

Stack: 선입후출 FILO

리스트 append, pop 함수

## 스택 구현 예제 (Python)

```
stack = []

# 삽입(5) - 삽입(2) - 삽입(3) - 삽입(7) - 삭제() - 삽입(1) - 삽입(4) - 삭제()
stack.append(5)
stack.append(2)
stack.append(3)
stack.append(7)
stack.pop()
stack.append(1)
stack.append(4)
stack.pop()

print(stack[::-1]) # 최상단 원소부터 출력
print(stack) # 최하단 원소부터 출력
```

Queue: 선입선출/ 은행 업무/ 대기열

리스트로 구현 가능하지만 시간복잡도에서 유리 collections의 deque

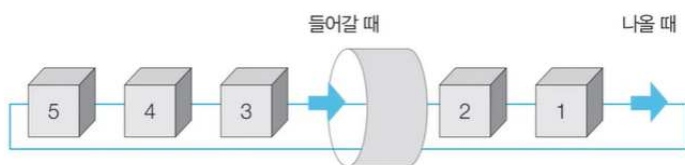
## 큐 구현 예제 (Python)

```
from collections import deque

# 큐(Queue) 구현을 위해 deque 라이브러리 사용
queue = deque()

# 삽입(5) - 삽입(2) - 삽입(3) - 삽입(7) - 삭제() - 삽입(1) - 삽입(4) - 삭제()
queue.append(5)
queue.append(2)
queue.append(3)
queue.append(7)
queue.popleft()
queue.append(1)
queue.append(4)
queue.popleft()

print(queue) # 먼저 들어온 순서대로 출력
queue.reverse() # 역순으로 바꾸기
print(queue) # 나중에 들어온 원소부터 출력
```



