그래프

- 객체와 객체들 사이의 연결 관계를 표현한 자료구조
- 정점(Vertex)의 집합과 정점을 연결하는 간선(Edge)의 집합으로 구성
  - Graph = (V, E) V : 정점들의 집합, E : 간선들의 집합

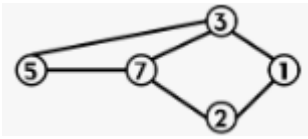
    | V | : 정점의 개수, | E | : 그래프에 포함된 간선의 개수

    그래프가 가지는 최대 간선의 수는 |V| * (|V| - 1) / 2 개
- 정점 하나에 연결된 간선의 수를 그 정점의 차수라고 한다
- 선형이나 트리로 표현하기 어려운 N : N 관계 원소들을 표현하기 좋음

종류

- 무향 그래프

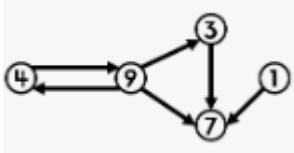
  : 서로 대칭적인 관계를 연결해 나타낸 그래프



- 유향 그래프

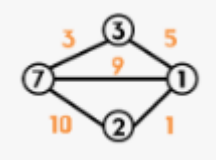
  : 방향성의 개념도 포함되어 간선을 화살표로 표현한 그래프

  서로 대칭적이지 않은 관계를 표현할 때 사용



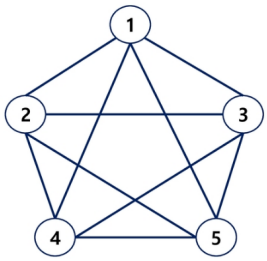
- 가중치 그래프

  : 유향 그래프에서 정점끼리 이동하는데 드는 비용을 간선에 부여한 그래프



- 완전 그래프

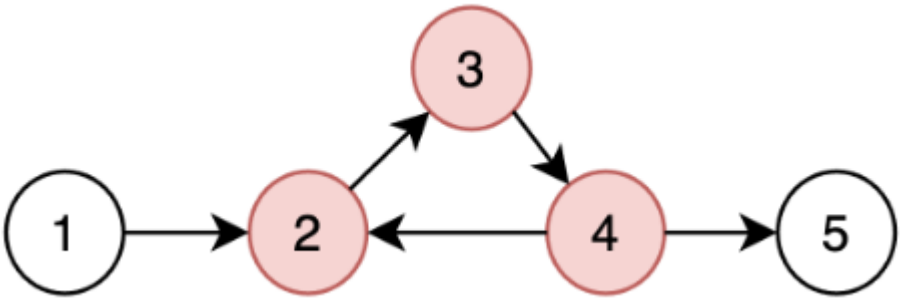
  : 정점에 대해 가능한 모든 간선을 가진 그래프



- 부분 그래프

  : 원래 그래프에서 일부의 정점이나 간선만 포함된 그래프

- 사이클 없는 유향 그래프 (DAG, Directed Acyclic Graph)



- 인접

  : 두 개의 정점 사이에 간선이 존재할 경우 두 정점은 인접해 있다고 한다

  a → b 로 이어져 있다고 할 때,

  b는 a의 인접 정점이지만, a는 b의 인접 정점이 아니다

- 경로

  : 간선들을 순서대로 나열한 것

  (0, 2), (2, 4), (4, 6) 등 해당 간선이 이어주는 정점을 나열하는 것으로 표현

  - 단순경로

    : 경로 중 하나의 정점은 최대 한 번만 지나는 경로

  - 순환경로

    : 시작한 정점에서 끝나는 경로, 또는 한 정점을 2번 이상 거치는 경로

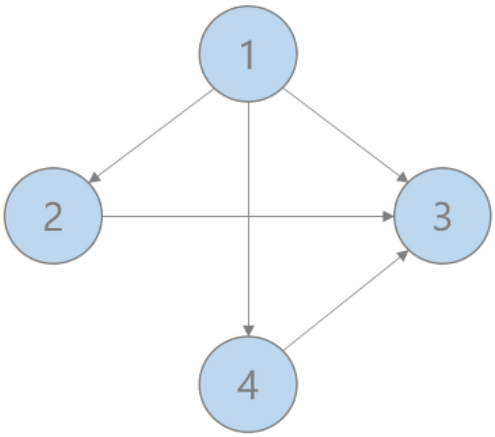
표현

- 인접 행렬

  : V * V 크기의 2차원 배열을 이용해서 간선 정보를 저장

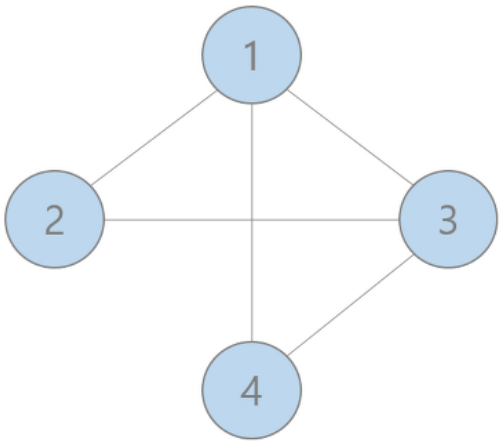
  그래프에 가중치가 없다면 boolean (0, 1) 형으로 연결 정보를 표현

