파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

그래프의 깊이우선탐색(Depth-First Search)
알고리즘을 학습하고 구현한다

Data Structures in Python Chapter 9

- Graph Introduction
- Graph Traversal – BFS
- **Graph Traversal – DFS**
- Topological Sort of DAG

Agenda

- Graph Traversals
  - BFS - Breadth First Search
  - **DFS - Depth First Search**

- Reference:
  - Problem Solving with Algorithms and Data Structures
  - Wikipedia: Depth-first search

Graph Traversals - Review

- Important graph-processing operations include:
  - Finding the shortest path to a given vertex (source) in a graph
  - Finding all of the items to which a given item is connected by paths

- **Breadth-First Search (BFS)**
  - Idea: Explore from a source in all possible directions, **layer by layer.**
  - It begins at the source vertex and explores its **neighbors first**.
  - Then, it explores their unexplored next neighbors, until it visits the target vertex or all.

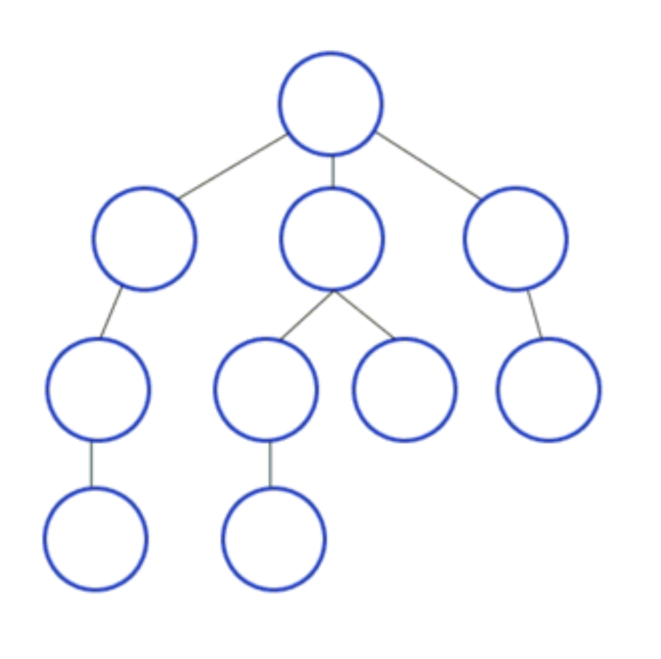
- **Depth-First Search (DFS)**
  - **Recursive Algorithm:**
    - Idea: Follow the first path you find as far as you can go.
    - Then, back up to last unexplored edge when you reach a dead end, then go as far you can.

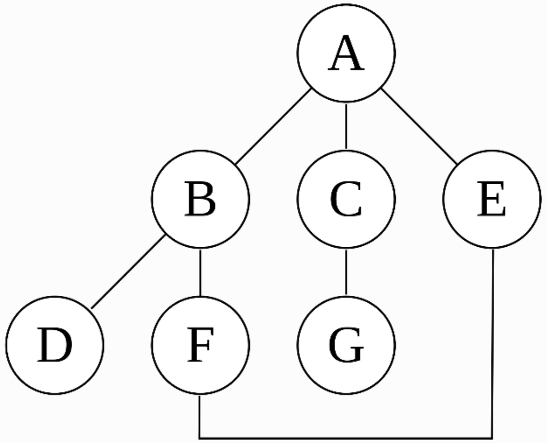
DFS Algorithm

- Application of DFS
  - For finding a path
  - For finding the strongly connected components of a graph
  - For detecting cycles
  - Topological Sorting
  - To test if a graph is bipartite

DFS Recursive Algorithm

- Recursive Algorithm:
  1. Mark the source vertex v as visited (or save it as a part of path).
  2. For every neighbor w of v if not visited,
     recursively call this function for that vertex w.
  3. Stop either when all vertices are visited, or the target vertex is found.



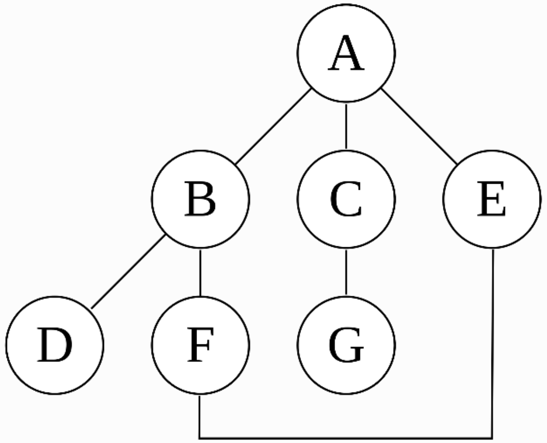


DFS Recursive Algorithm

- Recursive Algorithm:
  1. Mark the source vertex v as visited (or save it as a part of path).
  2. For every neighbor w of v if not visited,
     recursively call this function for that vertex w.
  3. Stop either when all vertices are visited, or the target vertex is found.