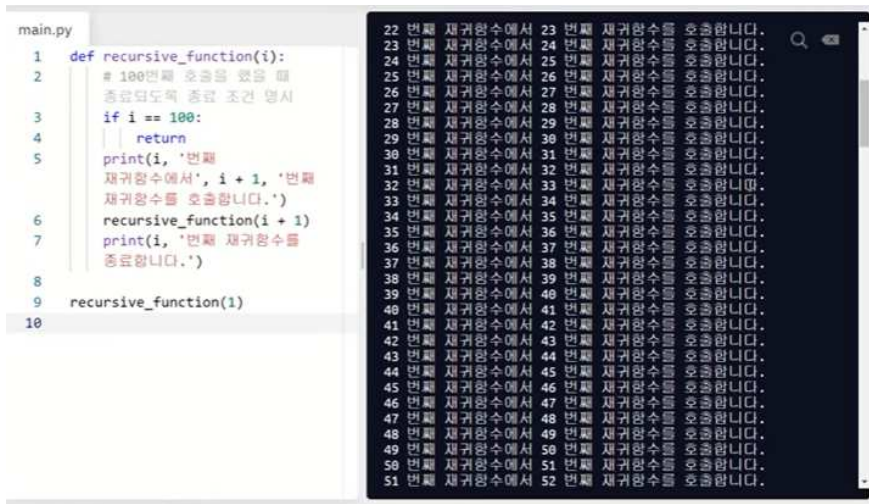
재귀함수

반복문 없이 재귀함수 만들 수 있음

## 재귀 함수의 종료 조건

- 재귀 함수를 문제 풀이에서 사용할 때는 재귀 함수의 종료 조건을 반드시 명시해야 합니다.
- 종료 조건을 제대로 명시하지 않으면 함수가 무한히 호출될 수 있습니다.
  - 종료 조건을 포함한 재귀 함수 예제

```
def recursive_function(i):  
    # 100번째 호출을 했을 때 종료되도록 종료 조건 명시  
    if i == 100:  
        return  
    print(i, '번째 재귀함수에서', i + 1, '번째 재귀함수를 호출합니다.')  
    recursive_function(i + 1)  
    print(i, '번째 재귀함수를 종료합니다.')  
  
recursive_function(1)
```



```
main.py  
1 def recursive_function(i):  
2     # 100번째 호출을 했을 때  
3     종료되도록 종료 조건 명시  
4     if i == 100:  
5         return  
6     print(i, '번째  
7     재귀함수에서', i + 1, '번째  
8     재귀함수를 호출합니다.')  
9     recursive_function(i + 1)  
10    print(i, '번째 재귀함수를  
11    종료합니다.')  
12  
13    recursive_function(1)
```

```
22 번째 재귀함수에서 23 번째 재귀함수를 호출합니다.  
23 번째 재귀함수에서 24 번째 재귀함수를 호출합니다.  
24 번째 재귀함수에서 25 번째 재귀함수를 호출합니다.  
25 번째 재귀함수에서 26 번째 재귀함수를 호출합니다.  
26 번째 재귀함수에서 27 번째 재귀함수를 호출합니다.  
27 번째 재귀함수에서 28 번째 재귀함수를 호출합니다.  
28 번째 재귀함수에서 29 번째 재귀함수를 호출합니다.  
29 번째 재귀함수에서 30 번째 재귀함수를 호출합니다.  
30 번째 재귀함수에서 31 번째 재귀함수를 호출합니다.  
31 번째 재귀함수에서 32 번째 재귀함수를 호출합니다.  
32 번째 재귀함수에서 33 번째 재귀함수를 호출합니다.  
33 번째 재귀함수에서 34 번째 재귀함수를 호출합니다.  
34 번째 재귀함수에서 35 번째 재귀함수를 호출합니다.  
35 번째 재귀함수에서 36 번째 재귀함수를 호출합니다.  
36 번째 재귀함수에서 37 번째 재귀함수를 호출합니다.  
37 번째 재귀함수에서 38 번째 재귀함수를 호출합니다.  
38 번째 재귀함수에서 39 번째 재귀함수를 호출합니다.  
39 번째 재귀함수에서 40 번째 재귀함수를 호출합니다.  
40 번째 재귀함수에서 41 번째 재귀함수를 호출합니다.  
41 번째 재귀함수에서 42 번째 재귀함수를 호출합니다.  
42 번째 재귀함수에서 43 번째 재귀함수를 호출합니다.  
43 번째 재귀함수에서 44 번째 재귀함수를 호출합니다.  
44 번째 재귀함수에서 45 번째 재귀함수를 호출합니다.  
45 번째 재귀함수에서 46 번째 재귀함수를 호출합니다.  
46 번째 재귀함수에서 47 번째 재귀함수를 호출합니다.  
47 번째 재귀함수에서 48 번째 재귀함수를 호출합니다.  
48 번째 재귀함수에서 49 번째 재귀함수를 호출합니다.  
49 번째 재귀함수에서 50 번째 재귀함수를 호출합니다.  
50 번째 재귀함수에서 51 번째 재귀함수를 호출합니다.  
51 번째 재귀함수에서 52 번째 재귀함수를 호출합니다.
```

스택에 데이터를 넣었다가 꺼내는 것과 같이 구현됨

재귀함수로 최대공약수 구현

A와 B의 최대공약수는 A와 A%B 의 최대공약수와 동일함

## 최대공약수 계산 (유클리드 호제법) 예제

- 두 개의 자연수에 대한 최대공약수를 구하는 대표적인 알고리즘으로는 유클리드 호제법이 있습니다.
- 유클리드 호제법
  - 두 자연수 A, B에 대하여 ( $A > B$ ) A를 B로 나눈 나머지를 R이라고 합시다.
  - 이때 A와 B의 최대공약수는 B와 R의 최대공약수와 같습니다.
- 유클리드 호제법의 아이디어를 그대로 재귀 함수로 작성할 수 있습니다.
  - 예시:  $\text{GCD}(192, 162)$

| 단계 | A   | B   |
|----|-----|-----|
| 1  | 192 | 162 |
| 2  | 162 | 30  |
| 3  | 30  | 12  |
| 4  | 12  | 6   |

컴퓨터가 함수를 연속적으로 호출하면 컴퓨터 메모리 내부의 스택 프레임에 쌓입니다.

- 그래서 스택을 사용해야 할 때 구현상 스택 라이브러리 대신에 재귀 함수를 이용하는 경우가 많습니다.

DFS(깊이 우선) >> 스택, 재귀함수

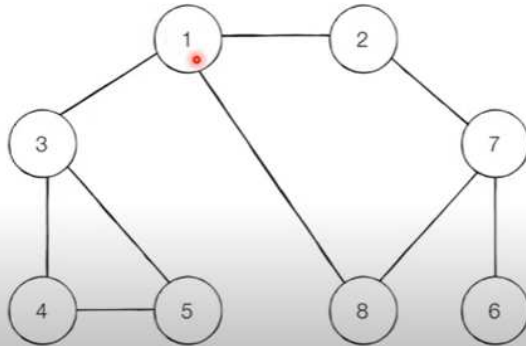
DFS는 스택 자료구조(혹은 재귀 함수)를 이용하며, 구체적인 동작 과정은 다음과 같습니다.

