DFS/BFS

탐색이란?

많은 양의 데이터 중에서 원하는 데이터를 찾는 과정을 의미한다. 보통 그래프, 트리 등의 자료구조 안에서 탐색을 하는 문제를 자주 다루며, 탐색 알고리즘의 대표로 **DFS, BFS** 가 있다.

Stack

FILO 구조를 띄는 자료구조이다. 코드로는 아래와 같이 표현할 수 있다.

```
stack = []

stack.append(5)
stack.append(2)
stack.pop()
stack.append(6)

print(stack) # [5, 6]
print(stack[::-1]) # 최상단 원소부터 출력 [6, 5]
```

Queue

FIFO 구조를 띄는 자료구조이다. 코드로는 아래와 같이 구현이 가능하다.

```
from collections import deque

queue = deque()

queue.append(5)
queue.append(2)
queue.append(3)
queue.append(7)
queue.popleft()
queue.append(1)
queue.append(4)
queue.popleft()

print(queue) # deque([3, 7, 1, 4])
queue.reverse()
print(queue) # deque([4, 1, 7, 3])
```

재귀 함수

DFS와 BFS를 구현하기 위해선 재귀 함수의 이해도 필요하다. 간단한 재귀 함수는 아래와 같다.

```
def recursive_function():
  print('재귀 함수 호출')
  recursive_function()

recursive_function()
```

위와 같이 작성 시, 함수가 계속 돌기 때문에 에러가 발생한다. 재귀 함수에서는 이를 방지하기 위해 반드시 **종료 조건 작성**이 필요하다. 예시는 아래와 같다.

```
def recursive_function(i):
  if i == 100:
    return

  print(i)

  i += 1
  recursive_function(i)

recursive_function(0)
```

0~99까지 출력 후 종료된다.

DFS

Depth-First Search의 약자로 깊이 우선 탐색이라고도 부르며, **그래프에서 깊은 부분을 우선적으로 탐색**하는 알고리즘이다.

> 그래프란?

그래프는 **노드(Node), 간선(Edge)** 으로 표현되며 이때 노드를 정점(Vertex) 이라고도 말한다. 그래프 탐색이란 하나의 노드를 시작으로 다수의 노드를 방문하는 것을 말한다. 또한 두 노드가 간선으로 연결되어 있다면 **두 노드는 인접하다** 라고 표현한다.

프로그래밍에서 그래프는 크게 2가지 방식으로 표현이 가능하며, 이는 **인접 행렬**과 **인접 리스트**이다.

- **인접 행렬** : 2차원 배열로 그래프의 연결 관계를 표현하는 방식
- **인접 리스트** : 리스트로 그래프의 연결 관계를 표현하는 방식

인접 행렬 방식

2차원 배열에 각 노드가 연결된 형태를 기록하는 방식이다. 아래는 예제이다. 각 노드끼리의 간선을 저장하며, 본인~본인은 0, 인접하지 않은 노드끼리는 INF를 저장한다.

```
INF = 999999999 # 무한의 비용 선언

graph = [
  [0, 7, 5],
  [7, 0, INF],
  [5, INF, 0]
]

print(graph)
```

인접 리스트 방식

모든 노드에 연결된 노드에 대한 정보를 차례대로 연결하여 저장하는 방식이다. 타 언어의 경우 연결 리스트를 직접 구현하거나 표준 라이브러리를 사용하지만 파이썬에서는 일반 리스트 자료형으로 구현이 가능하다. 아래는 예제이다.

```
graph = [[] for _ in range(3)]

# 노드 0에 연결된 노드 정보 저장(노드, 거리)
graph[0].append((1, 7))
graph[0].append((2, 5))

graph[1].append((0, 7))

graph[2].append((0, 5))

print(graph) # [[(1, 7), (2, 5)], [(0, 7)], [(0, 5)]]
```

인접 행렬 vs 인접 리스트

1. 메모리 측면
    - 인접 행렬 방식 : 모든 관계를 저장하기 때문에 노드 개수가 많을수록 메모리가 불필요하게 낭비된다.
    - 인접 리스트 방식 : 연결된 정보만을 저장하기 때문에 메로리를 효율적으로 사용한다.
2. 속도 측면
    - 인접 행렬 방식 : 특정 두 노드의 연결에 대한 정보를 얻는 속도가 빠르다.
    - 인접 리스트 방식 : 특정 두 노드가 연결되어 있는지에 대한 정보를 얻는 속도가 느리다. 연결된 데이터를 하나씩 확인해야 하기 때문이다.

