그래프
(Graph)

- 그래프 정의
- 용어 설명
- 그래프 표현
- 그래프 운행 : 깊이우선탐색과 너비우선탐색

4. 그래프 탐색(운행)

- 깊이우선탐색(DFS: Depth First Search)
- 너비우선탐색(BFS: Breadth First Search)

시험문제출제
중간고사 스타일봐서는
원리를 알아야됨

4.1 깊이우선탐색

- 깊이우선탐색(DFS: Depth First Search)
  - 그래프의 모든 정점을 한번만 방문하는 알고리즘
  - 트리에서의 Preorder Traversal의 일반화
  - 시작정점 v: 문제에서 정해지기도 하고 임의로 정할 수도 있음

시작정점부터 방문한다. 현재 방문하고 있는 정점 v와 인접한 정점을 하나씩 검사하면서 아직 방문되지 않은 (v와 인접한) 정점 w가 있으면 이 정점 w를 방문한다. 이 과정을 반복하면서 더 이상 갈 곳이 없는 정점(인접한 정점들이 모두 방문된 경우)에 도달하면 마지막에 따라왔던 간선을 따라 뒤로 돌아가서(backtrack), 인접한 정점을 방문하는 탐색을 반복한다.

깊이우선탐색 골격

- 깊이우선탐색(DFS: Depth First Search) 골격

recursive version

Algorithm dfs(g, v) // g: 그래프, 시작정점이 v인 깊이우선탐색

v를 방문하고 이를 “방문되었다고” 표시한다.

// v를 방문할 때 필요한 작업 수행

v와 인접한 각 정점 w에 대하여

만약 w가 방문되지 않았다면 dfs(g, w)

// w로 부터 backtrack 할 때 필요한 작업 수행

v가 “종료되었다”고 표시한다.

// v에서 시작하는 깊이우선탐색을 마친 후 필요한 작업 수행

깊이우선탐색 정점 상태

- 깊이우선탐색(DFS: Depth First Search)

정점 v의 상태:

(1) v를 아직 방문하지 않음(미방문 상태)

(2) v를 처음으로 방문

(3) v와 인접한 정점을 모두 방문 (v에서 시작하는 깊이우선탐색을 종료한 상태)

위의 세 가지 상태 중 (1), (2)만 필요한 경우도 있고, (1), (2), (3) 모두 필요한 경우도 있다.

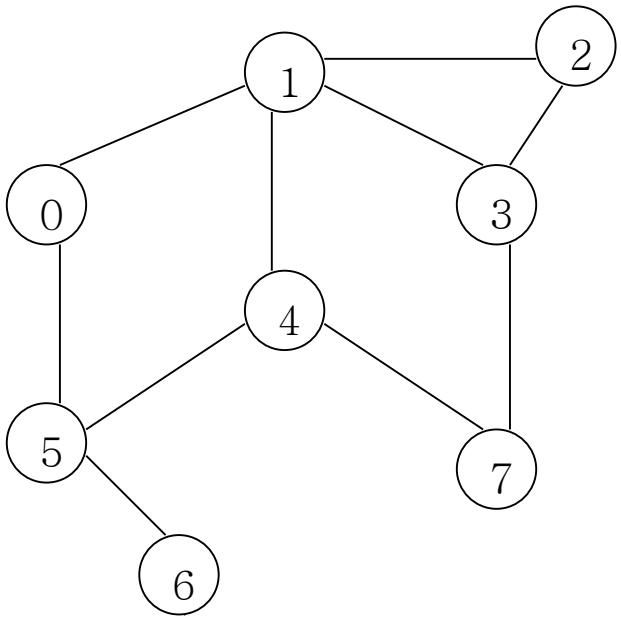
- 깊이우선탐색을 이용하여 그래프 g의 모든 정점을 방문하는 알고리즘

Algorithm dfsVisit(g)

G의 모든 정점을 “미방문” 상태로 초기화
G의 각 정점 v에 대하여,
만약 v가 미방문 상태라면,
dfs(g, v)

깊이우선탐색 예 (무방향그래프)