1. 탐색 시작 노드를 스택에 삽입하고 방문 처리를 합니다.
2. 스택의 최상단 노드에 방문하지 않은 인접한 노드가 하나라도 있으면 그 노드를 스택에 넣고 방문 처리합니다. 방문하지 않은 인접 노드가 없으면 스택에서 최상단 노드를 꺼냅니다.
3. 더 이상 2번의 과정을 수행할 수 없을 때까지 반복합니다.

ex) 깊은 방향으로 탐색, 6까지 먼저 방문하고 그 이후 7로 돌아가 8 방문

### DFS 동작 예시

- [Step 0] 그래프를 준비합니다. (방문 기준: 번호가 낮은 인접 노드부터)
  - 시작 노드: 1



|   |
|---|
|   |
|   |
|   |
|   |
| 8 |
| 7 |
| 2 |
| 1 |

구현

```
# DFS 메서드 정의
def dfs(graph, v, visited):
    # 현재 노드를 방문 처리
    visited[v] = True
    print(v, end=' ')
    # 현재 노드와 연결된 다른 노드를 재귀적으로 방문
    for i in graph[v]:
        if not visited[i]:
            dfs(graph, i, visited)
```

실행 결과

1 2 7 6 8 3 4 5

```
# 각 노드가 연결된 정보를 표현 (2차원 리스트)
graph = [
    [],
    [2, 3, 8],
    [1, 7],
    [1, 4, 5],
    [3, 5],
    [3, 4],
    [7],
    [2, 6, 8],
    [1, 7]
]
```

```
# 각 노드가 방문된 정보를 표현 (1차원 리스트)
visited = [False] * 9
```

```
# 정의된 DFS 함수 호출
dfs(graph, 1, visited)
```

BFS(너비우선 탐색) / 특정 조건에 대한 최단경로에서 사용

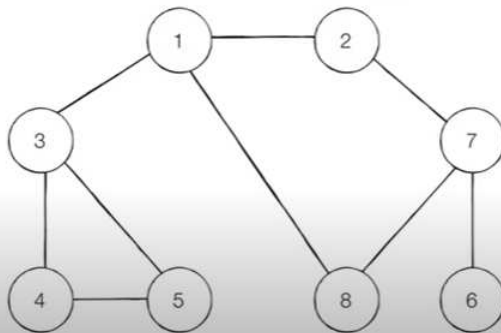
BFS는 너비 우선 탐색이라고도 부르며, 그래프에서 가까운 노드부터 우선적으로 탐색하는 알고리즘입니다.

BFS는 큐 자료구조를 이용하며, 구체적인 동작 과정은 다음과 같습니다.

1. 탐색 시작 노드를 큐에 삽입하고 방문 처리를 합니다.
2. 큐에서 노드를 꺼낸 뒤에 해당 노드의 인접 노드 중에서 방문하지 않은 노드를 모두 큐에 삽입하고 방문 처리합니다.
3. 더 이상 2번의 과정을 수행할 수 없을 때까지 반복합니다.

[Step 0] 그래프를 준비합니다. (방문 기준: 번호가 낮은 인접 노드부터)

- 시작 노드: 1



## BFS 소스코드 예제 (Python)

```
from collections import deque

# BFS 메서드 정의
def bfs(graph, start, visited):
    # 큐(Queue) 구현을 위해 deque 라이브러리 사용
    queue = deque([start])
    # 현재 노드를 방문 처리
    visited[start] = True
    # 큐가 빌 때까지 반복
    while queue:
        # 큐에서 하나의 원소를 뽑아 출력하기
        v = queue.popleft()
        print(v, end=' ')
        # 아직 방문하지 않은 인접한 원소들을 큐에 삽입
        for i in graph[v]:
            if not visited[i]:
                queue.append(i)
                visited[i] = True
```

```
# 각 노드가 연결된 정보를 표현 (2차원 리스트)
graph = [
    [],
    [2, 3, 8],
    [1, 7],
    [1, 4, 5],
    [3, 5],
    [3, 4],
    [7],
    [2, 6, 8],
    [1, 7]
]

# 각 노드가 방문된 정보를 표현 (1차원 리스트)
visited = [False] * 9

# 정의된 BFS 함수 호출
bfs(graph, 1, visited)
```

1 2 3 8 7 4 5 6