Pseudo code

```
def DFS(g, v):
    add v to path
    for each neighbor w of v
        if w not in path:
            DFS(g, w)
```



DFS Recursive Algorithm

- Recursive Algorithm:
  1. Mark the source vertex v as visited (or save it as a part of path).
  2. For every neighbor w of v if not visited,
     recursively call this function for that vertex w.
  3. Stop either when all vertices are visited, or the target vertex is found.

Pseudo code

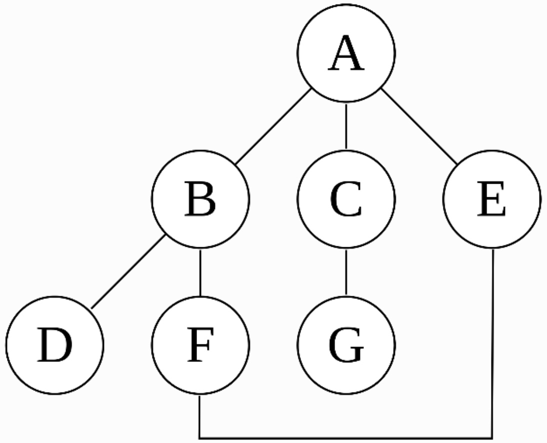
```
def DFS(g, v):
    add v to path
    for each neighbor w of v
        if w not in path:
            DFS(g, w)
```

abcdefg.txt
edge list

```
A B
A C
A E
B D
B F
C G
E F
```

Adjacency list

```
A: B C E
B: A D F
C: A G
E: A F
D: B
F: B E
G: C
```

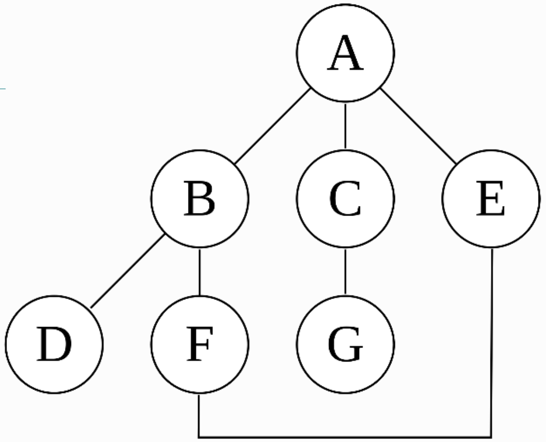


`recursive DFS: A,`

DFS Recursive Algorithm

abcdefg.txt
edge list

```
A B
A C
A E
B D
B F
C G
E F
```



Adjacency list

```
A: B C E
B: A D F
C: A G
E: A F
D: B
F: B E
G: C
```

Tracing recursive calls

```
DFS(A)
path[A]
A_w[B, C, E]
    DFS(B)
    path[A, B]
    B_w[A, D, F]
        DFS(D)
        path[A, B, D]
        D_w[B]
        DFS(F)
        path[A, B, D, F]
        F_w[B, E]
            DFS(E)
            path[A, B, D, F, E]
            E_w[A, F]
    DFS(C)
    path[A, B, D, F, E, C]
    C_w[A, G]
        DFS(G)
        path[A, B, D, F, E, C, G]
        G_w[C]
```

X_w […] indicates X's neighbor vertices

Pseudo code

```
def DFS(g, v):
    add v to path
    for each neighbor w of v
        if w not in path:
            DFS(g, w)
```

DFS Class

Pseudocode

```
def DFS(g, v):
    add v to path
    for each neighbor w of v
        if w not in path:
            DFS(g, w)
```

```
class DFS:
    def __init__(self, g, s):
        self._path = []
        self.dfs(g, s)

    def dfs(self, g, v):         # recursive DFS
        if g.countV() == len(self._path): return

        self._path.append(v)
        for w in g.neighbors(v):
            if w not in self._path:
                self.dfs(g, w)
```