다시 DFS로..

DFS는 특정한 경로를 탐색하다가 특정한 상황에서 최대한 깊숙이 들어가 노드를 방문한 후, 다시 돌아가 다른 경로를 탐색하는 알고리즘이다. DFS는 스택 자료구조를 사용하며 구체적인 동작 과정은 다음과 같다.

1. 탐색 시작 노드를 스택에 삽입하고 방문 처리를 한다.
2. 스택의 최상단 노드에 방문하지 않은 인접 노드가 있다면 그 인접 노드를 스택에 넣고 방문 처리를 한다. 방문하지 않은 인접 노드가 없다면 스택에서 최상단 노드를 꺼낸다.
3. 2번 과정을 더 이상 수행할 수 없을 때까지 반복한다.

DFS는 스택 자료구조를 사용하기 때문에 구현이 간단하며, O(N)의 시간이 소요된다는 특징이 있다. 또한 스택을 이용하는 알고리즘이기 때문에 실제 구현은 재귀 함수를 이용했을 때 매우 간결하게 구현할 수 있다.

1. 인접 리스트를 이용한 DFS

```
def dfs(graph, v, visited):
  visited[v] = True
  print(v, end=' ')

  for i in graph[v]:
    if not visited[i]:
      dfs(graph, i, visited)

graph = [
  [],
  [2, 3, 8],
  [1, 7],
  [1, 4, 5],
  [3, 5],
  [3, 4],
  [7],
  [2, 6, 8],
  [1, 7]
]

visited = [False] * len(graph)

dfs(graph, 1, visited)
```

2. 인접 행렬을 이용한 DFS

```
# N = 정점 수, M = 간선 수
N, M = map(int, sys.stdin.readline().strip().split())

graph = [[0] * (N + 1) for _ in range(N + 1)]

for _ in range(M):
    x, y = map(int, sys.stdin.readline().strip().split()) # 간선 입력
    graph[x][y] = graph[y][x] = 1

def dfs(start, stack=[]):
    stack.append(start)
    print(start, end=' ')

    for i in range(N + 1):
        if graph[start][i] == 1 and (i not in stack):
            dfs(i, stack)

dfs(1)
```

- 백준에서 DFS + 재귀를 이용할 경우 재귀의 최대 깊이를 설정해두어야 한다. sys를 import 하고 sys.setrecursionlimit(10**6) 만 작성해주면 된다.

BFS

너비 우선 탐색이라는 의미로 **가까운 노드부터 탐색하는 알고리즘**이다. DFS는 최대한 멀리 있는 노드를 우선적으로 탐색하지만 BFS는 그 반대이다.

BFS는 큐 자료구조를 이용하며, 인접한 노드를 반복적으로 큐에 넣도록 알고리즘을 작성하면 자연스럽게 먼저 들어온 것이 먼저 나가게 되어, 가까운 노드부터 탐색을 진행하게 된다.

1. 탐색 시작 노드를 큐에 삽입하고 방문 처리를 한다.
2. 큐에서 노드를 꺼내 해당 노드의 인접 노드 중에서 방문하지 않은 노드를 모두 큐에 삽입하고 방문 처리를 한다.
3. 2번의 과정을 더 이상 수행할 수 없을 때까지 반복한다.

BFS 또한 O(N)의 시간이 소요되지만 일반적인 경우 실제 수행 시간은 DFS보다 조금 더 좋은 편이다.

