아래 강의 영상에서 배운 내용을 정리해보려고 한다. 가감된 부분이 있을 수 있다.
https://youtu.be/7C9RgOcvkvo?list=PLRx0vPvlEmdAghTr5mXQxGpHjWqSz0dgC

그래프 탐색 알고리즘: DFS/BFS

탐색(Search)이란 많은 양의 데이터 중에서 **원하는 데이터를 찾는 과정**을 말한다.
대표적인 그래프 탐색 알고리즘으로는 DFS와 BFS가 있다.
DFS/BFS는 코딩 테스트에서 매우 자주 등장하는 유형이므로 반드시 숙지해야 한다.

스택 자료구조

먼저 들어온 데이터가 나중에 나가는 형식(선입후출, FILO, First In Last Out)의 자료구조이다.
입구와 출구가 동일한 형태로 스택을 시각화할 수 있다. 아래처럼.

왼쪽 프링글스 사진 출처

물론 프링글스 통에 먹던 과자를 다시 넣거나 휴지통에 버린 휴지를 다시 꺼내는 경우는 드물다.

스택 구현 예제

예제의 리스트는 'ㄷ'자 형태의 스택 자료구조를 흉내낸 것이라고 볼 수 있다.

```
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

''' 출력
[1, 3, 2, 5]
[5, 2, 3, 1]
'''
```

큐 자료구조

먼저 들어온 데이터가 먼저 나가는 형식(선입선출, FIFO, First In First Out)의 자료구조이다.
큐는 **입구와 출구가 모두 뚫려 있는 터널과 같은 형태**로 시각화 할 수 있다.

큐 구현 예제

큐 구현에 있어 리스트보다 덱(deque)을 이용하는 것이 시간 복잡도가 더 낮다.
예제의 큐는 이런 형태라고 볼 수 있다.

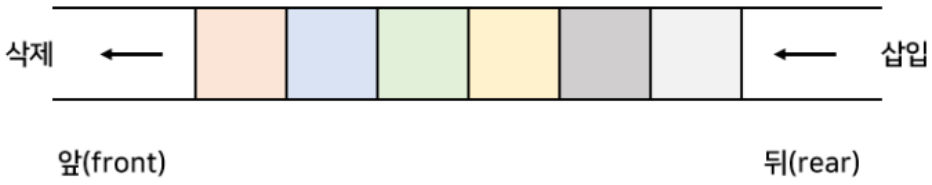


사진 출처

```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


''' 출력
deque([3, 7, 1, 4])
deque([4, 1, 7, 3])
'''
```

재귀 함수

재귀 함수(Recursive Function)란 **자기 자신은 다시 호출하는 함수**를 의미한다.

단순한 형태의 재귀 함수 예제는 다음과 같다.

- '재귀 함수를 호출합니다.'라는 문자열을 무한히 출력한다.
- 어느 정도 출력하다가 최대 재귀 깊이 초과 메시지가 출력된다.

```
def recursive_function():
    print('재귀 함수를 호출합니다.')
    recursive_function()

recursive_function()

''' 출력
재귀 함수를 호출합니다.
재귀 함수를 호출합니다.
재귀 함수를 호출합니다.
... (중략)
재귀 함수를 호출합니다.
Traceback (most recent call last):
  File "main.py", line 5, in <module>
    recursive_function()
  File "main.py", line 3, in recursive_function
    recursive_function()
  File "main.py", line 3, in recursive_function
    recursive_function()
  File "main.py", line 3, in recursive_function
    recursive_function()
  [Previous line repeated 992 more times]
  File "main.py", line 2, in recursive_function
    print('재귀 함수를 호출합니다.')
RecursionError: maximum recursion depth exceeded while calling a Python object
'''
```

재귀 함수의 종료 조건

재귀 함수를 문제 풀이에서 사용할 때는 재귀 함수의 종료 조건을 반드시 명시해야 한다.
종료 조건을 제대로 명시하지 않으면 함수가 무한히 호출될 수 있다.
종료 조건을 포함한 재귀 함수 예제는 다음과 같다.