DFS Class

Pseudocode

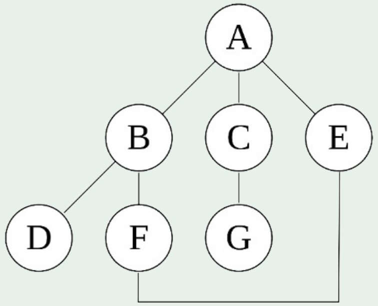
```
def DFS(g, v):
    add v to path
    for each neighbor w of v
        if w not in path:
            DFS(g, w)
```

```
class DFS:
    def __init__(self, g, s):
        self._path = []
        self.dfs(g, s)

    def dfs(self, g, v):          # recursive DFS
        if g.countV() == len(self._path): return

        self._path.append(v)
        for w in g.neighbors(v):
            if w not in self._path:
                self.dfs(g, w)

if __name__ == '__main__':
    g = Graph("abcdefg.txt")
    print(g)
    dfs = DFS(g, 'A')
    print(dfs._path)
```

abcdefg.txt
edge list

```
A B
A C
A E
B D
B F
C G
E F
```

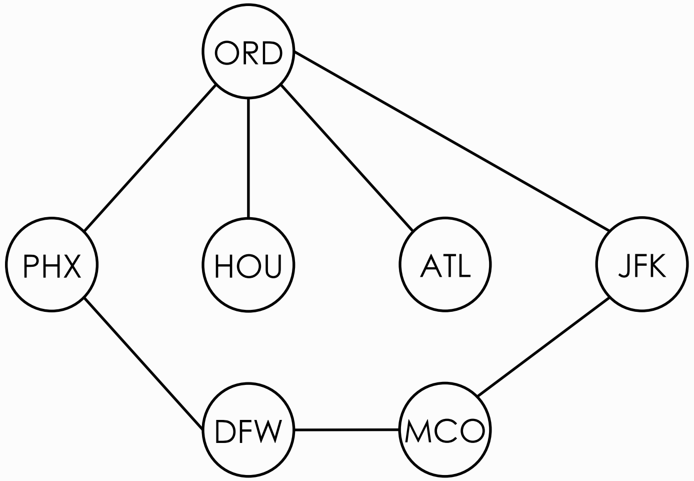
```
A: B C E
B: A D F
C: A G
E: A F
D: B
F: B E
G: C

['A', 'B', 'D', 'F', 'E', 'C', 'G']
```



DFS Example:

route7.txt
edge list

```
ORD PHX
MCO DFW
ORD HOU
JFK MCO
PHX MCO
ORD ATL
ORD JFK
PHX DFW
```



Adjacency list

```
ORD: PHX HOU ATL JFK
PHX: ORD DFW
MCO: DFW JFK
DFW: MCO PHX
HOU: ORD
JFK: MCO ORD
ATL: ORD
```

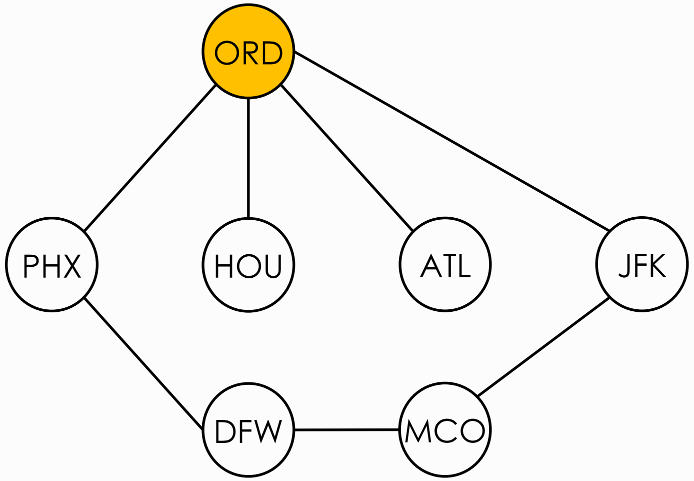
DFS Class

```
class DFS:
    def __init__(self, g, s):
        self._path = []
        self.dfs(g, s)

    def dfs(self, g, v):        # recursive DFS
        if g.countV() == len(self._path): return

        self._path.append(v)
        for w in g.neighbors(v):
            if w not in self._path:
                self.dfs(g, w)

if __name__ == '__main__':
    g = Graph("route7.txt")
    dfs = DFS(g, "ORD")
    print(dfs._path)
```

DFS Example:

route7.txt
edge list

```
ORD PHX
MCO DFW
ORD HOU
JFK MCO
PHX MCO
ORD ATL
ORD JFK
PHX DFW
```