가중치가 있다면, 각 간선의 가중치를 배열로 저장



0	1	1	1
0	0	1	0
0	0	0	0
0	0	1	0

[유향 그래프의 경우]



0	1	1	1
1	0	1	0
1	1	0	1
1	0	1	0

[무향 그래프의 경우]

무향 그래프의 경우, 한 간선이 있다면 그 반대로의 간선도 존재하므로

대각 성분 ([i][j], i = j)을 기준으로 대칭인 행렬이 나온다

인접행렬은 두 개의 노드가 연결되어있는지 확인하고 싶다면

행렬의 요소가 0인지 1인지만 확인하면 되기 때문에, O(1) 시간 복잡도를 가짐

- 노드는 적고 간선이 많은 밀집 그래프를 표현할 때 좋다

하지만 한 노드에 연결된 모든 노드에 방문해보고 싶을 경우,

총 O(V) 의 시간이 걸리게 된다

(극단적인 경우로, 노드의 개수가 1억개인데 간선이 2개라면

2개의 간선을 확인하기 위해 1억개의 노드 모두를 확인해야 한다)

```
node, edge = map(int, input().split())

adj_matrix = [[0 for _ in range(node)] for _ in range(node)]

for _ in range(edge):
    dep, arri = map(int, input().split())

    adj_matrix[dep - 1][arri - 1] = 1

    # 무향 그래프의 경우 대칭 형태를 가지므로 아래 코드 추가해 구현
    adj_matrix[arri - 1][dep - 1] = 1
```

무향 그래프에서는,

i번째 행의 합 = i번째 열의 합 = Vi의 차수

유향 그래프에서는

행 i의 합 = Vi의 진출 차수

열 i의 합 = Vi의 진입 차수

- 인접 리스트

  : 각 노드에 연결된 노드들을 원소로 갖는 리스트들의 배열

  무향 그래프의 경우에는 본인 노드 인덱스의 리스트 내에 서로를 원소로 가짐

  adj_list[i] : i 번째 노드에 연결된 노드들을 원소로 갖는 리스트

  인접 리스트는 인접 행렬과 달리 실제로 연결된 노드에 대한 정보만 저장

  - 모든 벡터들의 원소 개수 합이 간선의 개수와 동일

  하지만 노드 두 개의 연결 여부를 알고 싶을 때 인접 리스트를 사용하면

adj_list[i] 의 리스트를 순회하며 j 가 존재하는지 확인해야 한다 → O(V)

```
node, edge = map(int, input())

adj_list = [[] for _ in range(node)]

for _ in range(edge):
  dep, arri = map(int, input().split())
  agj_list[dep].append(arri)
```

그래프 탐색

- 깊이 우선 탐색 (Depth First Search)

  1, 시작 정점에서 갈 수 있는 방향 하나를 선택해 다음 정점으로 이동

  2. 선택된 정점에서 다시 위의 과정을 반복하며 갈 수 있는 경로가 없을 때까지 깊이를 우선적으로 탐색
     - 이미 방문한 곳은 다시 방문하지 않는다
  3. 더 이상 갈 정점이 없으면, 가장 최근 방문한 갈림길로 되돌아와 다른 방향으로 계속 과정 반복

  가장 마지막에 만났던 갈림길로 되돌아가 과정을 반복하는 것으로, 스택이나 재귀 호출 사용해 구현

  - 스택을 통한 구현

```
# graph는 주어질 그래프, root는 탐색을 시작할 정점
def dfs(graph, root):
  # 방문한 정점을 담을 리스트 visited 생성
  visited = []
  # 시작 정점을 요소로 추가하며 스택 생성
  stack = [root]

  # 스택이 비어있지 않는 한 반복
  while stack:
    # 현재 위치한 노드를 스택에서 빼며 반환
    node = stack.pop()

    # 현재 위치한 노드에 방문한 적이 없다면, 방문 기록에 추가
    # 그리고 해당 노드의 하위 노드를 스택에 추가
    # 얻고 싶은 결과물에 따라, 하위 노드 탐색 순서 조정을 위해 스택 배열 순서 바꿀 필요도 있음
```

```
    if node not in visited:
      visited.append(node)
      stack.extend(graph[node])
```