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

recursive_function(1)

''' 출력
1번째 재귀함수에서 2번째 재귀함수를 호출합니다.
2번째 재귀함수에서 3번째 재귀함수를 호출합니다.
3번째 재귀함수에서 4번째 재귀함수를 호출합니다.
... (중략)
99번째 재귀함수에서 100번째 재귀함수를 호출합니다.
99번째 재귀함수를 종료합니다.
98번째 재귀함수를 종료합니다.
97번째 재귀함수를 종료합니다.
... (중략)
1번째 재귀함수를 종료합니다.
'''
```

이처럼 재귀함수를 이용하게 되면 마치 **스택**에 데이터를 넣었다가 꺼내는 것과 마찬가지로 각각의 함수에 대한 정보가 실제로 스택 프레임에 담기게 되어서, 가장 처음에 호출된 함수부터 차례대로 호출이 되었다가 가장 마지막에 호출된 함수부터 차례대로 종료가 된다.

팩토리얼 구현 예제

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

수학적으로 0!과 1!의 값은 1이다.

```python
# 반복적으로 구현한 n!
def factorial_iterative(n):
    result = 1
    # 1부터 n까지의 수를 차례대로 곱하기
    for i in range(1, n + 1):
        result *= i
    return result

# 재귀적으로 구현한 n!
def factorial_recursive(n):
    if n <= 1: # n이 1 이하인 경우 1을 반환
        return 1
    # n! = n * (n - 1)!를 그대로 코드로 작성하기
    return n * factorial_recursive(n - 1)

# 각각의 방식으로 구현한 n! 출력(n = 5)
print("반복적으로 구현:", factorial_iterative(5))
print("재귀적으로 구현:", factorial_recursive(5))

''' 출력
반복적으로 구현: 120
재귀적으로 구현: 120
'''
```

최대공약수 계산(유클리드 호제법) 예제

두 개의 자연수에 대한 최대공약수를 구하는 대표적인 알고리즘으로는 유클리드 호제법이 있다.

유클리드 호제법
- 두 자연수 A, B에 대하여 (A > B) A를 B로 나눈 나머지를 R이라고 한다.
- 이 때 A와 B의 최대공약수는 B와 R의 최대공약수와 같다.

유클리드 호제법 증명은 아래 블로그 글을 보면 쉽게 이해할 수 있다.
https://blog.naver.com/papers/140207307545

유클리드 호제법 증명
유클리드 호제법 증명 A와 B의 최대공약수와 B와R…
blog.naver.com

유클리드 호제법의 아이디어를 그대로 재귀 함수로 작성할 수 있다.
큰 수가 꼭 첫 번째 매개변수로 들어가지 않아도 답은 똑같이 나온다.

작은 수를 큰 수로 나누면 몫은 0, 나머지는 큰 수가 되므로 큰 수와 작은 수의 위치가 바뀌어 다시 재귀 함수를 호출하기 때문이다.
다만 그래서 답은 똑같이 나와도 작은 수가 앞에 들어가면 큰 수가 앞에 있을 때보다 함수 호출 횟수가 한 번 더 늘어난다.

```
def gcd(a, b):
    if a % b == 0:
        return b
    else:
        return gcd(b, a % b)

print(gcd(192, 162))