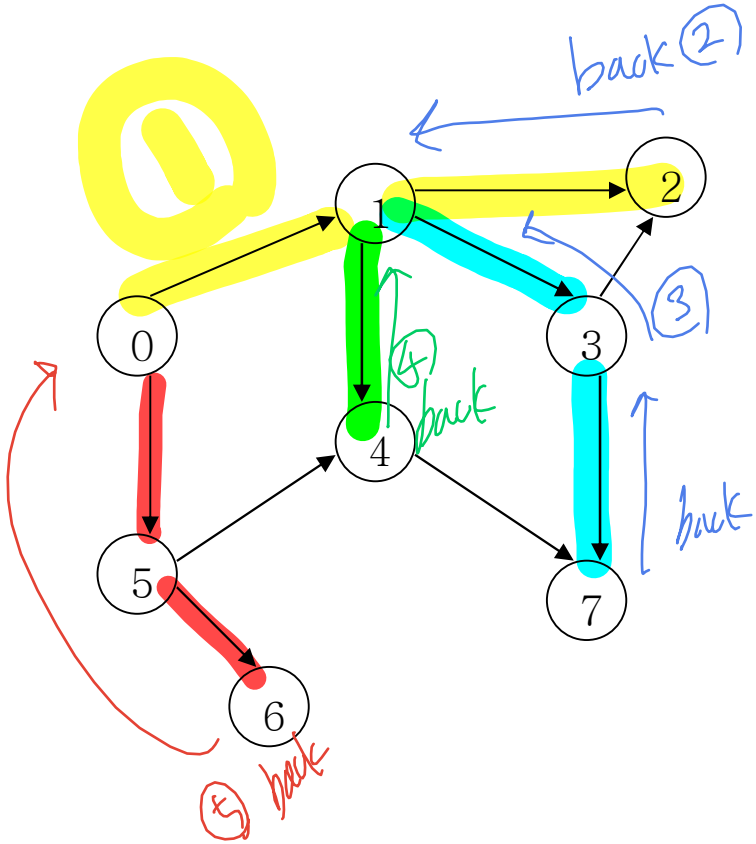
- 깊이우선탐색 예



출발정점: 0
깊이우선순서로 탐색시에 방문되는 정점들의 순서: 0, 1, 4, 5, 6, 7, 3, 2

깊이우선탐색 예 (방향그래프)

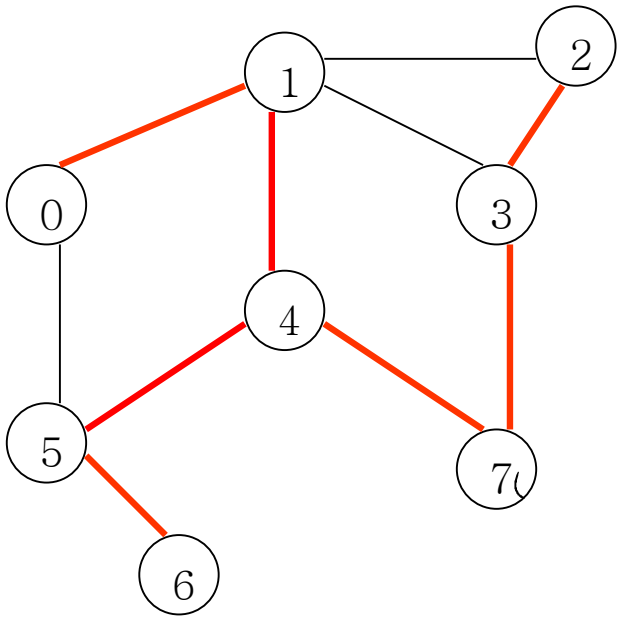
- 방향그래프의 깊이우선탐색



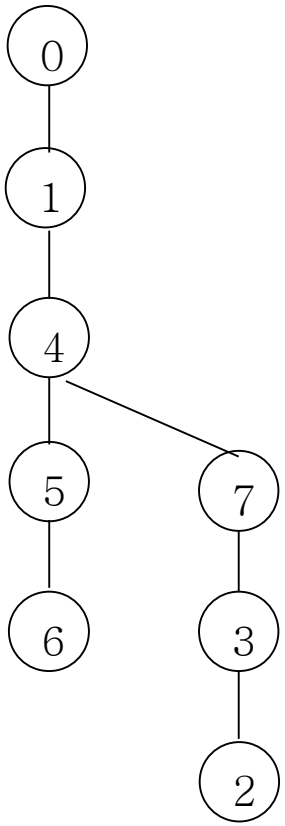
출발정점: 0
깊이우선순서로 탐색시에 방문되는 정점들의 순서:
0, 1, 2, 3, 7, 4, 5, 6

깊이우선탐색 트리 (무방향그래프)

- 깊이우선탐색시, 각 정점 u에 대하여 u와 인접한 미방문 정점 v를 방문할 때, 에지 (u,v)들로 구성된 root가 있는 트리 (v가 u의 child가 됨) : root는 시작정점임



출발정점: 0
깊이우선순서로 탐색시에 방문되는 정점들의 순서: 0, 1, 4, 5, 6, 7, 3, 2



DFS 트리

깊이우선탐색 예 1: 일차원 셀 방문하기

다음과 같이 n(1이상 5,000이하 정수)개의 셀들로 구성된 하나의 행이 있다. 각 셀에는 음이 아닌 정수가 있고, 이들 셀 중 하나(아래 그림의 동그라미 셀)에 마커가 있다.

(3)	6	4	1	3	4	2	5	3	0

각 단계에서 마크에 있는 셀의 정수만큼 왼쪽 혹은 오른쪽으로 마크를 이동할 수 있다. 단 왼쪽 끝 혹은 오른쪽 끝을 벗어날 수 없다. 아래 그림은 시작 위치에 있는 마커가 도달할 수 있는 셀 들의 일부를 보여준다.

