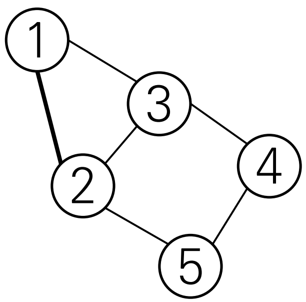
인접행렬을 이용한 그래프 저장

- 무향 그래프 : 간선의 방향이 없는 경우.

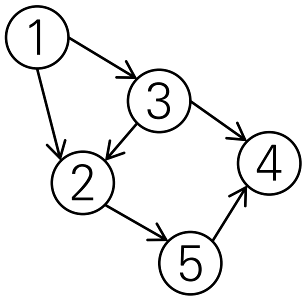


M	1	2	3	4	5
1	0	1	1	0	0
2	1				
3	1				
4	0				
5	0				

인접은 1,
아닌 경우 0

정점 수, 간선 수 간선의 정점 반복	5 6 1 2 1 3 2 3 3 4 2 5 5 4	read V, E for i : 1-〉E read n1, n2; M[n1][n2] = 1; M[n2][n1] = 1;

- 유향 그래프 : 간선에 방향이 있는 경우



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

도착

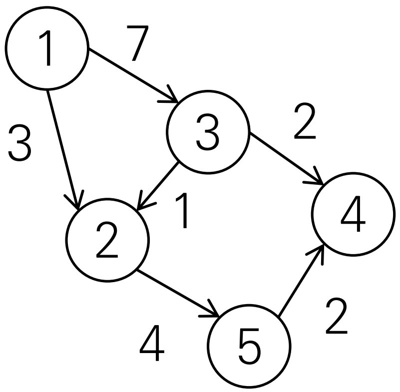
출발

출발 -> 도착 노드로 주어지는 경우

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

```
read n1, n2;
M[n1][n2] = 1;
```

▪ 가중치 그래프 : 비용이 있는 경우



	1	2	3	4	5
1	0	3	7	0	0
2	0				
3	0				
4	0				
5	0				

도착

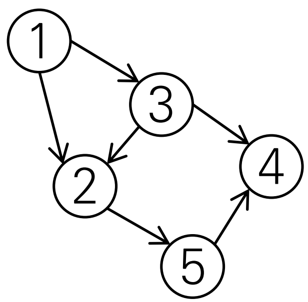
출발

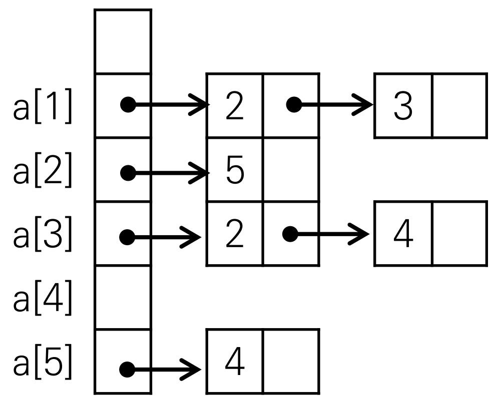
```
5 6
1 2 3
1 3 7
3 2 1
3 4 2
2 5 4
5 4 2
```

```
read n1, n2, w;
M[n1][n2] = w;
```

인접 리스트를 이용한 저장

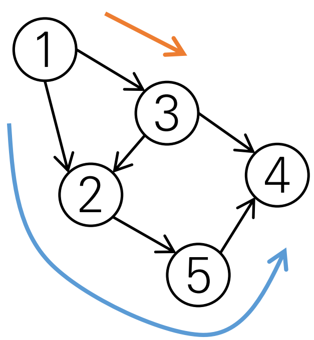
- 노드 개수가 많은 경우.





탐색

- 깊이 우선 탐색 (DFS)
  - 2개 이상의 선택이 가능할 때, 정해진 순서에 따라 다음 노드 선택.
  - 더 이상 갈 수 없으면 가장 가까운 이전 갈림길에서 다른 방향 선택.
    - 지나온 경로를 저장해야 함.