1. 인접 리스트를 이용한 BFS

```
from collections import deque

def bfs(graph, start, visited):
  queue = deque([start])
  visited[start] = True

  while queue:
    v = queue.popleft()
    print(v, end=' ')

    for i in graph[v]:
      if not visited[i]:
        queue.append(i)
        visited[i] = True

graph = [
  [],
  [2, 3, 8],
  [1, 7],
  [1, 4, 5],
```

```
  [3, 5],
  [3, 4],
  [7],
  [2, 6, 8],
  [1, 7]
]

visited = [False] * len(graph)

bfs(graph, 1, visited)
```

2. 인접 행렬을 이용한 BFS

```
# N = 정점 수, M = 간선 수
N, M = map(int, sys.stdin.readline().strip().split())

graph = [[0] * (N + 1) for _ in range(N + 1)]

for _ in range(M):
    x, y = map(int, sys.stdin.readline().strip().split())
    graph[x][y] = graph[y][x] = 1

def bfs(start):
    discovered = [start]
    queue = deque([start])

    while queue:
        v = queue.popleft()
        print(v, end=' ')

        for i in range(N + 1):
            if graph[v][i] == 1 and (i not in discovered):
                discovered.append(i)
                queue.append(i)

bfs(1)
```

💡 문제 1. 음료수 얼려 먹기

N x M 크기의 얼음 틀이 있다. 구멍이 뚫린 부분을 0, 칸막이가 있는 부분을 1로 할 때 나오는 아이스크림의 총 개수를 구하여라.

해결

DFS를 이용해 얼음 틀의 처음부터 탐색을 하며 0인 부분을 발견하면 상하좌우로 이어진 부분들을 모두 1로 바꾸고 True를 반환한다. 그리고 True의 개수를 출력하면 끝

코드

```
import sys

n, m = sys.stdin().readline().strip().split()

graph = []

for i in range(n):
  datas = list(map(int, sys.stdin().readline().strip()))
  graph.append(datas)

def dfs(x, y):
  if x <= -1 or x >= n or y <= -1 or y >= 1:
    return False

  if graph[x][y] == 0:
    graph[x][y] = 1

    dfs(x + 1, y)
    dfs(x - 1, y)
    dfs(x, y + 1)
    dfs(x, y - 1)

    return True

  return False

result = 0

for i in range(n):
  for j in range(m):
    if graph[i][j]:
      result += 1

print(result)
```

💡 문제 2. 미로 탈출

N x M 크기의 미로에 갇혀있다. 미로에는 여러 마리의 괴물이 존재해 이를 피해 탈출해야 한다. 초기 위치는 (1, 1)이고 미로의 출구는 (N, M)의 위치에 존재하며 한 번에 한 칸씩 이동이 가능하다. 이때, 괴물이 있는 부분은 0, 아닌 부분은 1로 표시되어 있다. 미로는 반드시 탈출할 수 있는 형태로 주어질 때, 탈출하기 위해 움직여야 하는 최소 칸의 개수를 구하라. 단, 시작 칸과 마지막 칸 모두 카운트한다.

해결

BFS를 이용해 이동을 해 가며 노드의 값을 이전 탐색 노드의 값 + 1로 변경해가면 괴물이 없는 칸의 경우 시작 위치에서부터의 거리가 모두 표시된다.

코드

```
import sys
from collections import deque

N, M = map(int, sys.stdin.readline().strip().split())

graph = []

for _ in range(N):
  graph.append(list(map(int, sys.stdin.readline().strip())))

dx = [-1, 1, 0, 0]
dy = [0, 0, 1, -1]

def bfs(x, y):
  queue = deque((x, y))

  while queue:
    x, y = queue.popleft()

    for i in range(4):
      nx = x + dx[i]
      ny = y + dy[i]

      if nx < 0 or ny < 0 or nx >= N or ny >= M:
        continue

      if graph[nx][ny] == 0:
        continue
```

```
      if graph[nx][ny] == 1:
        queue.append((nx, ny))
        graph[nx][ny] = graph[x][y] + 1

  return graph[N - 1][M - 1]

print(bfs(0, 0))
```