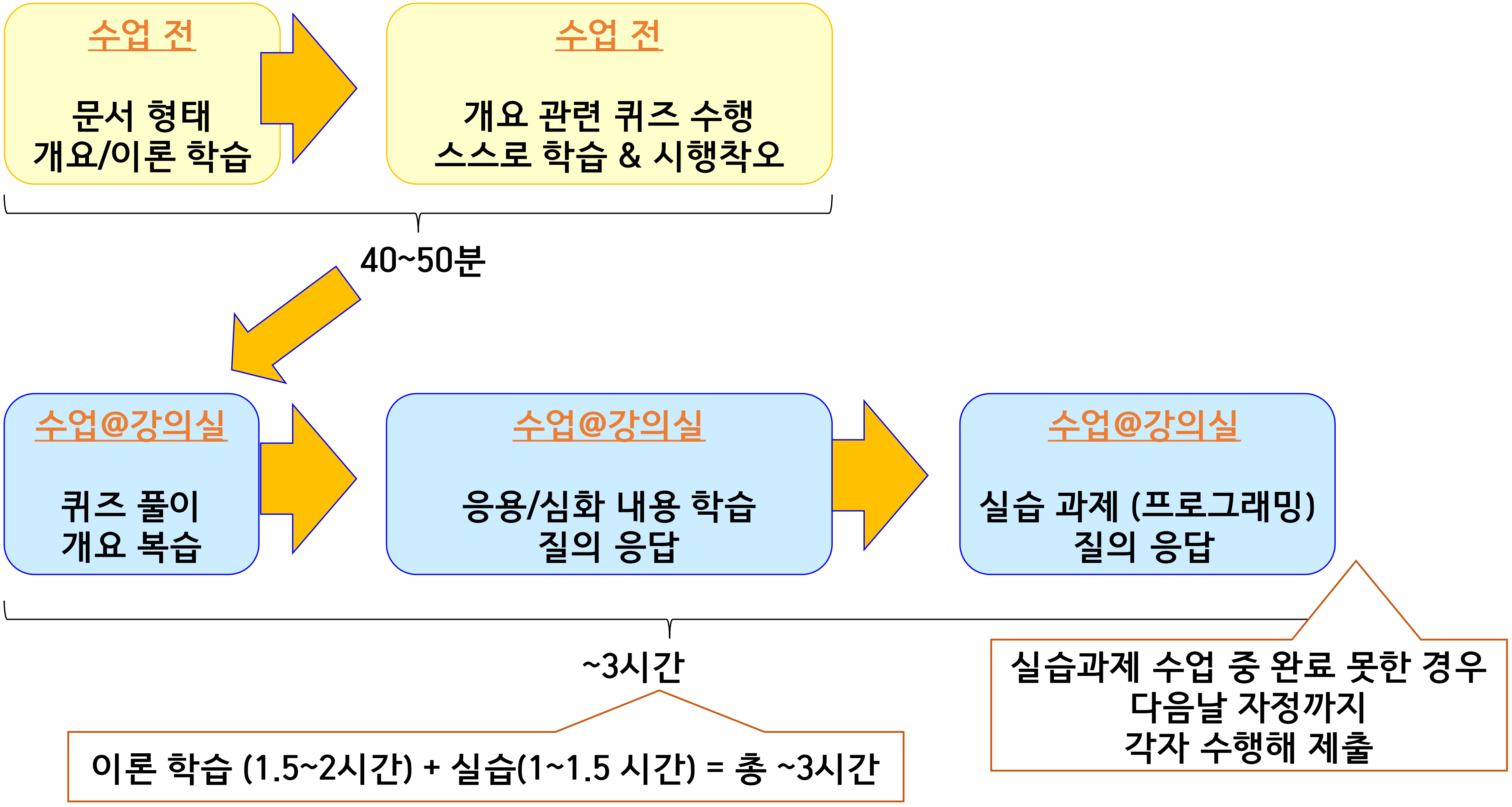
Union Find (Disjoint Set Forest)

Union Find 문제 정의, 해결 방법, 활용도 이해

lms의 수업자료 파일 이름에 '**-숫자**'가 붙은 것은
파일이 수정되었다는 뜻입니다. (예: [수업자료]**-3**.pdf)
기존에 받은 수업자료가 수정되었다면 다시 받아 두세요.

수업 전 예습 → 문제풀이/실습/질의응답 (플립 러닝, 거꾸로 학습)