Starting position	(3)	6	4	1	3	4	2	5	3	0
Step 1: Move right	3	6	4	(1)	3	4	2	5	3	0
Step 2: Move left	3	6	(4)	1	3	4	2	5	3	0
Step 3: Move right	3	6	4	1	3	4	(2)	5	3	0
Step 4: Move right	3	6	4	1	3	4	2	5	(3)	0
Step 5: Move left	3	6	4	1	3	(4)	2	5	3	0
Step 6: Move right	3	6	4	1	3	4	2	5	3	(0)

일차원 셀 방문하기

다음과 같이 n(1이상 5,000이하 정수)개의 셀들로 구성된 하나의 행이 있다. 각 셀에는 음이 아닌 정수가 있고, 이들 셀 중 하나(아래 그림의 동그라미 셀)에 마커가 있다. 시작 셀로부터 도달 가능하는 셀들의 위치를 구하는 프로그램을 작성하시오.

3	6	4	1	3	4	2	5	3	0

입력 예

10 // n

3 6 4 1 3 4 2 5 3 0

5 // 시작 셀 위치

일차원 셀 방문하기

```
void dfs(int v, int n, int cells[], bool visited[])
{
    visited[v] = true;
    cout << v << " ";
    // v에서 이동할 수 각 위치 (v의 왼쪽과 오른쪽)로 가 확인
    if (v - cells[v] >= 0) { // 왼쪽으로 갈 수 있는지 확인
        if (!visited[v-cells[v]])
            dfs(v-cells[v], n, cells, visited);
    }
    if (v + cells[v] < n) { // 오른쪽으로 갈 수 있는 지 확인
        if (!visited[v+cells[v]])
            dfs(v+cells[v], n, cells, visited);
    }
}
```

일차원 셀 방문하기

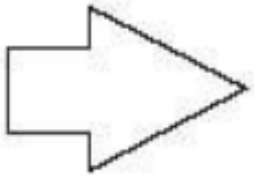
```
int main()
{
    int n;
    int startPosition;
    int *cells;
    bool *visited;
    scanf("%d", &n);
    cells = new int[n]; // 셀 배열
    visited = new bool[n];
    for (int i = 0; i < n; i++)
        scanf("%d", &cells[i]);
    scanf("%d", &startPosition);
    for (int i = 0; i < n; i++)
        visited[i] = false;
    dfs(startPosition, n, cells, visited);
    /*
    for (int i = 0; i < n; i++) {
        if (visited[i])
            cout << << " ";
    }
    */
    return 0;
}
```

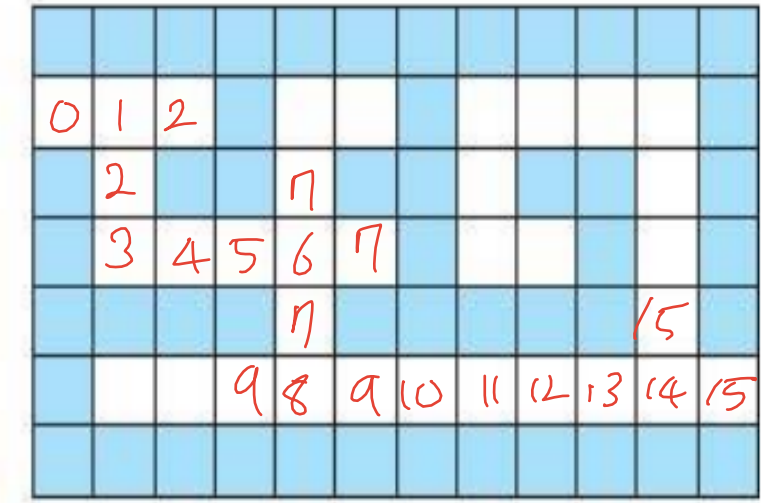
깊이우선탐색 예 2: 미로 문제

- 0은 통로의 일부, 1은 벽의 일부를 표현
- 출발지에서 목적지까지 가능 경로가 있는가?
- 출발지에서 목적지까지 최단경로를 구하라.