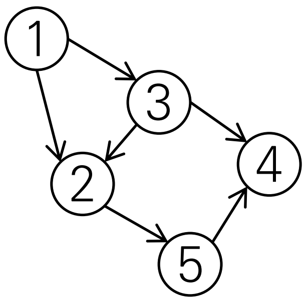


1 -> 2 -> 5 -> 4 -> 3

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

▪ 재귀를 사용한 DFS

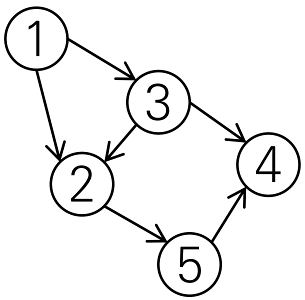
- 재귀의 각 단계가 방문중인 노드 번호를 저장.
- 방문한 노드에서 방문하지 않은 인접 노드 중 번호가 작은 곳으로 이동.




```
DFS(n)
    V[n] = 1                    // 방문 표시
    visit(n)                    // 노드에 대해 처리할 일
    for i : 1 -> N
        if( M[n][i] != 0 & V[i] == 0 )
            DFS(i)  // 인접하고 방문하지 않은 노드로 이동
```

✓반복 구조의 DFS

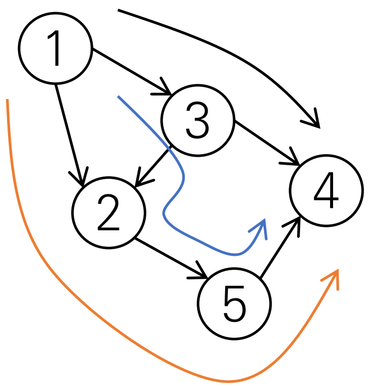
- 지나온 노드를 스택에 저장하거나, 방문하지 않고 남겨놓은 노드를 스택에 저장.




```
DFS(시작 노드 s)
    push(s)
    s 방문 표시
    while(stack_is_not_empty())
        n = pop( )
        visit(n)      // 노드에서 처리할 일
        n에 대해 i가 인접이고 아직 방문하지 않은 곳이면
            push(i)
            i 방문 표시
```

▪ DFS 응용

- 1에서 4번 노드에 도착할 수 있는 경로의 수 찾기.



가능한 경로

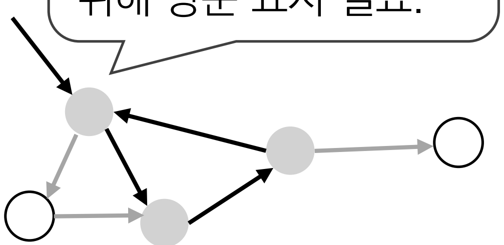
1 -> 2 -> 5 -> 4

1 -> 3 -> 2 -> 5 -> 4

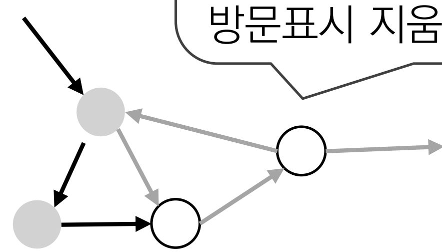
1 -> 3 -> 4

접근하는 경로가 다르면 중복을 허용해야 함.

되돌아 가는 것을 막기 위해 방문 표시 필요.



다른 경로에서의 접근을 위해 갈림길로 되돌아갈 때는 방문표시 지움.