수업 전

문서 형태
개요/이론 학습

수업 전

개요 관련 퀴즈 수행
스스로 학습 & 시행착오

40~50분

수업@강의실

퀴즈 풀이
개요 복습

수업@강의실

응용/심화 내용 학습
질의 응답

수업@강의실

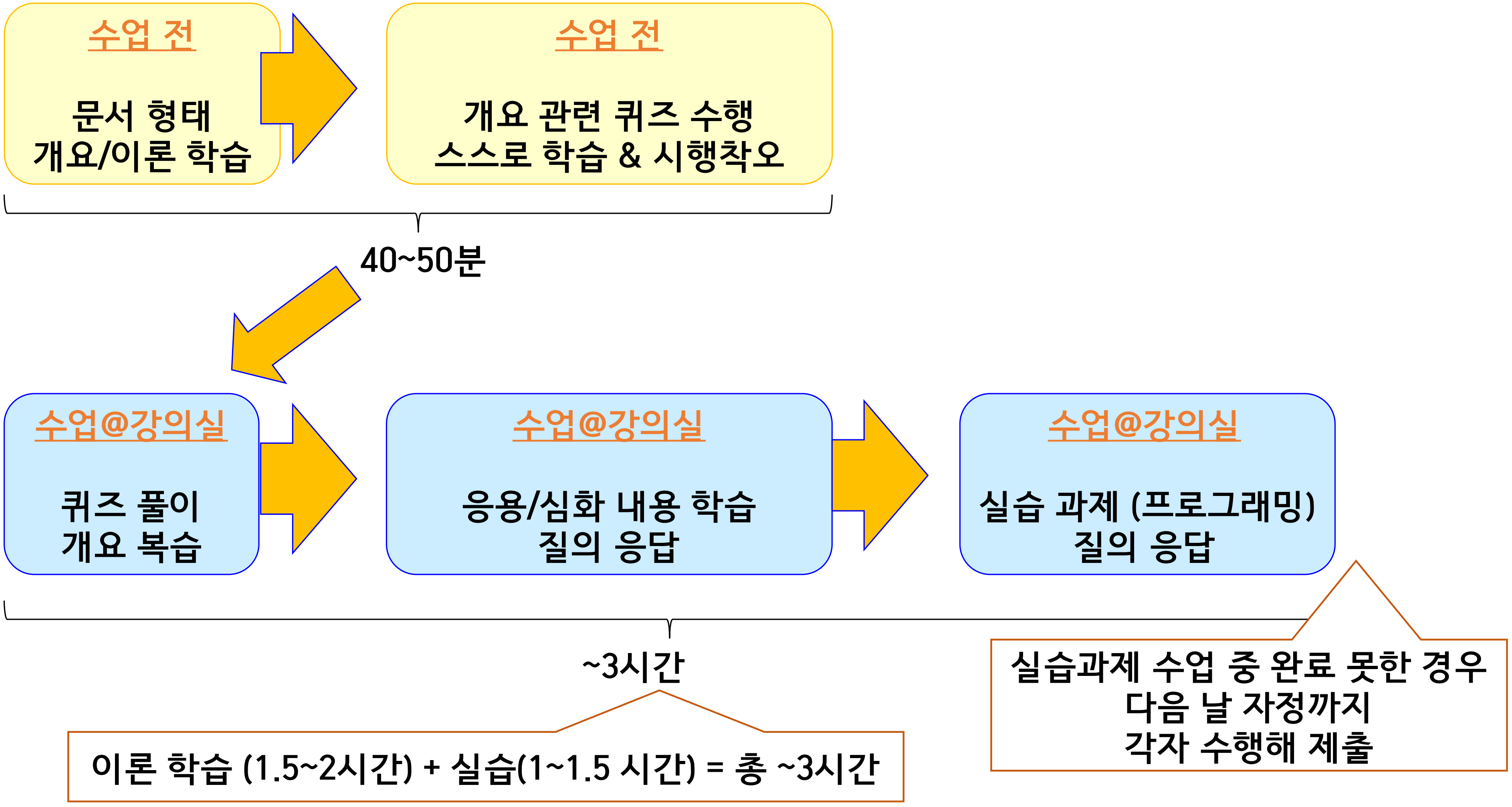
실습 과제 (프로그래밍)
질의 응답

~3시간

이론 학습 (1.5~2시간) + 실습(1~1.5 시간) = 총 ~3시간

실습과제 수업 중 완료 못한 경우
다음날 자정까지
각자 수행해 제출

출석 확인



수업 전

문서 형태
개요/이론 학습

수업 전

개요 관련 퀴즈 수행
스스로 학습 & 시행착오

40~50분

수업@강의실

퀴즈 풀이
개요 복습

수업@강의실

응용/심화 내용 학습
질의 응답

수업@강의실

실습 과제 (프로그래밍)
질의 응답

~3시간

이론 학습 (1.5~2시간) + 실습(1~1.5 시간) = 총 ~3시간

실습과제 수업 중 완료 못한 경우
다음 날 자정까지
각자 수행해 제출

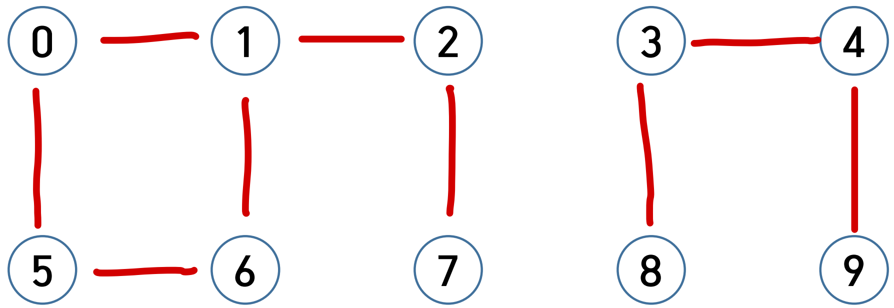
문제 정의: Union Find (연결 상태 변경 & 확인)

→ 연결수행

- N개 정점 주어짐
  - 0 ~ (N-1) 까지 정점(vertex)으로 표현
  - 간선(edge) 없는 상태에서 시작
- 2개의 명령 수행 필요
  - Union(a, b): 점 a와 b를 간선으로 연결 → 연결할때
  - Connected(a, b): a와 b 연결하는 경로 존재하는지 True/False로 응답 (이를 Find 명령이라고도 함) → 연결여부 확인.
- 목표: 이러한 명령 효율적으로 수행하는 알고리즘과 자료구조 설계

'connected' 관련 성질:
(1) connected(a, a) = True
(2) connected(a, b) = connected(b, a)
(3) connected(a, b) = True 이고
connected(b, c) = True이면,
connected(a, c) = True

- 예제 (N = 10)



union(4, 3)
union(3, 8)
union(6, 5)
union(9, 4)
union(2, 1)
connected(0, 7) · T
connected(8, 9) · T

union(5, 0)
union(7, 2)
union(6, 1)
union(1, 0)
connected(0, 7) · T

연결 상태가 **동적**으로 계속 변함. 따라서 이전에 했던 답 재활용하기 보다 그때그때 다시 답 확인 필요

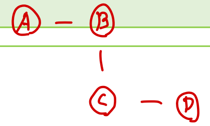
문제 정의: Union Find (연결 상태 **변경** & **확인**)

→ 두 노드가 연결됐나 확인

- N개 객체 주어짐
  - 0 ~ (N-1) 까지 정점(vertex)으로 표현
  - 간선(edge) 없는 상태에서 시작
- 2개의 명령 수행 필요
  - Union(a, b): 점 a와 b를 간선으로 연결
  - Connected(a, b): a와 b 연결하는 경로 존재하는지 True/False로 응답 (이를 Find 명령이라고도 함)
- 목표
  - 이러한 명령 수행하는 효율적 알고리즘 설계