- m: 행의 개수, n: 열의 개수
- 아래 예에서 m = 7, n = 12

```
111111111111
000100100001
101101101101
100000100101
111101111101
100000000000
111111111111
```





미로에서 깊이우선탐색을 이용한 경로 찾기

```
// map: 미로 정보를 저장하는 2차원 배열
// m, n: 미로의 행과 열 크기
// start, dest: 출발지와 목적지
// visited: 미로의 행과 열의 각 위치가 방문되었는지 아닌지를 위한 2차원 배열
Algorithm findPath(map, m, n, start, dest, visited) // start에서 dest 까지 가는 경로가 있는지를 판별하는 알고리즘
    visited[start.row][start.col] = true
    print("(" , start.row , ", " , start.col, ")")
    if (start.row == dest.row and start.col == dest.col)  // 목적지에 도달
        return true
    if(start.col < n-1) // 오른쪽
        if (map[start.row][start.col+1] == 0 and not visited[start.row][start.col+1]) {
            next.row = start.row
            next.col = start.col+1
            if(findPath(map, m, n, next, dest, visited))
                return true
     if(start.row < m-1)   // 아래쪽
        if (map[start.row+1][start.col] == 0 and not visited[start.row+1][start.col]) {
            next.row = start.row + 1
            next.col = start.col
            if(findPath(map, m, n, next, dest, visited))
                return true
```

미로에서 깊이우선탐색을 이용한 길찾기

```
if(start.col > 0) // 왼쪽
    if (map[start.row][start.col-1] == 0 and not visited[start.row][start.col-1])
        next.row = start.row
        next.col = start.col-1
        if(findPath(map, m, n, next, dest, visited))
            return true

// 위쪽
 ....
return false
```

미로에서 깊이우선탐색을 이용한 길찾기- C++

```
#include <iostream>
using namespace std;
typedef struct {
                int row;
                int col;
} Location;
bool findPath(int** map, int m, int n, Location start, Location dest,  bool** visited)
{
   Location next;
   visited[start.row][start.col] = true;
//   cout << "(" << start.row << "," << start.col << ")" << endl;
   if (start.row == dest.row and start.col == dest.col)  // 목적지에 도달
      return true;
   if(start.col < n-1){  // 오른쪽
      if (map[start.row][start.col+1] == 0 and not visited[start.row][start.col+1]) {
         next.row = start.row;
         next.col = start.col+1;
         if(findPath(map, m, n, next, dest, visited))
            return true;
      }
   }
```

미로에서 깊이우선탐색을 이용한 길찾기 - C++

```
    if(start.row < m-1) {  // 아래쪽
        if (map[start.row+1][start.col] == 0 and not visited[start.row+1][start.col]) {
            next.row = start.row + 1;
            next.col = start.col;
            if(findPath(map, m, n, next, dest, visited))
                return true;
        }
    }
    if(start.col > 0){ // 왼쪽
        if (map[start.row][start.col-1] == 0 and not visited[start.row][start.col-1]) {
            next.row = start.row;
            next.col = start.col-1;
            if(findPath(map, m, n, next, dest, visited))
                return true;
        }
    }
  // 위쪽
    ....
    return false;
}
```

미로에서 깊이우선탐색을 이용한 길찾기 - 파이썬

```
class Location:
    def __init__(self, row = 0, col = 0):
        self.row = row
        self.col = col


def findPath(map, m, n, start, dest, visited):
    next = Location()
    visited[start.row][start.col] = True
#    print("(", start.row, ",", start.col, ")")
    if start.row == dest.row and start.col == dest.col:
        return True
    if start.col < n-1:  # 오른쪽
        if map[start.row][start.col+1] == 0 and not visited[start.row][start.col+1]:
            next.row = start.row
            next.col = start.col + 1
            if findPath(map, m, n, next, dest, visited):
                return True
```

미로에서 깊이우선탐색을 이용한 길찾기 - 파이썬

```
if start.row < m-1:  // 아래쪽
    if map[start.row+1][start.col] == 0 and not visited[start.row+1][start.col]:
        next.row = start.row + 1
        next.col = start.col
        if findPath(map, m, n, next, dest, visited):
            return True

if start.col > 0:  // 왼쪽
    if map[start.row][start.col-1] == 0 and not visited[start.row][start.col-1]:
        next.row = start.row
        next.col = start.col - 1
        if findPath(map, m, n, next, dest, visited):
            return True

if start.row > 0:  // 위쪽
    if map[start.row-1][start.col] == 0 and not visited[start.row-1][start.col]:
        next.row = start.row - 1
        next.col = start.col
        if findPath(map, m, n, next, dest, visited):
            return True;
return False;
```

깊이우선탐색 예 – 그래프

- 깊이우선탐색(DFS: Depth First Search) 골격

recursive version

Algorithm dfs(g, v) // g: 그래프, 시작정점이 v인 깊이우선탐색

v를 방문하고 이를 “방문되었다고” 표시한다.