''' 출력
6
'''
```

- **예시**: GCD(192, 162)

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

재귀 함수 사용의 유의 사항

재귀 함수를 잘 활용하면 복잡한 알고리즘을 간결하게 작성할 수 있다.
단, 오히려 다른 사람이 이해하기 어려운 형태의 코드가 될 수도 있으므로 신중하게 사용해야 한다.
모든 재귀 함수는 반복문을 이용하여 동일한 기능을 구현할 수 있다.
재귀 함수가 반복문보다 유리한 경우도 있고 불리한 경우도 있다.
컴퓨터가 함수를 연속적으로 호출하면 컴퓨터 메모리 내부의 스택 프레임에 쌓인다.
그래서 스택을 사용해야 할 때 구현상 **스택 라이브러리 대신에 재귀 함수를 이용**하는 경우가 많다(예: DFS).

DFS(Depth-First Search)

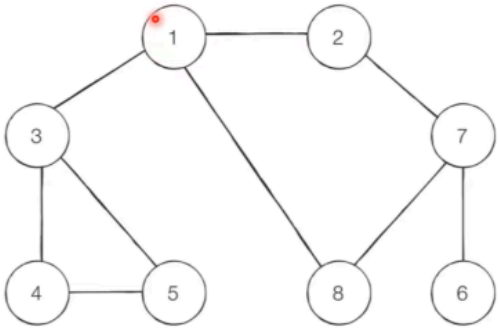
DFS는 **깊이 우선 탐색**이라고도 부르며 그래프에서 **깊은 부분을 우선적으로 탐색하는 알고리즘**이다.
DFS는 **스택 자료구조(혹은 재귀 함수)를 이용**하며, 구체적인 동작 과정은 다음과 같다.

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

DFS 동작 예시

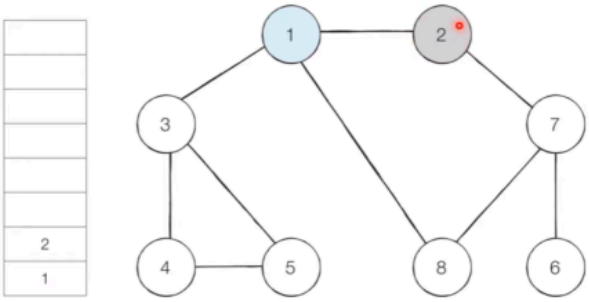
DFS 동작 예시

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



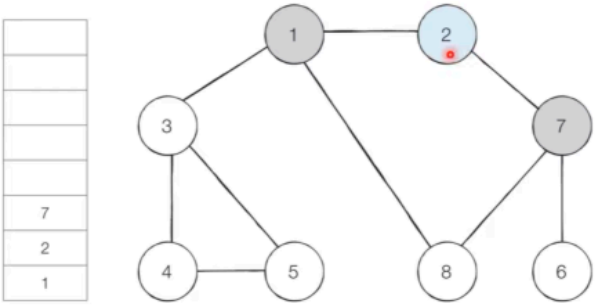
DFS 동작 예시

- **[Step 2]** 스택의 최상단 노드인 '1'에 방문하지 않은 인접 노드 '2', '3', '8'이 있습니다.
  - 이 중에서 가장 작은 노드인 '2'를 스택에 넣고 방문 처리를 합니다.



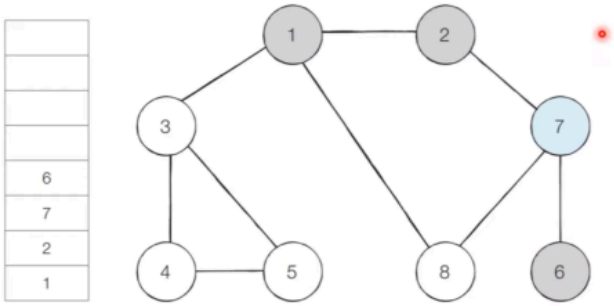
DFS 동작 예시

- **[Step 3]** 스택의 최상단 노드인 '2'에 방문하지 않은 인접 노드 '7'이 있습니다.
  - 따라서 '7'번 노드를 스택에 넣고 방문 처리를 합니다.



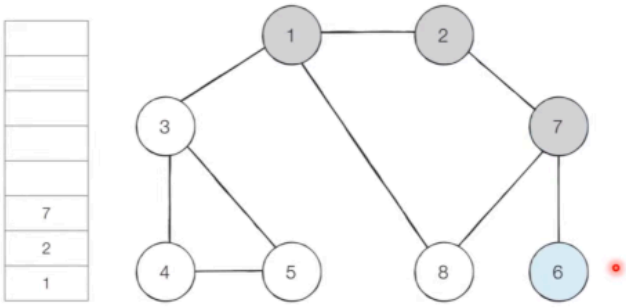
DFS 동작 예시

- **[Step 4]** 스택의 최상단 노드인 '7'에 방문하지 않은 인접 노드 '6', '8'이 있습니다.
  - 이 중에서 가장 작은 노드인 '6'을 스택에 넣고 방문 처리를 합니다.



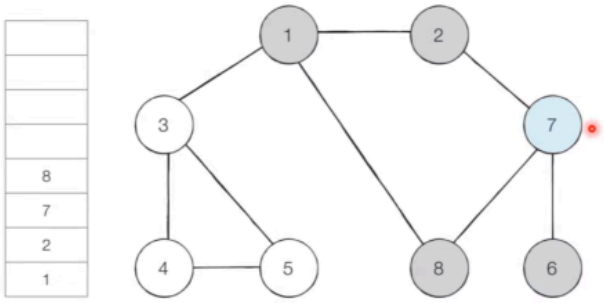
DFS 동작 예시

- **[Step 5]** 스택의 최상단 노드인 '6'에 방문하지 않은 인접 노드가 없습니다.
  - 따라서 스택에서 '6'번 노드를 꺼냅니다.



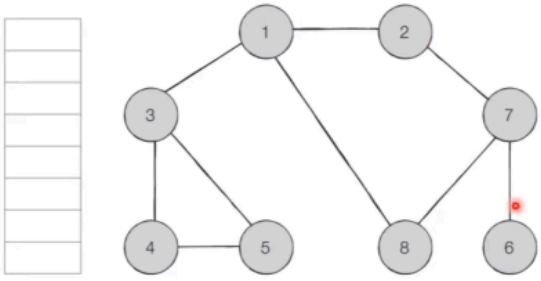
DFS 동작 예시

- **[Step 6]** 스택의 최상단 노드인 '7'에 방문하지 않은 인접 노드 '8'이 있습니다.
  - 따라서 '8'번 노드를 스택에 넣고 방문 처리를 합니다.



DFS 동작 예시

- 이러한 과정을 반복하였을 때 **전체 노드의 탐색 순서**(스택에 들어간 순서)는 다음과 같습니다.



탐색 순서: 1 → 2 → 7 → 6 → 8 → 3 → 4 → 5

DFS 소스코드 예제

앞서 나온 DFS 동작 예시의 그래프와 동일한 그래프이다.