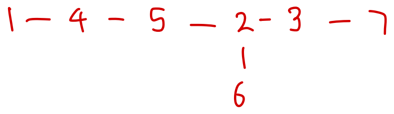
'connected' 관련 성질:
(1) connected(a, a) = True
(2) connected(a, b) = connected(b, a)
(3) connected(a, b) = True 이고
connected(b, c) = True이면,
connected(a, c) = True

ex)



∴ connected (a, d) = true

- 예제 (N = 8)



union(4, 1)
union(4, 5)
union(2, 3)
union(6, 2)
union(3, 6)
union(3, 7)
connected(1, 7) : F

union(5, 2)
connected(1, 7) : True
connected(0, 6) : False

[Q] 이러한 차례로 명령이 실행될 때, 각 connected 명령의 결과를 True/False로 답하시오.

Union - Find : 동적상황

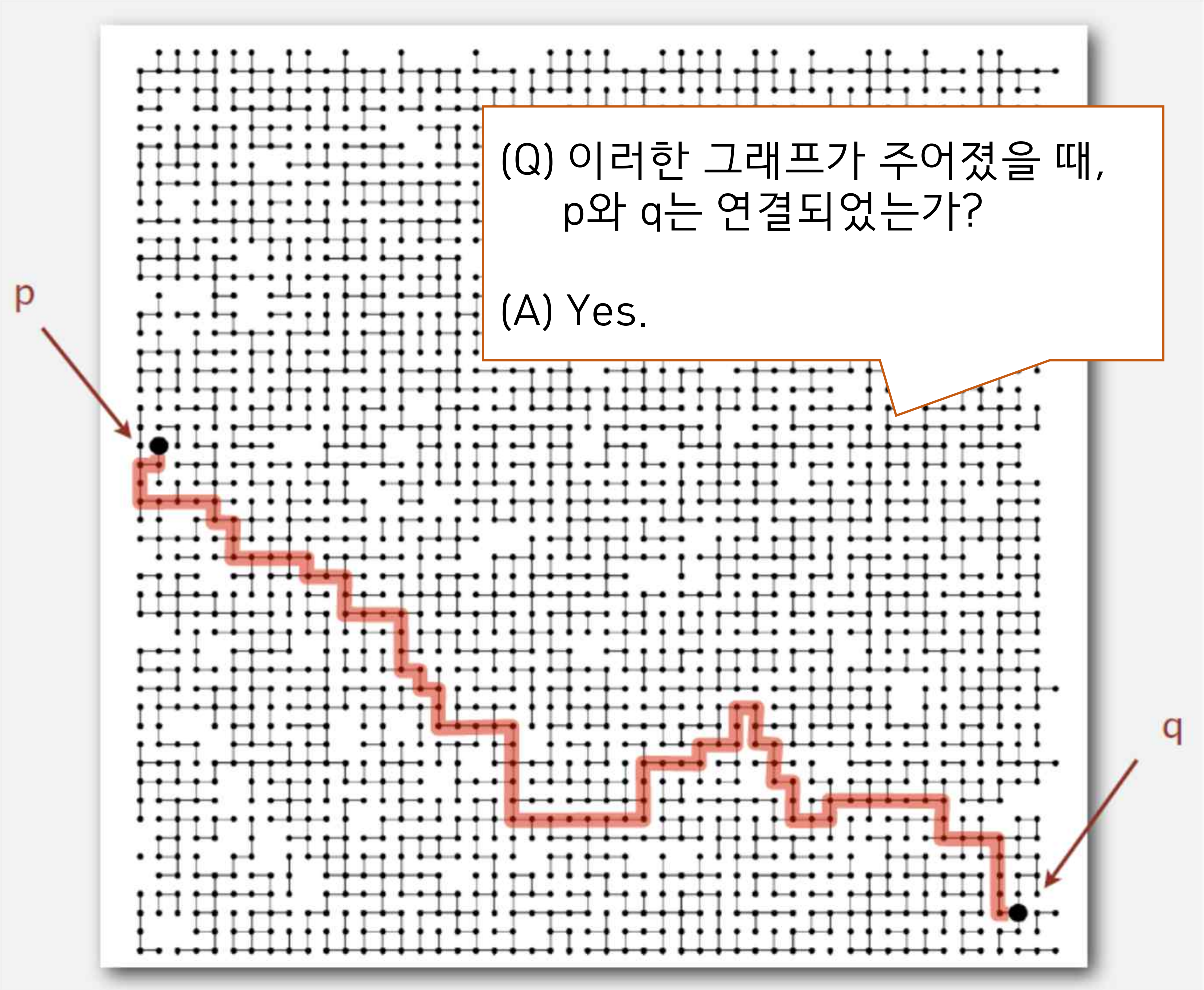
Union Find 문제의 해는 어디에 활용되는가?

→ 임의의 두 노드가 서로 연결이 되어 있는지.

- 그래프로 표현할 수 있는 경우 중
- 두 점 간 연결되었는지(**connectivity**) 확인하며 간선 계속 추가해보는 **동적 상황** (원하는 연결 상태 되도록 간선을 조금씩 더 해보는 상황)
- 연결 상태를 (다 받아오기 까지 시간 걸려서) 조금씩 받아오는 동시에 연결 상태 확인하는 상황 → 부모노드를 타고. 타고. 타고가는 방법을 사용하는것.

(유의사항)
- 그래프 전체가 미리 주어지고 그 형태가 변하지 않고 고정된 경우는 (**정적 상황**) 이번 시간과 다른 상황이며, 따라서 다른 알고리즘 사용
- 두 점 연결하는 최단 경로 찾는 것과 다른 문제이며, 이번 시간과 다른 알고리즘 사용

⇒ Union-Find는 노드가 실시간으로 추가되는 "동적상황"에 적합함.



(Q) 이러한 그래프가 주어졌을 때, p와 q는 연결되었는가?

(A) Yes.