// v를 방문할 때 필요한 작업 수행

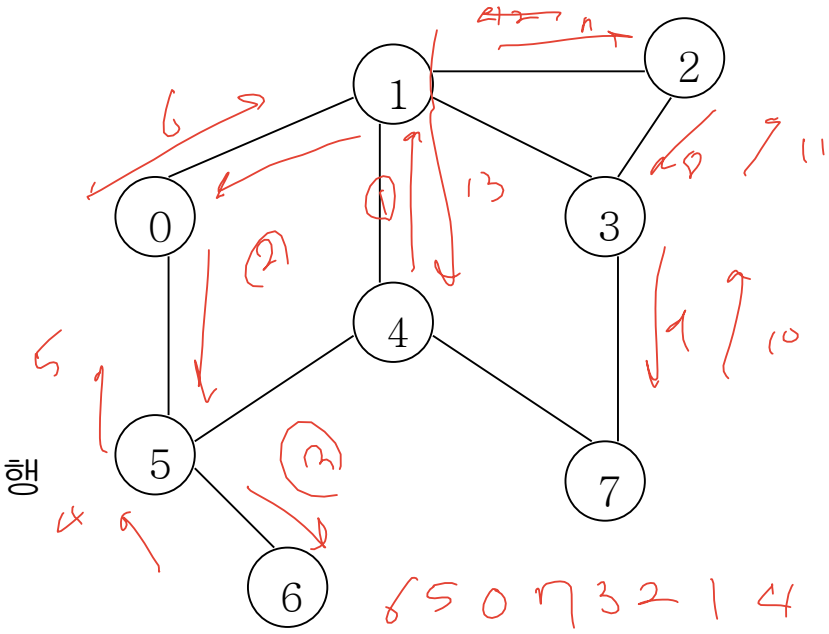
v와 인접한 각 정점 w에 대하여

만약 w가 방문되지 않았다면 dfs(g, w)

// w로 부터 backtrack 할 때 필요한 작업 수행

v가 “종료되었다”고 표시한다.

// v에서 시작하는 깊이우선탐색을 마친 후 필요한 작업 수행



출발정점: 0
깊이우선순서로 탐색시에 방문되는 정점들의 순서: 0, 1, 4, 5, 6, 7, 3, 2

파이썬 깊이우선탐색 – 인접행렬, 인접리스트1(연결구조), 인접리스트 2(list of lists)

```
def dfs1(adjMatrix, n, v, visited):
    visited[v] = True # color[v] = 'gray'
    print(v, end = ' ')
    for w in range(n):
        if visited[w] == False and adjMatrix[v][w] == 1:
            dfs1(adjMatrix, n, w, visited)
    # visited[v] = 'black'
```

```
class Node:
    def __init__(self,v):
        self.vertex = v
        self.next = None

def dfs2(adjList1, n, v, visited):
    visited[v] = True  # color[v] = 'gray'
    print(v, end = ' ')
    node = adjList1[v]
    while node is not None:
        w = node.vertex
        if visited[w] == False:
            dfs2(adjList1, n, w, visited)
        node = node.next
    # visited[v] = 'black'
```

```
def dfs3(adjList2, n, v, visited):
    visited[v] = True # color[v] = 'gray'
    print(v, end = ' ')

    for i in range(len(adjList2[v])):
        w = adjList2[v][i]
        if visited[w] == False:
            dfs3(adjList2, n, w, visited)
    # visited[v] = 'black'
```

```
n, m = input().split()
n, m = int(n), int(m)
adjMatrix = [[0 for _ in range(n)] for _ in range(n)]
adjList1 = [None for _ in range(n)]
adjList2 = [ [] for i in range(n)]
visited = [False for _ in range(n)] # 'white'

for i in range(m):
    u, v = input().split()
    u, v = int(u), int(v)
# adjacency matrix
    adjMatrix[u][v] = 1
    adjMatrix[v][u] = 1

# adjacency list1 (linked list)
    node = Node(v)
    node.next = adjList1[u]
    adjList1[u] = node
    node = Node(u)
    node.next = adjList1[v]
    adjList1[v] = node

# adjacenct list2
    adjList2[u].append(v)
    adjList2[v].append(u)
```

시간복잡도: dfs1 : $O(n^2)$

dfs2, dfs3: O(n+m)

n: vertex 개수, m: edge 개수

깊이우선탐색 응용(파이썬) - 정점 0에서 시작하는 깊이우선 탐색시 방문되는 정점들을 순서대로 출력

```
print("dfs at 0 using adjacency matrix")
dfs1(adjMatrix, n, 0, visited)
print()

print("dfs at 0 using adjacency list1")
for i in range(n):
    visited[i] = False # visited[i] = 'white'
dfs2(adjList1, n, 0, visited)

print()
print("dfs at 0 using adjacency list2")
for i in range(n):
    visited[i] = False # visited[i] = 'white'
dfs3(adjList2, n, 0, visited)
print()
```