- 1에서 4번 노드에 도착할 수 있는 경로의 수 찾기.

```
// n = 1, k = 4

DFS(n, k)
    if( n == k )
        cnt++;
    else
        V[n] = 1;                       // 방문 표시
        for i : 1 -> N
            if( M[n][i] == 1 && V[i] == 0 )
                DFS(i, k);              // 인접하고 방문하지 않은 노드로 이동
        V[n] = 0;                       //  방문 표시 삭제
```

for문 밖에서 처리

- 1에서 4번 노드에 도착할 수 있는 최단 거리 찾기.
  - 모든 경로를 찾는 것이 기본.
  - 지나온 간선의 수를 인자로 전달.

```
// 호출 조건 : n = 1, k = 4, e = 0, min = INF



DFS(n, k, e)
    if( n == k )
        if( min > e )
            min = e;
    else
        V[n] = 1;                       // 방문 표시
        for i : 1 -> N
            if( M[n][i] == 1 && V[i] == 0 )
                DFS(i, k, e+1);         // 인접하고 방문하지 않은 노드로 이동
        V[n] = 0;                       //  방문 표시 삭제
```

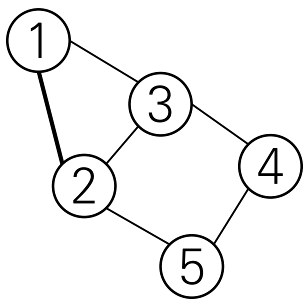
- 1에서 4번 노드에 도착할 수 있는 최단 거리 찾기.
  - 호출을 줄이려면 중단 조건 추가.

```
// 호출 조건 : n = 1, k = 4, e = 0, min = INF

DFS(n, k, e)
    if( n == k )
        if( min 〉 e )
            min = e;
    else if( e 〉= min )            // 현재까지 거리가 기존의 min보다 크면 다른 경로
        return;
    else
        V[n] = 1;                   // 방문 표시
        for i : 1 –〉 N
            if( M[n][i] == 1 && V[i] == 0 )
                DFS(i, k, e+1);     // 인접하고 방문하지 않은 노드로 이동
        V[n] = 0;                   //  방문 표시 삭제
```

연습

- 다음 그래프를 DFS로 탐색하고 방문 순서를 출력하시오.
  - 조건 : 1번에서 시작. 재귀 DFS 적용. 인접한 노드 중 작은 번호부터 방문.

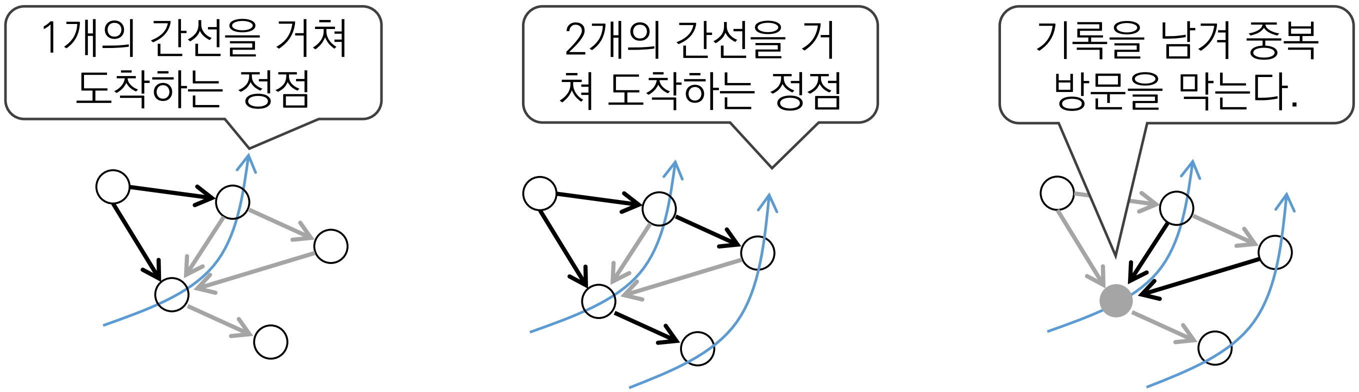


입력
노드 수, 간선 수
간선 정보
5 6
1 2 1 3 3 2 3 4 2 5 5 4

✓ 마지막 노드만 출력하려면?

▪ BFS (너비 우선 탐색)

- 시작 정점부터 거쳐가는 간선의 수가 같은 순서로 탐색하는 방식.