```python
# 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)

# 각 노드가 연결된 정보를 표현 (2차원 리스트)
graph = [
    [], # 0번 노드는 없으므로 비워둠(사용하지 않음)
    [2, 3, 8], # 1번 노드와 연결된 노드
    [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)

''' 출력
1 2 7 6 8 3 4 5
'''
```

BFS(Breadth-First Search)

BFS는 **너비 우선 탐색**이라고도 부르며, 그래프에서 **가까운 노드부터 우선적으로 탐색하는 알고리즘**이다.
BFS는 **큐 자료구조**를 이용하며, 구체적인 동작 과정은 다음과 같다.

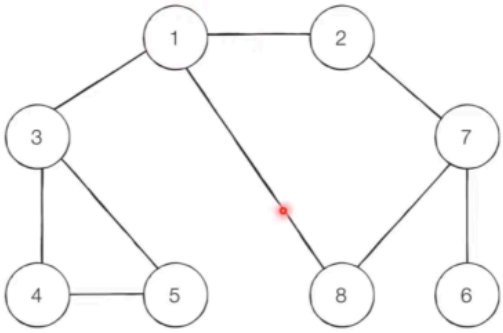
1. 탐색 시작 노드를 큐에 삽입하고 방문 처리를 한다.
2. 큐에서 노드를 꺼낸 뒤에 해당 노드의 인접 노드 중에서 방문하지 않은 노드를 모든 큐에 삽입하고 방문 처리한다.
3. 더 이상 2번의 과정을 수행할 수 없을 때까지 반복한다.

BFS는 특정 조건에서의 최단 경로 문제를 해결하기 위한 목적으로 사용되기도 한다.

BFS 동작 예시

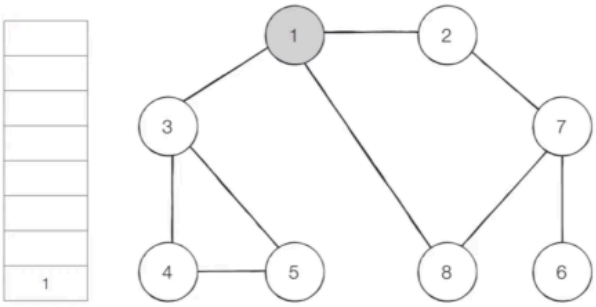
BFS 동작 예시

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



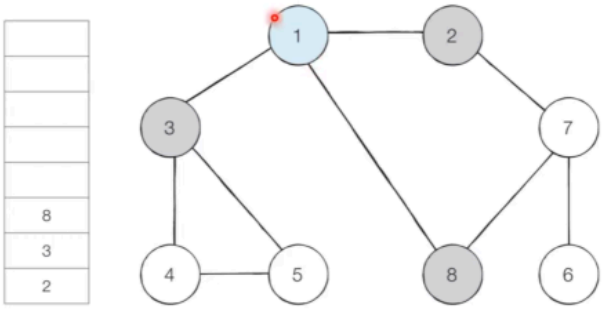
BFS 동작 예시

- **[Step 1]** 시작 노드인 '1'을 큐에 삽입하고 방문 처리를 합니다.



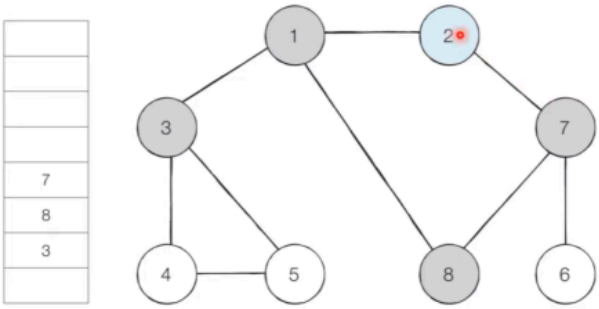
BFS 동작 예시

- **[Step 2]** 큐에서 노드 '1'을 꺼내 방문하지 않은 인접 노드 '2', '3', '8'을 큐에 삽입하고 방문 처리합니다.