Adjacency list

```
ORD: PHX HOU ATL JFK
PHX: ORD DFW
MCO: DFW JFK
DFW: MCO PHX
HOU: ORD
JFK: MCO ORD
ATL: ORD
```

DFS Class

```
class DFS:
    def __init__(self, g, s):
        self._path = []
        self.dfs(g, s)

    def dfs(self, g, v):        # recursive DFS
        if g.countV() == len(self._path): return

        self._path.append(v)
        for w in g.neighbors(v):
            if w not in self._path:
                self.dfs(g, w)

if __name__ == '__main__':
    g = Graph("route7.txt")
    dfs = DFS(g, "ORD")
    print(dfs._path)
```

Sample Run: (ORD – Recursive DFS)

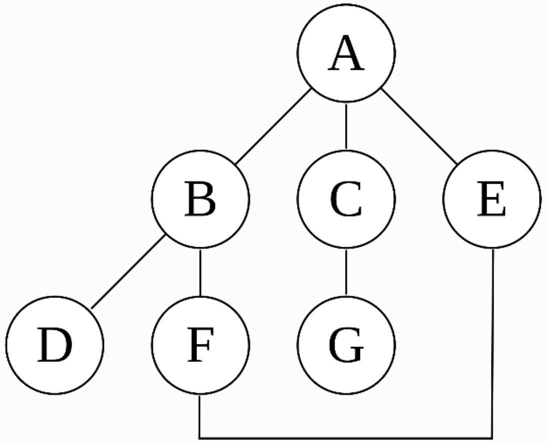
```
['ORD', 'PHX', 'DFW', 'MCO', 'JFK', 'HOU', 'ATL']
```

DFS Iterative Algorithm

- Iterative Algorithm:
  1. Mark the source vertex v to the **stack**.
  2. Repeat if stack is not empty
     1. Pop stack for v and add v to path (as visited)
     2. For each neighbor w of v, push w to stack if not in path.

Pseudo code

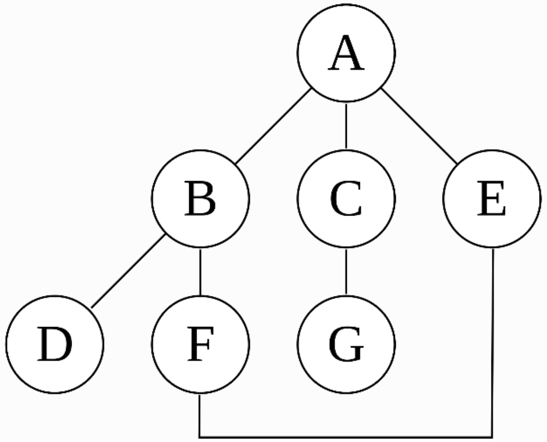
```
def IDFS(g, v):
    push v to stack
    while stack:
        v = pop stack & add v to path
        for each neighbor w of v:
            if w not in path
                push w stack
```



DFS Iterative Algorithm

Adjacency list

```
A: B C E
B: A D F
C: A G
E: A F
D: B
F: B E
G: C
```



Pseudo code

```
def IDFS(g, v):
    push v to stack
    while stack:
        v = pop stack & add v to path
        for each neighbor w of v:
            if w not in path
                push w stack
```

path visited

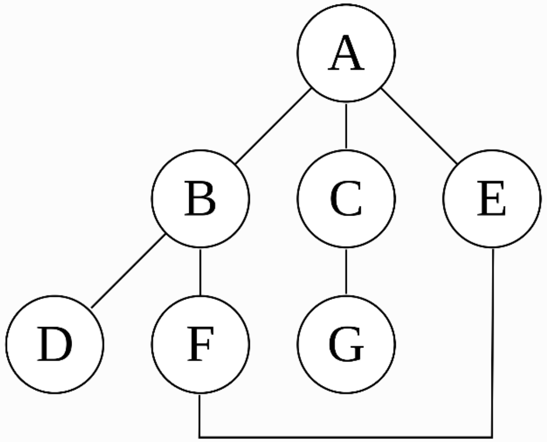
```
stack: ['A']              path: []          pop stack  A  and add A to path
stack: []                 path: ['A']       v = A, w = B, C, E
```

```
recursive DFS: A, B, D, F, E, C, G
iterative DFS: A, E, F, B, D, C, G
```

