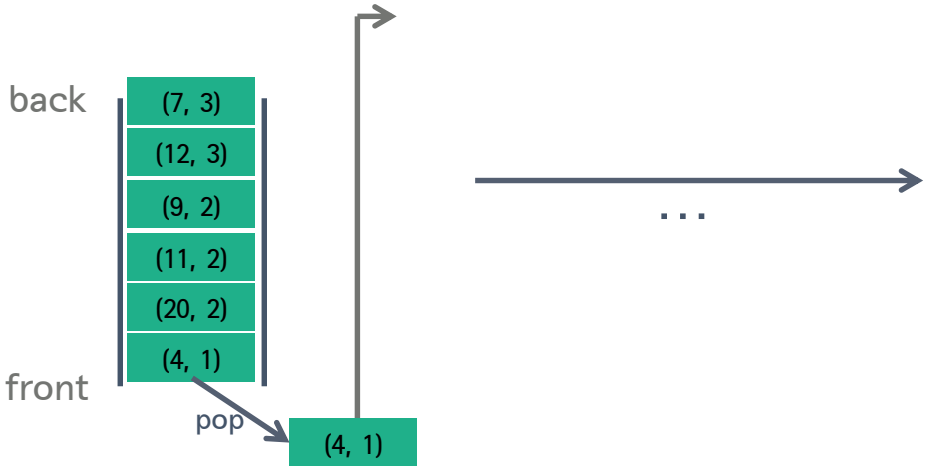
- 주의! 5, 7, 12중에서 5는 한번 방문했던 적이 있는 정점!
- Visited배열을 통해서 재방문을 하지 않는다!



숨바꼭질

https://www.acmicpc.net/problem/1697

- 이 같은 방법으로 동생의 위치 K가 탐색될 때 까지 BFS를 반복한다.



숨바꼭질

https://www.acmicpc.net/problem/1697

- 주의 : visited배열을 통해서 재방문을 하지 않게 한다.
- X+1, X-1, 2*X를 탐색하는 과정에서 1 ~ 100000의 범위 안에 있는지 꼭 확인해야 한다.
- C++ code : https://github.com/OfficialDominyellow/AlgorithmByDominyellow/blob/master/BackjoonOnlineJudge/1697.cc