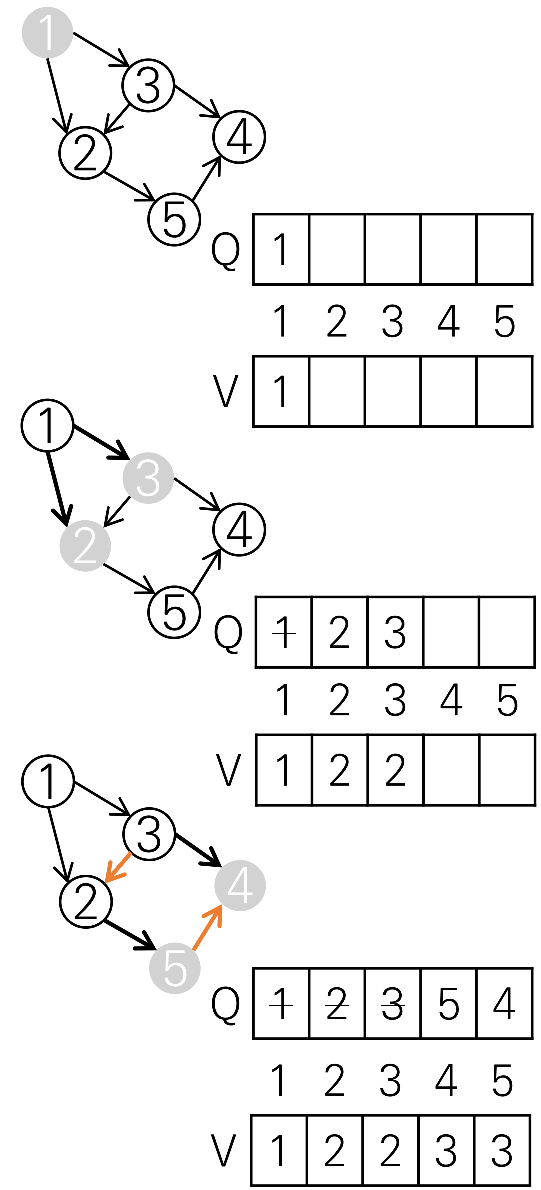


- 시작에서 n개의 간선을 지나 도착하는 정점의 인접 정점은 n+1개의 간선을 지나 도착하게 됨.
- 거리가 n인 정점들을 처리할 때 n+1인 인접 정점들을 저장함.
- 거리 n인 정점들을 처리하면, 저장해둔 n+1 정점들을 꺼내 처리함.

- BFS

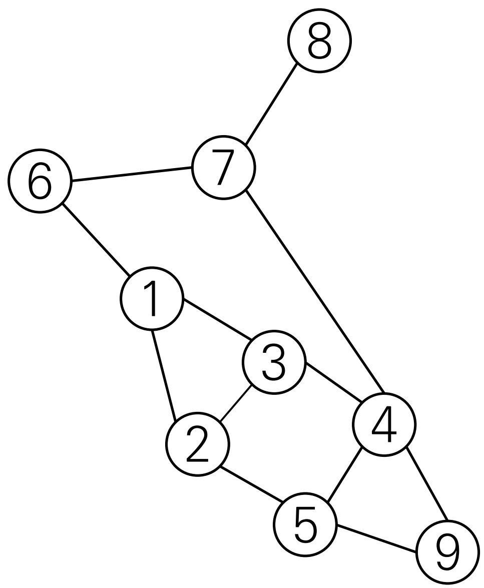
```
BFS(s)
    enQ(s)                      // 시작점 enqueue
    V[s] = 1                    // 방문 표시
    while( is_not_emptyQ())
        n = deQ()
        visit(n)                // 노드에 대해 처리할 일
        for i : 1 -> N
            // i가 인접이면서 방문안한 노드면
            if( A[n][i] == 1 && V[i] == 0 )
                enQ(i)
                V[i] = V[n] + 1
```

'V[i] = 1' 대신 'V[i] = V[n] + 1'로 표시하면 인접한 정점으로 부터의 거리를 알 수 있다.