DFS Iterative Algorithm

Adjacency list

```
A: B C E
B: A D F
C: A G
E: A F
D: B
F: B E
G: C
```



push neighbors B, C, E

stack top

path visited

```
stack: ['A']               path: []
stack: ['B', 'C', 'E']     path: ['A']
```

pop stack A and add A to path

v = A, w = B, C, E

Pseudo code

```
def IDFS(g, v):
    push v to stack
    while stack:
        v = pop stack & add v to path
        for each neighbor w of v:
            if w not in path
                push w stack
```

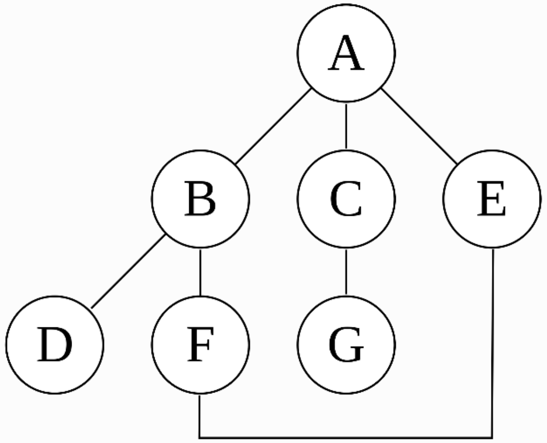
recursive DFS: A, B, D, F, E, C, G

iterative DFS: **A, E, F, B, D, C, G**

DFS Iterative Algorithm

Adjacency list

```
A: B C E
B: A D F
C: A G
E: A F
D: B
F: B E
G: C
```



Pseudo code

```
def IDFS(g, v):
    push v to stack
    while stack:
        v = pop stack & add v to path
        for each neighbor w of v:
            if w not in path
                push w stack
```

push neighbors E, F

stack top *path visited*

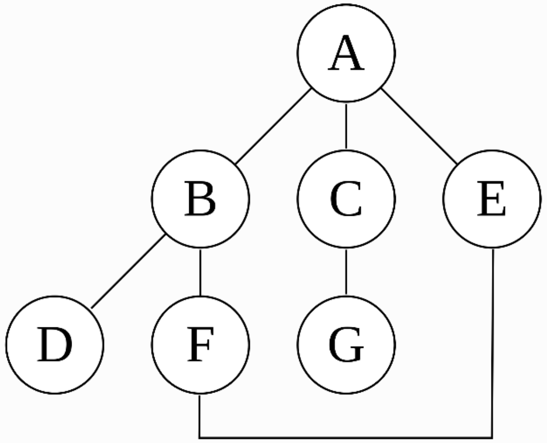
```
stack: ['A']              path: []            pop stack  A  and add A to path
stack: ['B', 'C', 'E']    path: ['A']         pop stack E and add E to path
stack: ['B',  'C', 'F']   path: ['A', 'E']    v = E, w = A, F
```

```
recursive DFS: A, B, D, F, E, C, G
iterative DFS: A, E, F, B, D, C, G
```

DFS Iterative Algorithm

Adjacency list

```
A: B C E
B: A D F
C: A G
E: A F
D: B
F: B E
G: C
```



Pseudo code

```
def IDFS(g, v):
    push v to stack
    while stack:
        v = pop stack & add v to path
        for each neighbor w of v:
            if w not in path
                push w stack
```

stack top   *path visited*

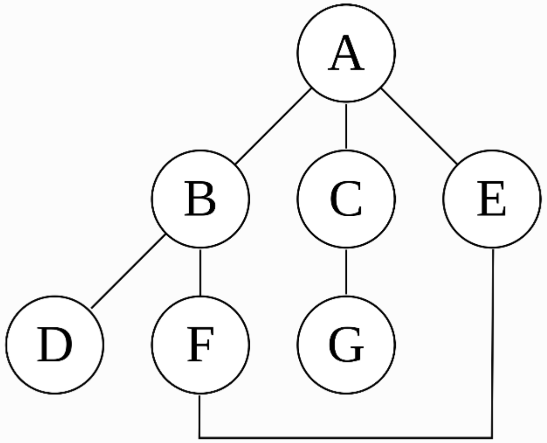
```
stack: ['A']             path: []              pop stack  A  and add A to path
stack: ['B', 'C', 'E']   path: ['A']           pop stack  E  and add E to path
stack: ['B', 'C', 'F']   path: ['A', 'E']
stack: ['B', 'C', 'B']   path: ['A', 'E', 'F']
```

```
recursive DFS: A, B, D, F, E, C, G
iterative DFS: A, E, F, B, D, C, G
```