Union Find 문제의 해는 어디에 활용되는가? Connectivity(연결상태) 확인

(Q) 컴퓨터망에서 두 장비가 연결되었는지 확인

(Q) 비행기 노선도가 주어졌을 때 임의의 두 지역이 서로 도달 가능한지 확인

(Q) Kruskal's minimum spanning tree algorithm의 일부로 활용

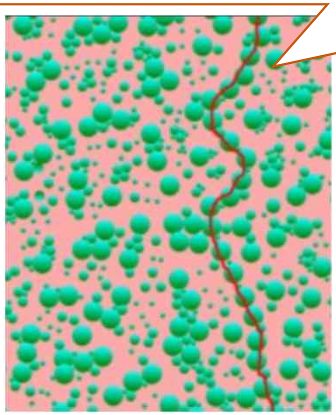
Minimum Spanning Tree 배울 때 이번 시간에 배운 자료구조 다시 활용

\>>> 이 외에도 많음 <<<

(Q) 픽셀로 이루어진 이미지에서 같은 물체를 구성하는 픽셀 확인

(Q) 플라스틱 판에 전도체(금속)를 뿌렸을 때 한쪽 끝에서 다른 쪽까지 연결되어 있는지 (따라서 전기가 흐르는지) 확인 (Percolation)

이번 시간 실습 과제



Union Find (Disjoint Set Forest)

Union Find 문제 정의, 활용도, 해결 방법 이해

01. 예습자료 & 퀴즈 주요 내용 복습

02. Union Find 문제 정의 및 활용

SUM

① connect(a,b) / union(a,b)

② "동적상황"에 적합

03. 첫 번째 방법: Quick-Find

04. 두 번째 방법: Quick-Union

05. Quick-Union의 개선

06. 실습: Percolation 문제에 Union-Find의 해(WQU) 적용

최종 목표: union과 connected 둘 다 효율적으로 수행할 수 있는 자료구조와 알고리즘 설계

	union	Connected (find)	공간
인접리스트	~1	~V + E	~V + E
QF		~1	~V
QU			
WQU			

union(a,b), connected(a,b) 수행하려면 그래프 연결 상태 저장 필요

- 어떤 자료구조에 저장해야 할까?

[Q] 일반적으로 그래프 저장에 많이 사용되는

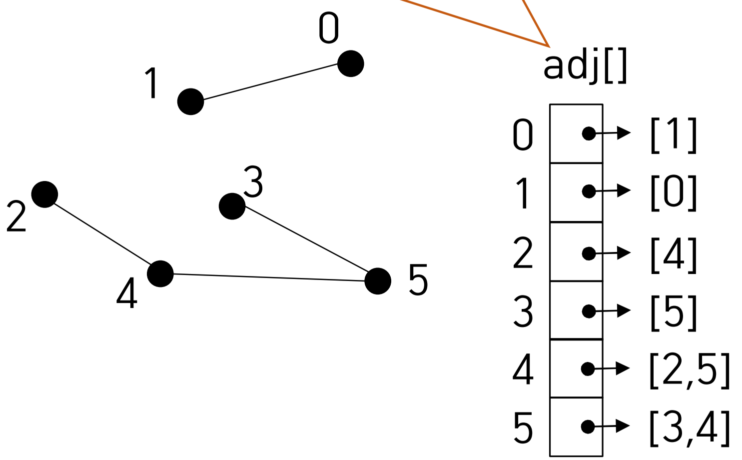
개별 간선 정보 저장하는 오른쪽 방식 생각해 보자.

(V x V 배열보다 compact)

union, connected는 어느 정도의 시간이 걸리는가?

V(Vertex)를 정점 수, E(Edge)를 간선 수라고 하자.

Adjacency list 방식:
adj[v]에 v에 인접한 정점 목록 저장



0	[1]
1	[0]
2	[4]
3	[5]
4	[2,5]
5	[3,4]

∴ O(V + E)

간선수

정점수

union(a,b), connected(a,b) 수행하려면 무엇을 어떻게 저장?

- 그래프 **연결 상태** 저장해야 함

[Q] 개별 간선 정보 저장하는 오른쪽의 기본 방식 생각해 보자.

union, connected는 빠른가?

[Q] 개별 간선 정보 꼭 저장 해야 하나?

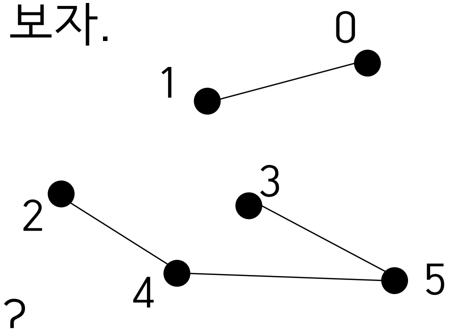
꼭 필요하지 않다면 더 간단한 방법 생각해 봐도 될까?

(개별 간선 정보 필요한 작업:

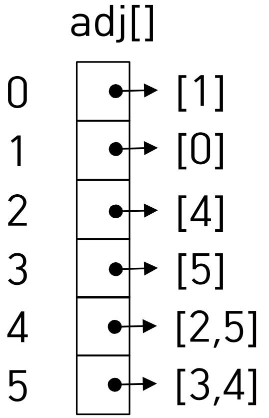
"(a, b) 사이 경로를 보이시오",

"(a, b) 사이에 간선 존재하나(둘을 직접 잇는 간선)",

"(a, b) 사이 간선 삭제하시오", …)



왼쪽 그래프의 개별 간선 정보를 저장한 예



첫 번째 해결책: 각 객체가 어느 연결된 덩어리(connected component)에 속하는지 계속 기록하고 업데이트

∴ set() = { } 의 형태 !