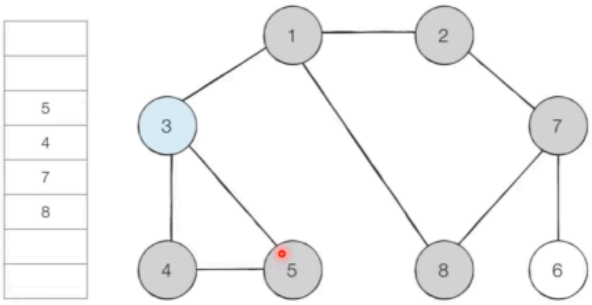
BFS 동작 예시

- **[Step 3]** 큐에서 노드 '2'를 꺼내 방문하지 않은 인접 노드 '7'을 큐에 삽입하고 방문 처리합니다.



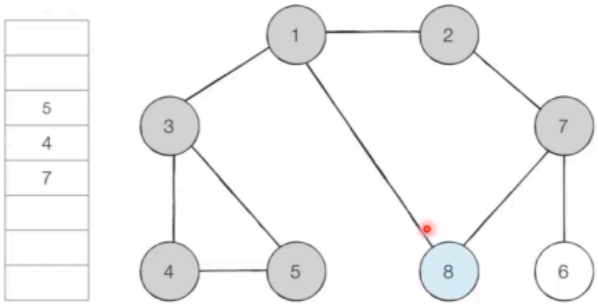
BFS 동작 예시

- **[Step 4]** 큐에서 노드 '3'을 꺼내 방문하지 않은 인접 노드 '4', '5'를 큐에 삽입하고 방문 처리합니다.



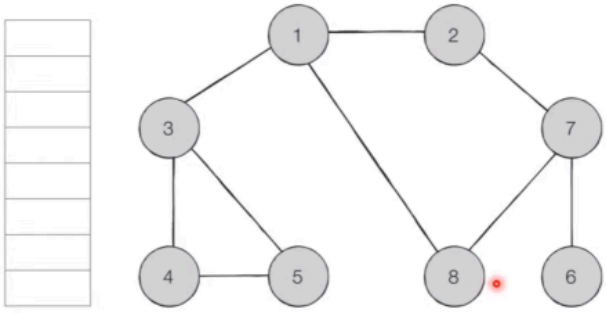
BFS 동작 예시

- **[Step 5]** 큐에서 노드 '8'을 꺼내고 방문하지 않은 인접 노드가 없으므로 무시합니다.



BFS 동작 예시

- 이러한 과정을 반복하여 **전체 노드의 탐색 순서**(큐에 들어간 순서)는 다음과 같습니다.



탐색 순서: 1 → 2 → 3 → 8 → 7 → 4 → 5 → 6

BFS 소스코드 예제

앞서 나온 DFS, BFS 동작 예시의 그래프와 동일한 그래프이다.

```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 = [
    [], # 0번 노드는 없으므로 비워둠(사용하지 않음)
    [2, 3, 8], # 1번 노드와 연결된 노드
    [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)

''' 출력
1 2 3 8 7 4 5 6
'''
```

DFS/BFS 예시: 음료수 얼려 먹기

〈문제〉 음료수 얼려 먹기: 문제 설명

- N × M 크기의 얼음 틀이 있습니다. 구멍이 뚫려 있는 부분은 0, 칸막이가 존재하는 부분은 1로 표시됩니다. 구멍이 뚫려 있는 부분끼리 상, 하, 좌, 우로 붙어 있는 경우 서로 연결되어 있는 것으로 간주합니다. 이때 얼음 틀의 모양이 주어졌을 때 생성되는 총 아이스크림의 개수를 구하는 프로그램을 작성하세요. 다음의 4 × 5 얼음 틀 예시에서는 아이스크림이 총 3개 생성됩니다.