DFS Iterative Algorithm

Adjacency list

```
A: B C E
B: A D F
C: A G
E: A F
D: B
F: B E
G: C
```



Pseudo code

```
def IDFS(g, v):
    push v to stack
    while stack:
        v = pop stack & add v to path
        for each neighbor w of v:
            if w not in path
                push w stack
```

stack top *path visited*

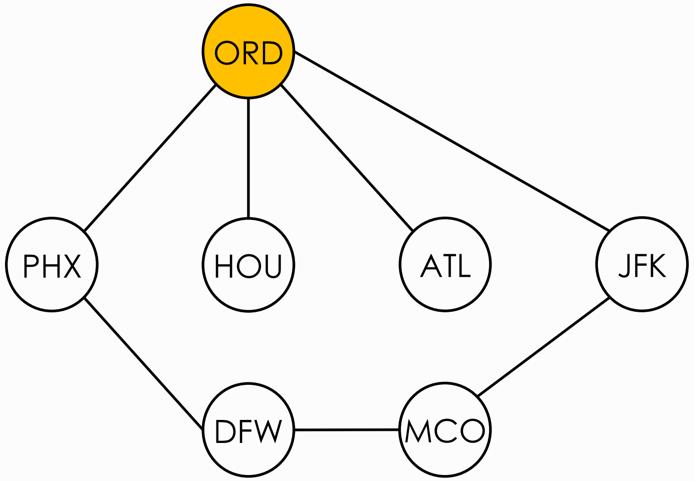
```
stack: ['A']             path: []          pop stack  A  and add A to path
stack: ['B', 'C', 'E']   path: ['A']       pop stack  E  and add E to path
stack: ['B', 'C', 'F']   path: ['A', 'E']
stack: ['B', 'C', 'B']   path: ['A', 'E', 'F']
stack: ['B', 'C', 'D']   path: ['A', 'E', 'F', 'B']
stack: ['B', 'C']        path: ['A', 'E', 'F', 'B', 'D']
stack: ['B', 'G']        path: ['A', 'E', 'F', 'B', 'D', 'C']
stack: ['B']             path: ['A', 'E', 'F', 'B', 'D', 'C', 'G']
```

recursive DFS: A, B, D, F, E, C, G
iterative DFS: **A, E, F, B, D, C, G**

IDFS Class Exercise

route7.txt
edge list

```
ORD PHX
MCO DFW
ORD HOU
JFK MCO
PHX MCO
ORD ATL
ORD JFK
PHX DFW
```



Adjacency list

```
ORD: PHX HOU ATL JFK
PHX: ORD DFW
MCO: DFW JFK
DFW: MCO PHX
HOU: ORD
JFK: MCO ORD
ATL: ORD
```

IDFS Class

```
class IDFS(DFS):
    def dfs(self, graph, v):  # iterative DFS
        stack = [v]           # use list as a stack

        while stack:


            # your code here







if __name__ == '__main__':
    g = Graph("route7.txt")
    dfs = IDFS(g, "ORD")
    print(dfs._path)
```

Sample Run: (ORD – Iterative DFS)

```
['ORD', 'JFK', 'MCO', 'DFW', 'PHX', 'ATL', 'HOU']
```

Summary

- Depth First Search (DFS) is another algorithm for traversing or searching for a graph. There are two ways to implement DFS algorithm with **iterative** and **recursive** approaches.
- **Time Complexity:**
  Since all the vertices are visited, the time complexity for DFS on a graph is $\boldsymbol{O(V\ +\ E)}$, where V is the number of vertices and E is the number of edges.

학습 정리

1) 깊이우선탐색(Depth-First Search) 은 재귀 혹은 스택을 이용하는 알고리즘이 있다. 스택을 이용한 알고리즘은

   Step 1: 탐색 시작 노드 v를 스택에 삽입하고 방문 처리를 한다

   Step 2: 스택 최상단 노드 v에서 방문하지 않은 인접 노드 w가 있으면,
   w를 스택에 넣고 방문 처리한다
   방문하지 않은 인접 노드가 없으면, 스택에서 최상단 노드 v를 꺼낸다

   Step 3: Step 2의 과정을 더 이상 수행할 수 없을 때까지 반복한다

2) DFS로 경로를 찾을 수 있다

3) DFS의 시간 복잡도는 O(V + E)이다

파이썬으로 배우는 데이터 구조

수고했습니다
곧 다음 시간에
다시 뵙겠습니다