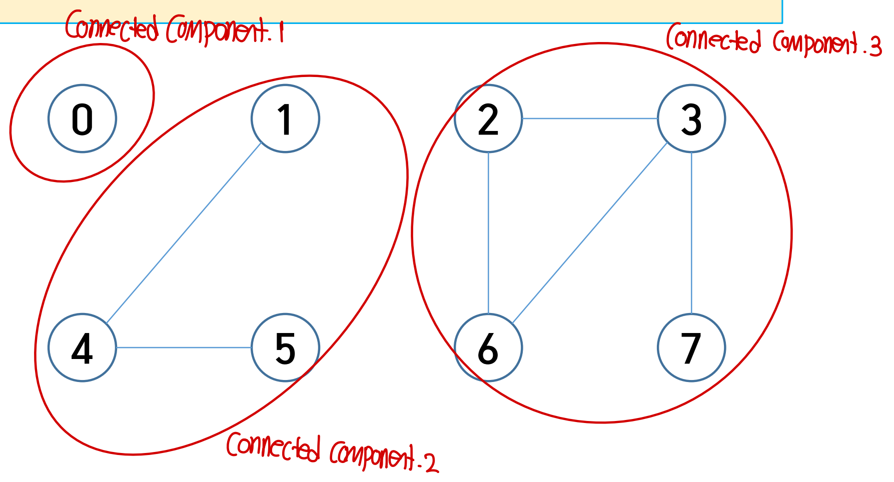
- Connected component: 서로 연결된 정점들의 maximal한 집합

'connected' 관련 성질:

(1) connected(a, a) = True

(2) connected(a, b) = connected(b, a)

(3) connected(a, b) = True 이고
connected(b, c) = True이면,
connected(a, c) = True



[Q] 몇 개의 connected components가 존재하는가? 이들은 각각 무엇인가?

⇒ 3개 component {0} {1. 4. 5} {2. 3. 6. 7}

[Q] {4, 5}는 connected component인가?

⇒ T

[Q] {2, 3, 6}은 connected component인가?

⇒ T

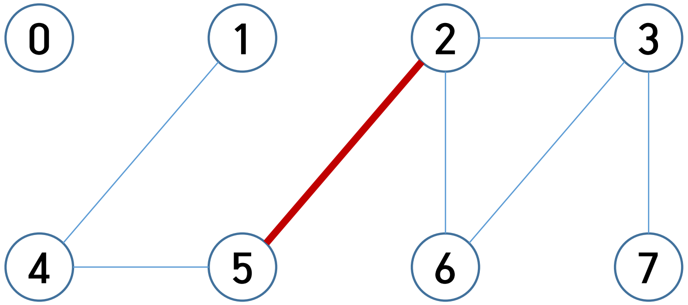
첫 번째 해결책: 각 객체가 어느 연결된 덩어리(connected component)에 속하는지 계속 기록하고 업데이트

- Connected component: 서로 연결된 객체들의 maximal한 집합

'connected' 관련 성질:
(1) connected(a,a) = True

(2) connected(a,b) = connected(b,a)

(3) connected(a,b) = True 이고
connected(b,c) = True이면,
connected(a,c) = True



[Q] 몇 개의 connected components가 존재하는가? 이들은 각각 무엇인가?

⇒ 2개 {0}, {1·2·3·4·5·6·7}

[Q] {1, 4, 5}는 connected component인가?

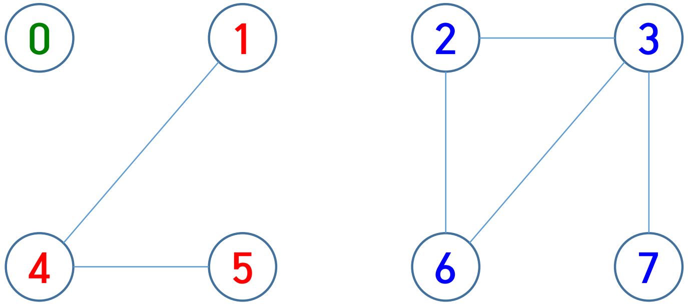
⇒ T

[Q] {2, 3, 6, 7}은 connected component인가?

⇒ T

첫 번째 해결책(Quick-Find): 각 객체가 어느 연결된 덩어리(connected component)에 속하는지 크기 N(정점개수)의 배열에 기록 & 업데이트

- 길이 N인 정수 배열 ids[] 사용해 각 객체가 어느 connected component에 속하는지 기록
- ids[i]: 객체 i가 속한 component의 id
- component의 id: 서로 다른 component에 서로 다른 숫자 부여. 오른쪽 예제에서는 component에 속한 객체 중 가장 작은 번호 사용



index:	0	1	2	3	4	5	6	7
ids[]	0	1	2	2	1	1	2	2

{1, 4, 5} 중 가장 작은 번호(1) 사용

But, 너무 많은 번호변경 발생. → 변경(union)시 O(N).
탐색(find)시 O(1).

Quick-find	Initialize	union	find
Time Complexity	O(n)	O(n)	O(1)

* 빅오는 Input 기준

· Initialize – 각 인덱스번호에 맞게 초기화

[0]	[1]	[2]	. . .	[N]
0	1	2	. . .	N

⇒ O(N)

· Union – 같은 set의 최소번호로 초기화

[0]	[1]	[2]	[3]	[4]	[5] ..	[N]
0	1	2	2	4	5	N

⇓

union (1,2)

[0]	[1]	[2]	[3]	[4]	[5] ..	[N]
0	1	1	1	4	5	N

합칠 노드 : 3개

⇒ O(N)

· find – 해당하는 하나의 인덱스에만 접근

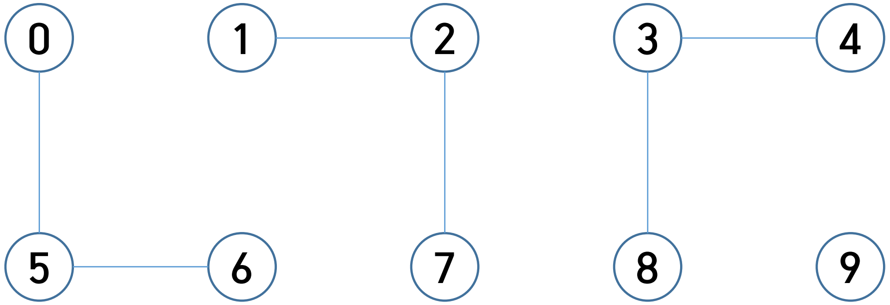
[0]	[1]	[2]	[3]	[4]	[5] ..	[N]
0	1	2	2	4	5	N

find (3)