```
00110
00011
11111
00000
```

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

〈문제〉 음료수 얼려 먹기: 문제 조건

난이도 ●◐○ | 풀이 시간 30분 | 시간제한 1초 | 메모리 제한 128MB

입력 조건
- 첫 번째 줄에 얼음 틀의 세로 길이 N과 가로 길이 M이 주어집니다. (1 <= N, M <= 1,000)
- 두 번째 줄부터 N + 1번째 줄까지 얼음 틀의 형태가 주어집니다.
- 이때 구멍이 뚫려있는 부분은 0, 그렇지 않은 부분은 1입니다.

출력 조건
- 한 번에 만들 수 있는 아이스크림의 개수를 출력합니다.

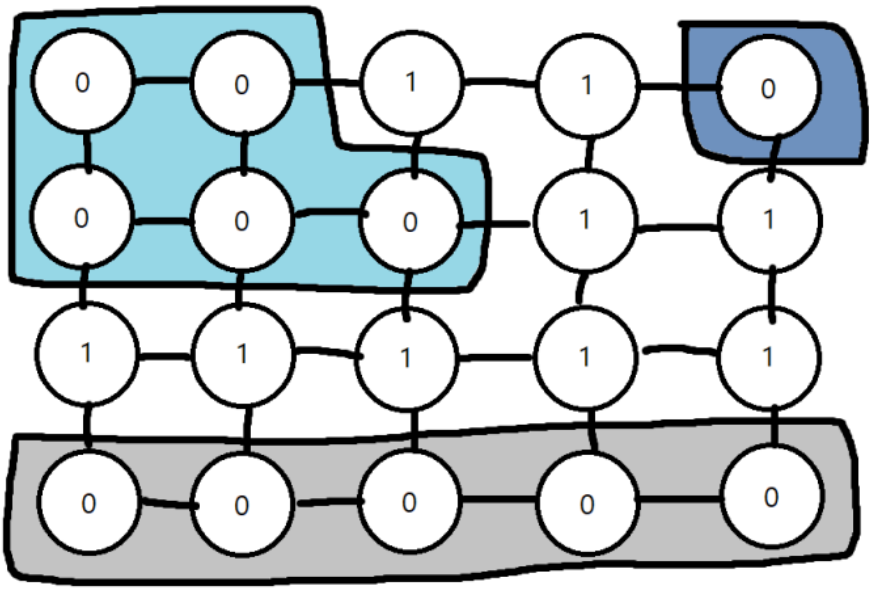
입력 예시

```
4 5
00110
00011
11111
00000
```

출력 예시

```
3
```

이 문제는 DFS 혹은 BFS로 해결할 수 있다.
얼음을 얼릴 수 있는 공간이 상, 하, 좌, 우로 연결되어 있다고 표현할 수 있으므로 그래프 형태로 모델링 할 수 있다.
위 입력 예시를 가지고 4*5 크기의 얼음 틀 그래프를 그려본다면 아래와 같은 모습일 것이다.



DFS를 활용하는 알고리즘은 다음과 같다.

1. 특정한 지점의 주변 상, 하, 좌, 우를 살펴본 뒤에 주변 지점 중에서 값이 0이면서 아직 방문하지 않은 지점이 있다면 해당 지점을 방문한다.

2. 방문한 지점에서 다시 상, 하, 좌, 우를 살펴보면서 방문을 진행하는 과정을 반복하면, **연결된 모든 지점은 방문할 수 있다.**
3. 모든 노드에 대하여 1 ~ 2번의 과정을 반복하며, 방문하지 않은 지점의 수를 카운트한다.

좀 더 쉽게 말로 풀어보면 이렇다.
0을 만나면 그 0을 포함해 그 0과 이어져있는 모든 0을 1로 만든다.
그림판의 '색 채우기' 같은 느낌이다. 그것도 선택한 지점과 같은 색상으로 이어져있는 모든 곳의 색상을 지정한 색상으로 바꾸는 것이니까.
그리고 카운트한다. 이걸 모든 노드에서 실행하는 것이다.