파이썬 깊이우선탐색– 인접리스트(연결구조): 그래프를 클래스로 정의

```
class Node:
    def __init__(self, vertex):
        self.vertex = vertex
        self.link = None

class Graph:
    def __init__(self, size):
        self.adjList = [None]*size
        self.color = ['white']*size
        self.size = size

    def add_edge(self, v1, v2):
        new_node = Node(v2)
        new_node.link = self.adjList[v1]
        self.adjList[v1] = new_node
  # in case of undirected graph

        new_node = Node(v1)
        new_node.link = self.adjList[v2]
        self.adjList[v2] = new_node
```

```
    def dfs(self, v):
        self.color[v] = 'gray'; # v를 방문하였다고 표시함
        print(v, end = ' ')
        node = self.adjList[v];
        while node != None:
            w = node.vertex
            if self.color[w] == 'white':
                self.dfs(w)
            node = node.link;
        self.color[v] = 'black' # 종료된 상태


    def printGraph(self):
        for v in range(self.size):
            print(v, end = ': ')
            current = self.adjList[v]
            while current is not None:
                print(current.vertex, end = ' ')
                current = current.link
            print()
```

```
n = int(input())
m = int(input())
#m,m = [int (x) for x in input().split()]
g = Graph(n)
for i in range(m):
    v1, v2 = [int(x) for x in input().split()]
    g.add_edge(v1,v2)

g.printGraph()
g.dfs(0)
```

시간복잡도: O(n+m),

n: vertex 개수, m: edge 개수

C++ 깊이우선탐색 – 인접행렬

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
typedef enum Color_Type  {white, gray, black}
        ColorType;

// 그래프 표현: 인접행렬

void dfs(int** adjMatrix, int n,  int v, ColorType * color)
{
   color[v] = gray; // visited[v] = 1; // v를 방문 표시
   printf("%d ", v);
   for (int w = 0; w < n; w++) {
      if (adjMatrix[v][w] == 1
          && color[w] == white)
         dfs(adjMatrix, n, w, color);
   }
 //  color[v] = black;  // 종료된 상태
}
```

시간복잡도: $O(n^2)$

```
typedef enum Color_Type  {white, gray, black} ColorType;

// #define INFINITY 1000000000

int main()

{  int** adjMatrix;
   int n, m, v1, v2, wt; // wt : in the case of weigthed graph
   int i, j;
   cin >> n >> m; // 정점 수와 에지 수 입력
   adjMatrix = (int **)malloc(sizeof(int *)*n);
   for (i = 0; i < n; i++)
      adjMatrix[i] = (int*)malloc(sizeof(int)*n);
   for(i = 0; i < n; ++i)
     for(j = 0; j < n; j++)
        adjMatrix[i][j] = 0;
   for(i = 0; i < m; i++)   {
      cin >> v1 >>v2;  // 에지 (v1, v2) 입력
      adjMatrix[v1][v2] = 1;
/* in case of undirected graph */
      adjMatrix[v2][v1] = 1;
   }

   ColorType *color;

   color = new ColorType[n];

   for(int v = 0; v < n; v++)

      color[v] = white;  // 미방문 상태

  dfs(adjMatrix, n, 0, color);
```

C++ 깊이우선탐색 – 인접리스트 (C++ 연결구조)

```
#include <iostream>
#include <vector>
using namespace std;
typedef enum Color_Type  {white, gray, black} ColorType;
struct Node {
    int  vertex;
    // int weight; //
    struct Node *link;
};
```

```
// 그래프 표현: 인접리스트(직접 만든 연결리스트)
void dfs(Node** adjList, int n,  int v,
            vector<ColorType> &color)
{
  color[v] = gray; // v를 방문하였다고 표시함
  cout << v << " ";
  Node* ptr = adjList[v];
  while (ptr != NULL){
       int w = ptr->vertex;
       if (color[w] == white)
          dfs(adjList, n, w, color);
     ptr = ptr->link;
  }
  color[v] = black;  // 종료된 상태
}
```

시간복잡도: O(n+m),

n: vertex 개수, m: edge 개수

깊이우선탐색 응용 (C++) - 정점 0에서 시작하는 깊이우선 탐색시 방문되는 정점들을 순서대로 출력

```
int main()
{
   int n, m;
   int v1, v2;
   int i;
   cin >> n >> m;
   vector<ColorType> color(n);
   Node** adjList;
   adjList = new Node*[n];
   for(i = 0; i < n; i++)
        adjList[i] = NULL;
   for(i = 0; i < m; i++) {
        int v1, v2, wt;
        Node * ptr;
//          cin  >> v1 >> v2 >> wt;
        cin  >> v1 >> v2;
        ptr = new Node;
        ptr->vertex = v2;     // ptr->weight = wt;
        ptr->link = adjList[v1];
        adjList[v1] = ptr;
// * in case of undirected graph */
        ptr = new Node;
        ptr->vertex = v1;     // ptr->weight = wt;
        ptr->link = adjList[v2];
        adjList[v2] = ptr;
   }

   dfs(adjList, n, 0, color);
// free the dynamiclly allocated memory
   for (i = 0; i < n; i++){
        Node *ptr;
        while (adjList[i] != NULL){
           ptr = adjList[i];
           adjList[i] = ptr->link;
           delete ptr;
        }
   }

   delete [] adjList;
   return 0;
}
```

C++ 깊이우선탐색 – 인접리스트 (C++ vector of vectors)