⇒ O(1)

첫 번째 해결책(Quick-Find): 각 객체가 어느 연결된 덩어리(connected component)에 속하는지 크기 N의 배열에 기록 & 업데이트

- 길이 N인 정수 배열 ids[] 사용해 각 객체가 어느 connected component에 속하는지 기록
- ids[i]: 객체 i가 속한 component의 id
- component의 id: 서로 다른 component에 서로 다른 숫자 부여. 오른쪽 예제에서는 component에 속한 객체 중 가장 작은 번호 사용



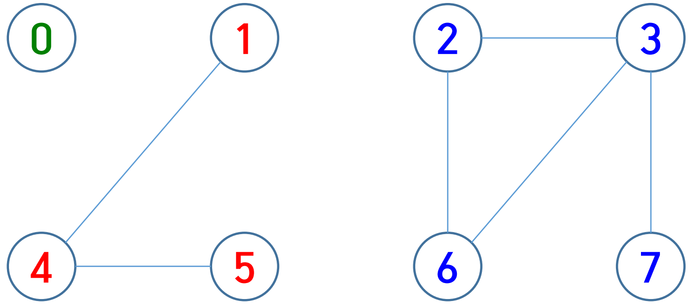
∴ Quick-Find 일때의 ids[] 상태

index:	0	1	2	3	4	5	6	7	8	9
ids[]	0	1	1	3	3	0	0	1	3	9

[Q] 배열 ids[]에 저장할 값을 써보시오.

첫 번째 해결책(Quick-Find): 각 객체가 어느 연결된 덩어리(connected component)에 속하는지 크기 N의 배열에 기록 & 업데이트

- 길이 N인 정수 배열 ids[] 사용해 각 객체가 어느 connected component에 속하는지 기록
- ids[i]: 객체 i가 속한 component의 id
- component의 id: 서로 다른 component에 서로 다른 숫자 부여. 오른쪽 예제에서는 component에 속한 객체 중 가장 작은 번호 사용



index:	0	1	2	3	4	5	6	7
ids[]	0	1	2	2	1	1	2	2

(Q) 왜 이렇게 저장하는가?

(A1) Connected(a, b)에 빠르게 답할 수 있음. How?
Connected(혹은 Find)에 빠르게 답할 수 있는 방법이므로 Quick-Find라 함 [Quick-find]

O(1)

initialize	union	find
O(N)	O(N)	O(1)

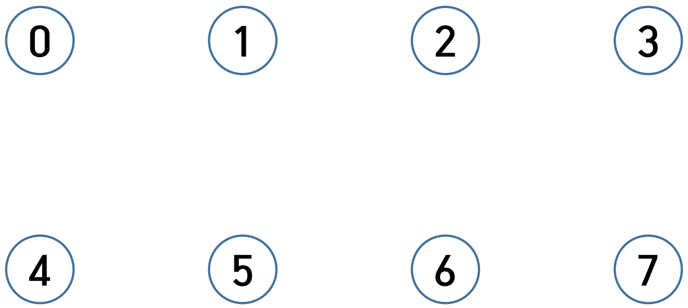
(A2) N × N 배열이나 adjacency-list 사용해 개별 간선 정보를 일일이 저장하지 않아도 됨. 오른쪽과 같이 우리가 필요한 정보는 1 × N 배열에 다 담을 수 있으므로

첫 번째 해결책(Quick-Find): 각 객체가 어느 연결된 덩어리(connected component)에 속하는지 크기 N의 배열에 기록 & 업데이트

- ids[i] = i로 초기화
  (자기 번호)
- connected(a, b): return (ids[a] == ids[b])
- union(a, b):
- ids[i] == ids[b]인 모든 i에 대해 ids[i] = ids[a]로 값 교체
- 혹은
- ids[i] == ids[a]인 모든 i에 대해 ids[i] = ids[b]로 값 교체

a와 같은 component에 있던 모두와
b와 같은 component에 있던 모두가
같은 component ID 가지도록 변경 필요

[Q] a, b의 id 중 더 작은 값 사용한다고
가정하고 ids[]의 변화 과정을 써보자.



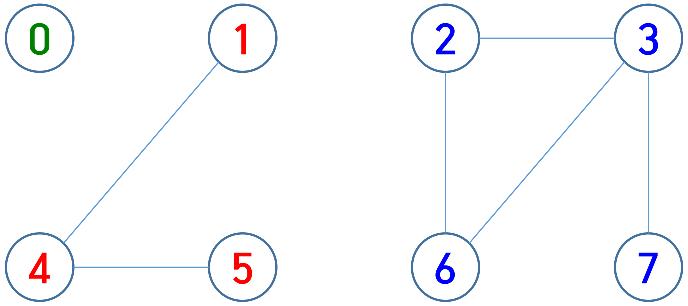
index:	0	1	2	3	4	5	6	7
ids[]	0	1	2	3	4	5	6	7
union(4,1)	0	1	2	3	1	5	6	7
union(4,5)	0	1	2	3	1	1	6	7
union(2,3)	0	1	2	2	1	1	6	7
union(6,2)	0	1	2	2	1	1	2	7
union(3,6)	0	1	2	2	1	1	2	7
union(3,7)	0	1	2	2	1	1	2	2

connected(1,7)

F $1 \neq 2$

첫 번째 해결책(Quick-Find): 각 객체가 어느 연결된 덩어리(connected component)에 속하는지 크기 N의 배열에 기록 & 업데이트

- ids[i] = i로 초기화
- connected(a,b): return (ids[a] == ids[b])
- union(a,b):
- ids[i] == ids[b]인 모든 i에 대해 ids[i] = ids[a]로 값 교체
- 혹은
- ids[i] == ids[a]인 모든 i에 대해 ids[i] = ids[b]로 값 교체



index:	0	1	2	3	4	5	6	7
ids[]	0	1	2	2	1	1	2	2

[Q] 배열 id[]에 저장할 값이 어떻게 변하며, 어떤 값을 사용해 답하는지 써보시오.