```
# DFS로 특정 노드를 방문하고 연결된 모든 노드들도 방문
def dfs(x, y):
    # 주어진 범위를 벗어나는 경우에는 즉시 종료
    if x <= -1 or x >= n or y <= -1 or y >= m:
        return False
    # 현재 노드를 아직 방문하지 않았다면
    if graph[x][y] == 0:
        # 해당 노드 방문 처리
        graph[x][y] = 1
        # 상, 하, 좌, 우의 위치들도 모두 재귀적으로 호출
        dfs(x - 1, y)
        dfs(x, y - 1)
        dfs(x + 1, y)
        dfs(x, y + 1)
        return True
    return False

# N, M을 공백을 기준으로 구분하여 입력받기
n, m = map(int, input().split())

# 2차원 리스트의 맵 정보 입력 받기
graph = []
for i in range(n):
    graph.append(list(map(int, input())))

# 모든 노드(위치)에 대하여 음료수 채우기
result = 0
for i in range(n):
    for j in range(m):
        # 현재 위치에서 DFS 수행
        if dfs(i, j) == True:
            result += 1

print(result) # 정답 출력
```

DFS/BFS 예시: 미로 탈출

〈문제〉 미로 탈출: 문제 설명

- 동빈이는 N × M 크기의 직사각형 형태의 미로에 갇혔습니다. 미로에는 여러 마리의 괴물이 있어 이를 피해 탈출해야 합니다.
- 동빈이의 위치는 (1, 1)이며 미로의 출구는 (N, M)의 위치에 존재하며 한 번에 한 칸씩 이동할 수 있습니다. 이때 괴물이 있는 부분은 0으로, 괴물이 없는 부분은 1로 표시되어 있습니다. 미로는 반드시 탈출할 수 있는 형태로 제시됩니다.
- 이때 동빈이가 탈출하기 위해 움직여야 하는 최소 칸의 개수를 구하세요. 칸을 셀 때는 시작 칸과 마지막 칸을 모두 포함해서 계산합니다.

〈문제〉 미로 탈출: 문제 조건

난이도 ●◐○ | 풀이 시간 30분 | 시간제한 1초 | 메모리 제한 128MB

입력 조건 • 첫째 줄에 두 정수 N, M(4≤N, M≤200)이 주어집니다. 다음 N개의 줄에는 각각 M개의 정수(0 혹은 1)로 미로의 정보가 주어집니다. 각각의 수들은 공백 없이 붙어서 입력으로 제시됩니다. 또한 시작 칸과 마지막 칸은 항상 1입니다.

출력 조건 • 첫째 줄에 최소 이동 칸의 개수를 출력합니다.

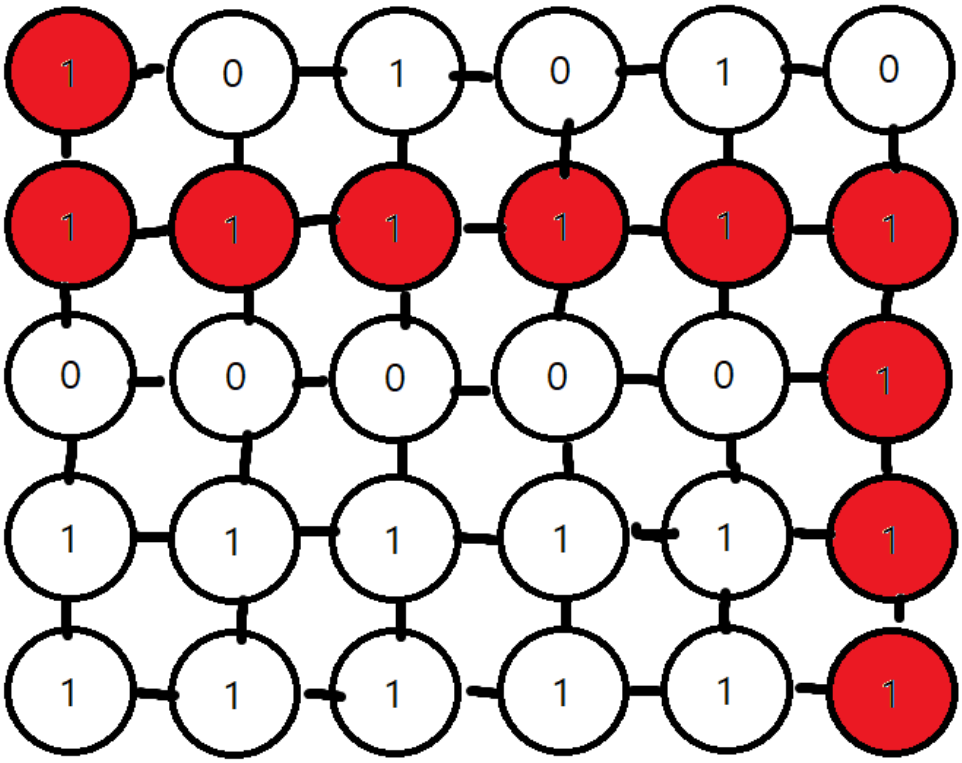
입력 예시