```
#include <vector>
#include <iostream>
using namespace std;
typedef enum Color_Type {white, gray, black}
        ColorType;
// 그래프 표현: 인접리스트 (vector of vectors)
void dfs(vector<vector<int> > &adjList, int n, int v,
         vector<ColorType> &color)
{
  color[v] = gray; // v를 방문하였다고 표시함
  cout << v << " ";

  vector<int>::iterator itr;
  for (itr = adjList[v].begin(); itr != adjList[v].end();
       ++itr)
  {
     int w = *itr;
     if(color[w] == white)
       dfs(adjList, n, w, color);
  }

// for (int i = 0; i < adjList[v].size(); ++i) {
//    int w = adjList[v][i];
//    if(color[w] == white)
//      dfs(adjList, n, w, color);
// }
  color[v] = black;  // 종료된 상태
}
```

```
int main()
{
  int n, m;
  int v1, v2;
  cin >> n >>m;
  vector<vector<int> > adjList(n);
  for (int i = 0; i < m; ++i) {
     cin >> v1 >> v2;
     adjList[v1].push_back(v2);
  //  In case of undirected graph
     adjList[v2].push_back(v1);
  }
  vector<ColorType> color(n);
  for(int v = 0; v < n; v++)
    color[v] = white;  // 미방문 상태
  dfs(adjList, n, 0, color);
  return 0;
}
```

시간복잡도: O(n+m),

n: vertex 개수, m: edge 개수

C++ 깊이우선탐색 - 인접리스트 (C++ vector of lists)

```
#include <vector>
#include <list>
#include <iostream>
using namespace std;
typedef enum Color_Type  {white, gray, black}
      ColorType;
// 그래프 표현: 인접리스트 (vector of lists)
void dfs(vector< list< int > > &adjList, int n,  int v,
      vector<ColorType> &color)
{
  color[v] = gray; // v를 방문하였다고 표시함
  cout << v << " ";
  list< int >::iterator itr;
  for (itr = adjList[v].begin(); itr != adjList[v].end(); ++itr)
  {
       int w = *itr;
       if (color[w] == white)
         dfs(adjList, n, w, color);
  }
  color[v] = black;  // 종료된 상태
}
```

```
int main()
{
  int n, m;
  int v1, v2;
  cin >> n >>m;
  vector< list< int > > adjList(n);
  for (int i = 0; i < m; ++i) {
    cin >> v1 >> v2; // 에지 (v1, v2)
    adjList[v1].push_back(v2);
// In case of undirected graph
    adjList[v2].push_back(v1);
  }
  vector<ColorType> color(n);
  for(int v = 0; v < n; v++)
     color[v] = white;  // 미방문 상태
  dfs(adjList, n, 0, color);
  return 0;
}
```

시간복잡도: O(n+m),

n: vertex 개수, m: edge 개수

4.2. 너비우선탐색

- 시작정점으로부터 거리가 증가하는 순서대로 정점들을 방문함 (거리: 경로에 있는 에지들의 수) – 연결요소, 최단경로 찾기 등에 이용
- 너비우선탐색은 큐를 이용

// 너비우선 탐색을 하면서 너비우선탐색트리를 찾는 알고리즘 (v: 시작정점)

```
Algorithm bfs(g, n, v, parent)
// g는 그래프, n은 정점 수, v는 출발정점
// parent는 배열로서 parent[u]는 bfs 트리에서 u의
// 부모노드를 가리킴

// visited는 각 정점이 방문되었는지를 위한 크기 n인 배열
   for w = 0 to n-1
         visited[w] = false