union(5,2)	0	1	1	1	1	1	1	1
connected(1,7)	T							
connected(0,6)	F							
union(0,3)	0	0	0	0	0	0	0	0

[Q] 이 방법의 단점은 무엇이라고 생각하는가? (유의: 값을 변경할 부분만 아니라 변경하지 않는 부분도 빠짐없이 확인 필요)

∴ Quick - Find

```
N = 8

ids = []
for idx in range(N):      ⇒ O(N)
    ids.append(idx)       # 정점 번호로 ids[] 초기화

def connected(p, q): # Connected (a,b)
    return ids[p] == ids[q]              Quick - Find ⇒ O(1)

def minMax(a, b): # 두 element 중 최소값 찾기.
    if a < b: return a, b
    else: return b, a

def union(p, q): # Union (a,b)
    id1, id2 = minMax(ids[p], ids[q])
    for idx, _ in enumerate(ids):                ⇒ O(N)
        if ids[idx] == id2: ids[idx] = id1
```

두 값 a, b 중 (더 작은 값, 더 큰 값) 반환

```
union(4,1)
union(4,5)
union(2,3)
union(6,2)
union(3,6)
union(3,7)
print(connected(1,7))
union(5,2)
print(connected(1,7))
print(connected(0,6))
union(0,3)
print(connected(0,6))
```

이번 시간 첨부파일 QuickFind.py 참조

첫 번째 해결책(Quick-Find)의 Cost Model

Algorithm	ids[] 초기화	union	find(connected)
Quick-Find	~N	~N	~1 (상수시간)

[Q] 각각이 왜 그러한지 생각해 보시오.

- 가장 많이 수행해야 하는
- union 명령을 N회 수행하려면
- N^2에 비례한 횟수의 메모리 접근 필요

```
N = 8

ids = []
for idx in range(N):
    ids.append(idx)

def connected(p, q):
    return ids[p] == ids[q]

def minMax(a, b):
    if a < b: return a, b
    else: return b, a

def union(p, q):
    id1, id2 = minMax(ids[p], ids[q])
    for idx, _ in enumerate(ids):
        if ids[idx] == id2: ids[idx] = id1
```

Union Find (Disjoint Set Forest)

Union Find 문제 정의, 활용도, 해결 방법 이해

01. 예습자료 & 퀴즈 주요 내용 복습

02. Union Find 문제 정의 및 활용

03. 첫 번째 방법: Quick-Find

04. 두 번째 방법: Quick-Union

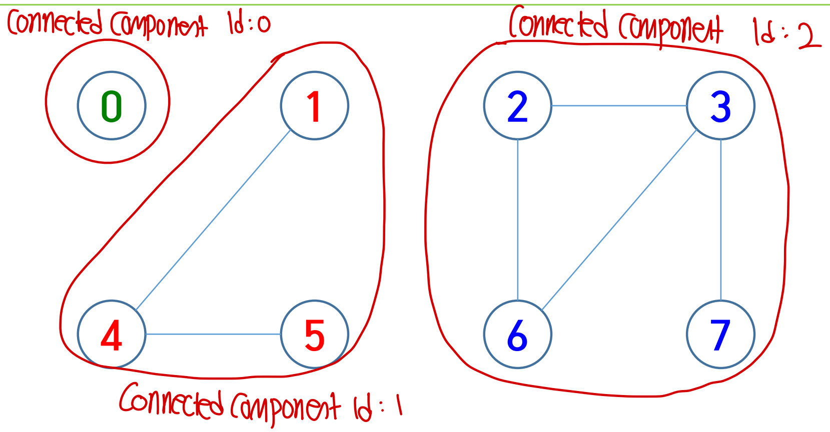
05. Quick-Union의 개선

06. 실습: Percolation 문제에 Union-Find의 해(WQU) 적용

QF(Quick-Find)에서 사용하던 구조의 다른 해석

· Quick-Find : ids 설정시 connected component 의 최소값

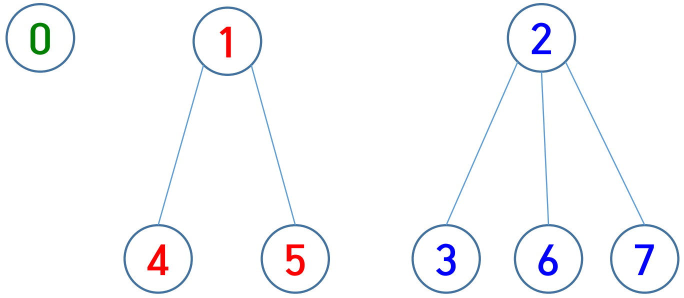
- connected component를 한 덩어리로 봄
- ids[i]: 객체 i가 연결된 component의 id
- connected(p,q) == True if ids[p] == ids[q]



index:	0	1	2	3	4	5	6	7
ids[]	0	1	2	2	1	1	2	2

· Quick-union : ids 설정시 parent node, ids[i] == i 일시 해당 i는 root nod

- connected component를 **tree**로 봄
- ids[i]: 객체 i의 **parent** (root가 아닌 parent!)
- 만약 객체 i가 root라면 ids[i] = i
- connected(p, q) == True if **root(p) == root(q)**



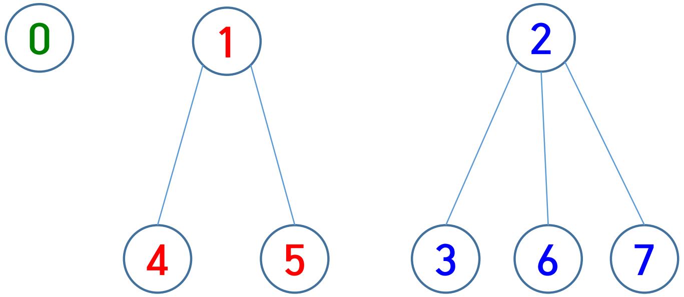
index:	0	1	2	3	4	5	6	7
ids[]	0	1	2	2	1	1	2	2