- 재귀를 통한 구현

```
def dfs(graph, root, visited=[]):
  # 방문 기록에 시작 정점 추가
  visited.append(root)

  # 시작 정점의 하위 노드들에 대해, 방문 기록에 없다면 추가하고
  # 하위 노드의 하위 노드들에 대해 다시 위 과정 반복

  for node in graph[root]:
    if node not in visited:
      dfs(graph, node, visited)

  # 탐색 종료 후 방문 기록 반환
  return visited
```

- dfs 이동 경로

```
# 목표에 맞는 이동 경로를 저장할 paths 리스트 추가
paths = []

# 그래프, 시작 정점, 목표 정점, 방문 기록을 담을 빈 리스트로 구성된 함수 선언
def dfs_paths(graph, start, end, visited=[]):
  # 이전 방문 정점들을 그대로 두며, 새로 도착한 정점을 추가
  visited = visited + [start]

  # 목표하는 정점에 도착할 경우, 경로 리스트에 추가
  if start == end:
    paths.append(visited)


  # 현재 노드의 하위 노드들 중, 방문 리스트에 없으면 함수 재귀 호출
  for node in graph[start]:
    if node not in visited:
      dfs_paths(graph, node, end, visited)

  # 반복 종료 후 경로 리스트 반환
  return paths
```

- 너비 우선 탐색 (Breadth First Search)

너비를 우선으로 하므로, 가장 얕은 노드부터 우선적으로 탐색한다

멀리 떨어진 노드는 가장 나중에 탐색되기 때문에, 두 노드 사이의 최단 경로를 찾을 때 유용

큐를 이용해 탐색 순서를 저장하고, 선입선출의 방식으로 탐색하게 된다

- 큐를 통한 구현

```
# 모듈로 큐를 구현하기 쉬운 데크 사용
from collections import deque

def bfs(graph, root):
  # root가 포함된 큐 생성
  queue = deque([root])
  # 방문한 노드들을 기록할 빈 리스트 생성
  visited = []

  # 큐가 비어있지 않는 한 반복
  while queue:
    node = queue.pop()

    # 방문 기록에 없다면 추가, 위에서 node를 pop으로 반환하므로,
    # extendleft 를 통해 새 요소를 역순으로 왼쪽에 추가
    # pop 을 수행할 시 먼저 들어와있던 요소를 우선적으로 제거하기 위함
    if node not in visited:
      visited.append(node)
      queue.extendleft(graph[node])

  # 탐색이 끝나면 반복 기록 반환
  return visited
```

트리

- 그래프와 같이 노드와 노드 사이를 연결하는 간선으로 구성되어 있지만,

  방향성을 지니며 두 노드 사이에 반드시 1개의 경로만을 가진다

  따라서 사이클이 존재하지 않으며 위 특성 때문에 ‘최소 연결 트리’ 로도 불림

- 부모 - 자식 관계가 성립되어 레벨 개념이 존재하며, 최상위 노드는 루트 노드

  노드가 N 개라면 간선은 N-1개

- 용어
    1. 루트 노드 : 부모 노드가 없는 최상위 노드
    2. 단말 노드 : 자식 노드가 없는 노드
    3. 크기 : 트리에 포함된 모든 노드의 개수
    4. 깊이 : 루트 노드로부터의 거리
    5. 높이 : 깊이 중 최댓값
    6. 차수 : 각 노드에서 나가는 간선 개수

이진트리

- 모든 노드가 최대 2개의 자식 노드를 가지고 있는 트리

- 트리 순회

  : 트리 자료구조에 포함된 노드들을 특정 방법으로 한 번씩 방문사는 방법

  1. 전위 순회 : 노드 - 왼쪽 자식 - 오른쪽 자식 순서로 방문
  2. 중위 순회 : 왼쪽 자식 - 노드 - 오른쪽 자식 순서로 방문
  3. 후위 순회 : 왼쪽 자식- 오른쪽 자식 - 노드 순서로 방문