   parent[v] = -1
   visited[v] = true   // v를 방문

// Q: 큐
   Q.enqueue(v)   // 큐에 v를 삽입
```

```
while(!Q.isEmpty())
   u = Q.dequeue() // 큐에서 삭제
   u 와 인접한 각 정점 w에 대하여,
      if(!visited[w])
         visited[w] = true
         Q.Enqueue(w)
         parent[w] = u // bfs 트리 에지
```

방문되어 있었다면 True

인접행렬 이용 시 성능이 떨어짐~

시간복잡도: $O(n^2)$, 인접리스트 사용시 $O(n+m)$

n: vertex 개수, m: edge 개수

4.2 너비우선탐색

- C++ STL queue 이용한 프로그램

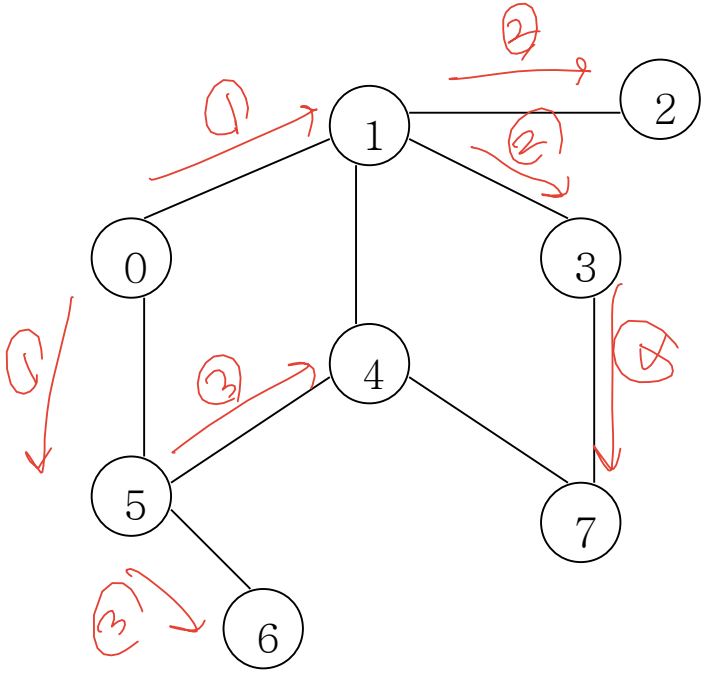
```
#include <queue>
void bfs(Graph  g, int n, int v, int parent[])
{
   queue<int> Q;
   ColorType *color;
   int w;
   color = new ColorType[n];
   for(w = 0; w < n; w++)
         color[w] = white;

   parent[v] = -1;
   color[v] = gray;
   Q.push(v); // 큐에 v를 넣는다
   while(!Q.empty()) {
      u = Q.front(); // 큐의 가장 앞에 있는 원소
      Q. pop();      // 큐의 가장 앞에 있는 원소 삭제
      u와 인접한 각 정점 w에 대하여 {
         if(color[w] == white) {
            color[w] = gray;
            Q.push(w); // 큐에 w를 넣는다
            parent[w] = u; // 에지 (v,w)를 처리
         }
      }
      color[u] = black;
   }
}
```

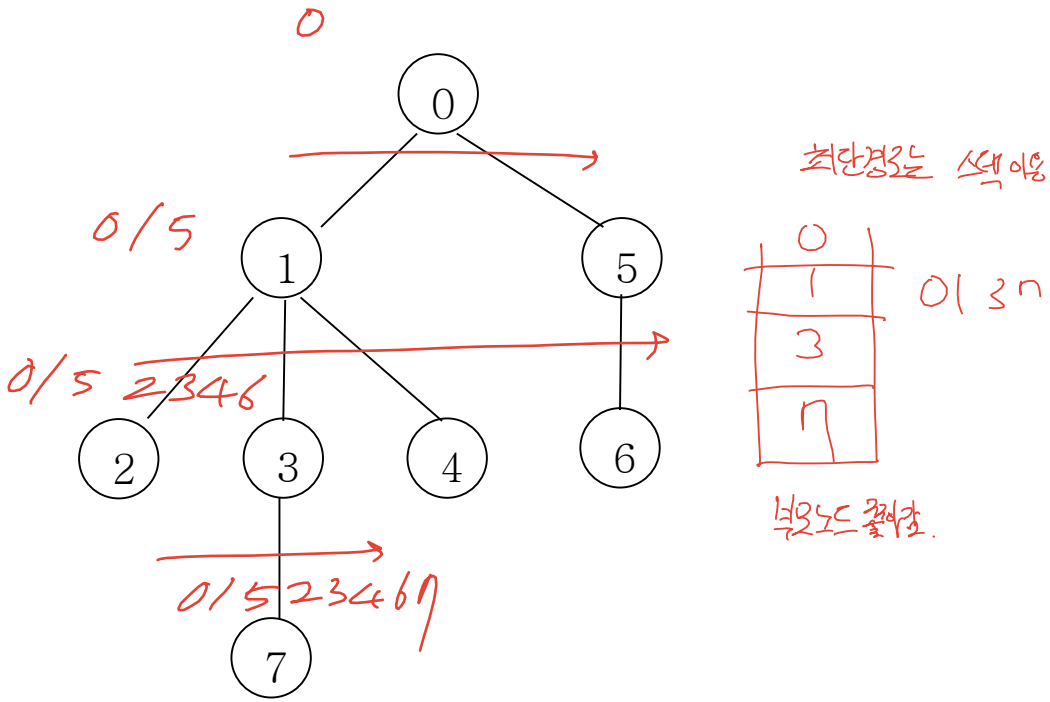
시간복잡도: $O(n^2)$, 인접리스트 사용시 $O(n+m)$

n: vertex 개수, m: edge 개수

너비우선탐색 예

- 너비우선탐색 예





BFS 트리

시작정점: 0
너비우선순서로 탐색시 방문되는 정점들의 순서: 0, 1, 5, 2, 3, 4, 6, 7

- Depth First Search는 스택을 이용하고 Breadth First Search는 큐를 이용한다.