```
5 6
101010
111111
000001
111111
111111
```

출력 예시

```
10
```

BFS는 시작 지점에서 가까운 노드부터 차례대로 그래프의 모든 노드를 탐색한다.
상, 하, 좌, 우로 연결된 모든 노드로의 거리가 1로 동일하다.
따라서 (1, 1) 지점부터 BFS를 수행하여 모든 노드의 최단 거리 값을 기록하면 해결할 수 있다.
위 입력 예시를 가지고 미로를 그래프로 표현해본다면 아래와 같은 모습일 것이다. 붉은 칸은 최소 칸이다.



문제 해결 아이디어는 다음과 같다.

1. 처음에 (1, 1)의 위치에서 시작한다.

2. 위 그래프를 기준으로 (1, 1) 좌표에서 상, 하, 좌, 우로 탐색을 진행하면 바로 아래 노드인 (2, 1) 위치의 노드를 방문하게 되고 새돕게 방문하는 (2, 1) 노드의 값을 2로 바꾸게 된다.
3. 마찬가지로 BFS를 계속 수행하면 결과적으로 최단 경로의 값들이 1씩 증가하는 형태로, 미로의 출구에 근접할수록 숫자가 커지는 형태로 변경된다.

아래 입력을 넣었을 때 시작 칸 값이 1이 아닌 3이 나오는 게 이상해서 코드를 찍어봤더니 재미있는 것을 발견할 수 있었다.
미로 크기가 2*2 이상이면 시작 칸→다음 칸(1+1=2), 다음 칸→시작 칸(2+1=3)으로 이동하는 경우가 발생하여 시작 칸 값이 3이 된다.
시작 칸이 1이기 때문에 발생한 일이다.
이 때문에 혹시 반례가 생기는 것은 아닐까 걱정했는데 어차피 마지막 칸의 값이 중요한 거라 상관없는 듯하다.
일단 알고 있으면 좋을 듯.

```
3 4
1000
1111
0001
```

```python
from collections import deque

# BFS 소스코드 구현
def bfs(x, y):
    # 큐(Queue) 구현을 위해 deque 라이브러리 사용
    queue = deque()
    queue.append((x, y))
    # 큐가 빌 때까지 반복하기
    while queue:
        x, y = queue.popleft()
        # 현재 위치에서 4가지 방향으로의 위치 확인
        for i in range(4):
            nx = x + dx[i]
            ny = y + dy[i]
            # 미로 찾기 공간을 벗어난 경우 무시
            if nx < 0 or nx >= n or ny < 0 or ny >= m:
                continue
            # 벽인 경우 무시
            if graph[nx][ny] == 0:
                continue
            # 해당 노드를 처음 방문하는 경우에만 최단 거리 기록
            if graph[nx][ny] == 1:
                graph[nx][ny] = graph[x][y] + 1
                # print(x, y, end='->')
                # print(nx, ny)
                queue.append((nx, ny))

    # print("최종 그래프")
    # for i in range(n):
    #     for j in range(m):
    #         print(graph[i][j], end=' ')
    #     print()

    # 가장 오른쪽 아래까지의 최단 거리 반환
    return graph[n - 1][m - 1]

# N, M을 공백을 기준으로 구분하여 입력 받기
n, m = map(int, input().split())
# 2차원 리스트의 맵 정보 입력 받기
graph = []
for i in range(n):
    graph.append(list(map(int, input())))

# 이동할 네 가지 방향 정의 (상, 하, 좌, 우)
dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]

# BFS를 수행한 결과 출력
print(bfs(0, 0))
```