- 이진 탐색 트리

  : 이진 탐색 트리는 모든 노드의 크기가 왼쪽 자식 < 부모 < 오른쪽 자식

  모든 노드의 데이터 값은 중복되는 값이 존재하지 않는다

  - 포화 이진 트리

    : 모든 레벨에서 노드들이 채워져 있는 트리

  - 완전 이진 트리

    : 마지막 레벨을 제외하고 노드가 모두 채워져 있는 트리로,

    왼쪽부터 차례로 채워나감

```
class node:
  def __init__(self, item=-1):
    # 루트 노드를 만들어주고, 왼쪽 / 오른쪽 자식이 들어갈 자리 마련
    self.data = item
    self.left = None
    self.right = None

  def insert(self, item):
    # 이진 탐색 트리의 조건에 따라, 부모 노드보다 입력 데이터가 크면 왼쪽, 아니면 오른쪽으로 지정
    # 이미 해당 자리가 차 있다면 재귀를 이용해 그 자식 노드 레벨에서 다시 반복
    # 값이 같은 노드가 있을 경우는 이미 존재하므로 오류 발생
    if self.data > item:
      if self.left:
        self.left.insert(item)
      else:
        self.left = node(item)
    elif self.data < item:
      if self.right:
        self.right.insert(item)
      else:
        self.right = node(item)
    else:
      raise KeyError("Item %d is already exist!" % item)

  def inorder(self):
    ret = []
    if self.left:
      ret += self.left.inorder()
    ret += [self.data]
    if self.right:
      ret += self.right.inorder()
    return ret

  def height(self):
    # 자식 노드의 노드 중 최댓값에 부모 노드의 높이 때문에 + 1
    l, r = 0, 0
    if self.left:
      l = self.left.height()
    if self.right:
      r = self.right.height()
    return max(l, r) + 1

  def size(self):
    # 양쪽 자식 노드 둘의 크기의 합 + 1
    l, r = 0, 0
    if self.left:
      l = self.left.size()
    if self.right:
      r = self.right.size()
    return l + r + 1

  def search(self, target, parent=None):
    # 특정 값이 있는지 없는지 검사하고, delete에서 사용을 위해 부모까지 반환한다.
    if self.data == target:
      return self, parent
    elif self.data > target:
```

```
      if self.left:
        return self.left.search(target, self)
    else:
      if self.right:
        return self.right.search(target, self)




class myBinarySearchTree:
  def __init__(self):
    self.root = None

  def insert(self, item):
    if self.root is None:
      self.root = node(item)
    else:
      self.root.insert(item)

  def inorder(self):
    return self.root.inorder()

  def height(self):
    return self.root.height()

  def size(self):
    return self.root.size()

  def search(self, target):
    return True if self.root.search(target) else False

  def delete(self, target):
    targetNode, targetParent = self.root.search(target)
    # 삭제할 node가 자식이 모두 없는 경우
    if targetNode.left is None and targetNode.right is None:
      if targetNode.data < targetParent.data:
        targetParent.left = None
      else:
        targetParent.right = None
      del targetNode

    # 삭제할 node가 오른쪽 자식만 있는 경우
    elif targetNode.left is None:
      if targetNode.data < targetParent.data:
        targetParent.left = targetNode.right
      else:
        targetParent.right = targetNode.right
      del targetNode

    # 삭제할 node가 왼쪽 자식만 있는 경우
    elif targetNode.right is None:
      if targetNode.data < targetParent.data:
        targetParent.left = targetNode.left
      else:
        targetParent.right = targetNode.left
      del targetNode

    # 삭제할 node가 자식이 둘 다 있는 경우
```

```
    else:
      haha = targetNode.left
      # 대신 삭제될 node(haha)의 부모가 원래 삭제될 node(targetNode)인 경우
      if haha.right is None:
        targetNode.data = haha.data
        targetNode.left = haha.left
        del haha

      # targetNode 보다 작은 가장 큰 값을 찾는 과정
      else:
        hehe = targetNode
        while haha.right:
          hehe = horse
          haha = haha.right

        targetNode.data = haha.data
        hehe.right = None
        del haha
```

- 이진 트리 순회 구현

```
# 트리를 다룰 빈 딕셔너리 생성
tree = {}

# 부모 노드를 key, 자식 노드를 value에
# 인덱스를 통해 value에 접근 가능
for _ in range(N):
  root, left, right = input().split()
  tree[root] = [left, right]


# 전위 순회  (루트 -> 왼쪽 -> 오른쪽)
# 첫 줄 아래에 탐색 멈출 부분을 위해 root가 아닌 조건 부여
def preorder(root):
  if root != ' ':
    print(root)
    # left 값을 다시 새로운 root로 지정해 반복
    preorder(tree[root][0])
    # right 값을 다시 새로운 root로 지정해 반복
    preorder(tree[root][1])

# 중위 순회  (왼쪽 -> 루트 -> 오른쪽)
def inorder(root):
  if root != ' ':
    inorder(tree[root][0])
    print(root)
    inorder(tree[root][1])

# 후위 순회  (왼쪽 -> 오른쪽 -> 루트)
def postorder(root):
  if root != ' ':
    postorder(tree[root][0])
    postorder(tree[root][1])
```

```
    print(root)
```