연습

- 다음 그래프의 1번 노드에서 다른 노드까지 최소한의 간선을 거쳐 도착한다고 할 때, 지나는 간선 수의 총 합을 구하시오.



각 노드까지의 거리

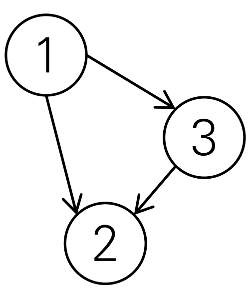
1	2	3	4	5	6	7	8	9
0	1	1	2	2	1	2	3	3

✓ 거리가 같은 노드는 최대 몇 개인가?

```
9 12
1 2 1 3 3 2 3 4 2 5 5 4 1 6 6 7 7 8 4 7 4 9 5 9
```

위상 정렬

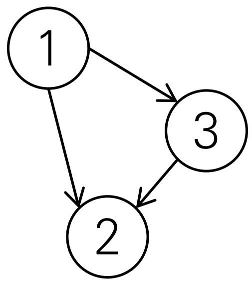
- 앞의 1, 3이 처리되어야 2번을 처리할 수 있는 경우.
  - 정점의 진입 차수를 활용.
  - 진입 차수가 0인 정점부터 시작.
  - 정점을 처리할 때 인접 정정에 처리되었음을 알림.
    - 인접 정점의 진입 차수를 하나 줄임.
    - 진입 차수가 0이 되면 다음 번에 처리할 차례가 됨.



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

	1	2	3
I	0	2	1

A : 인접행렬
I : 진입 차수



A : 인접행렬
I : 진입 차수

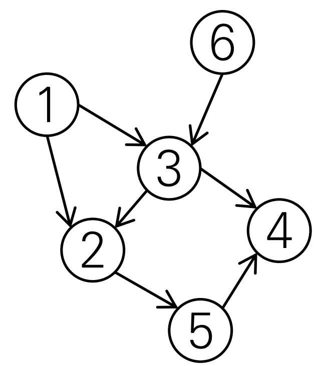
```
Sort( )
    for i : 0 -> N          // 진입차수가 0이면 enQ
        if( I[i] == 0 )
            enQ(i)
    while(is_not_emptyQ())
        n = deQ( )
        visit(n)                // 노드 n에서 해야할 일
        for i : 1 -> N
            if( A[n][i] == 1 )
                I[i]—    // n의 인접노드 진입차수 감소
                if( I[i] == 0 )  // 진입차수가 0이면 enQ
                    enQ(i)
```

Q	1								I	0	2	1
Q	~~1~~	3							I	0	2->1	1->0
Q	~~1~~	~~3~~	2						I	0	1->0	0
Q	~~1~~	~~3~~	~~2~~						I	0	0	0

연습

- 6명이 사람이 각자 갖고 있는 100원짜리 동전의 개수를 비교했더니 다음과 같은 조건이 되었다. 1번과 6번이 갖고 있는 금액이 100원 이었다면, 가장 많은 동전을 가진 사람은 최소한 몇 개의 동전을 갖고 있었는가?

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



- 1~V까지 각 사람을 노드로 표시.
- 노드 별 동전 수를 저장하는 배열 coin[] 선언, 0으로 초기화.
- visit(n)

노드 n으로 진입하는 모든 노드 i에 대해, coin[i]의 최대값 + 1을 coin[n]으로 정함. 진입하는 노드가 없는 경우 동전수는 1이 됨.

$coin[n] = \max_{1 \le i \le V} coin[i] + 1, adj[i][n]! = 0$이 있으면

$coin[n] = 1, adj[i][n]! = 0$이 없으면

```
visit(n) :
max = 0;
for i : 1-> V
    if( adj[i][n] != 0 )
        if( max < coin[i] )
            max = coin[i]
coin[n] = max+1;
```

✓ 이 문제의 경우,
진입차수 0인 노드 i는 coin[i] = 1,
while에서는 visit(n)대신 enQ(i) 후에
coin[i] = coin[n] + 1로 처리해도 됨.