QF(Quick-Find)에서 사용하던 구조의 다른 해석

- connected component를 **tree**로 봄
- ids[i]: 객체 i의 **parent**
- 만약 객체 i가 root라면 ids[i] = i
- connected(p, q) == True if **root(p) == root(q)**

- **union(p, q): p가 속한 tree 상의 모든 node를 q가 속한 tree 아래로 옮겨 붙이기**

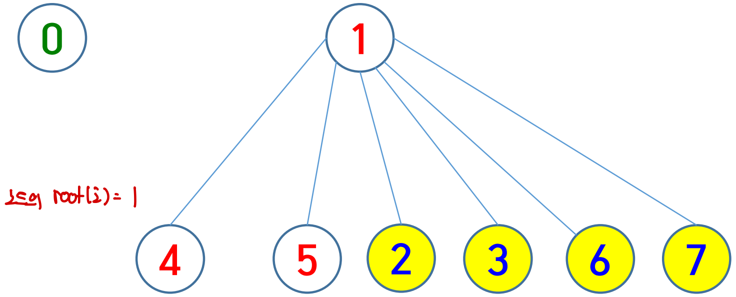
즉 하나의 root 아래로 모두 옮겨 붙임



union(2, 1)

· union(2,1) ⇒ root(i)가 2인 노드의 root(i) = 1

· root(p) 연산이 중요.



index:	0	1	2	3	4	5	6	7
ids[]	0	1	2	2	1	1	2	2

index:	0	1	2	3	4	5	6	7
ids[]	0	1	1	1	1	1	1	1

여러 객체가 위치 이동

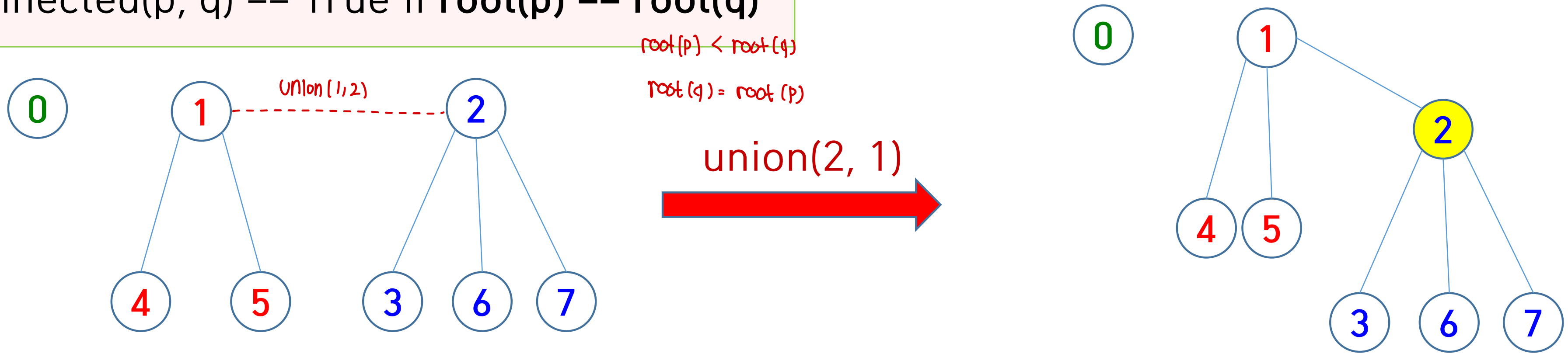
QU(Quick-Union): union할 때 모든 객체 아닌 root만 옮겨 붙임으로써 속도 향상

- connected component를 **tree**로 봄
- ids[i]: 객체 i의 **parent**
- 만약 객체 i가 root라면 ids[i] = i
- connected(p, q) == True if **root(p) == root(q)**

- **union(p, q): root(p)를 root(q) 아래로 옮겨 붙이기**

root(p) < root(q)

root(q) = root(p)



index:	0	1	2	3	4	5	6	7
ids[]	0	1	2	2	1	1	2	2

index:	0	1	2	3	4	5	6	7
ids[]	0	1	1	2	1	1	2	2

한 객체만 위치 이동

union은 빨라짐. 그런데 connected는 제대로 동작하나?

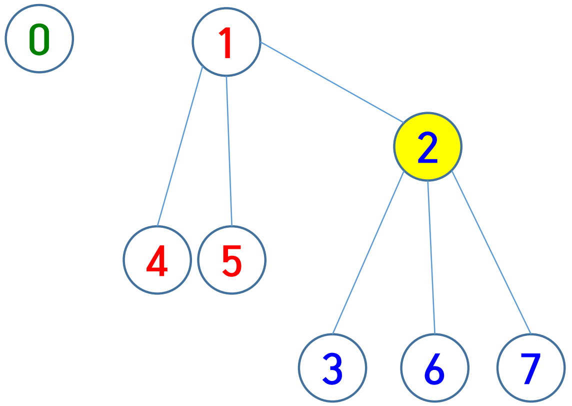
QU(Quick-Union): connected 답할 때는 root끼리 비교

- connected component를 **tree**로 봄
- ids[i]: 객체 i의 **parent**
- 만약 객체 i가 root라면 ids[i] = i
- connected(p, q) == True if **root(p) == root(q)**
- **root(i) = ids[ids[ids[… ids[i] …]]]**

- **union(p, q): root(p)를 root(q) 아래로 옮겨 붙이기**

[Q] connected(1, 2)에 답하는 과정을 보이시오.

[Q] connected(3, 4)에 답하는 과정을 보이시오.



index:	0	1	2	3	4	5	6	7
ids[]	0	1	1	2	1	1	2	2

QU(Quick-Union) 수행 예

- connected component를 tree로 봄
- ids[i]: 객체 i의 parent
- 만약 객체 i가 root라면 ids[i] = i
- connected(p, q) == True **if root(p) == root(q)**
- root(i) = ids[ids[ids[··· ids[i] ···]]]
- union(p, q): **root(p)를 root(q) 아래로** 붙이기

N=10

union(6, 5)

union(5, 0)

union(2, 1)

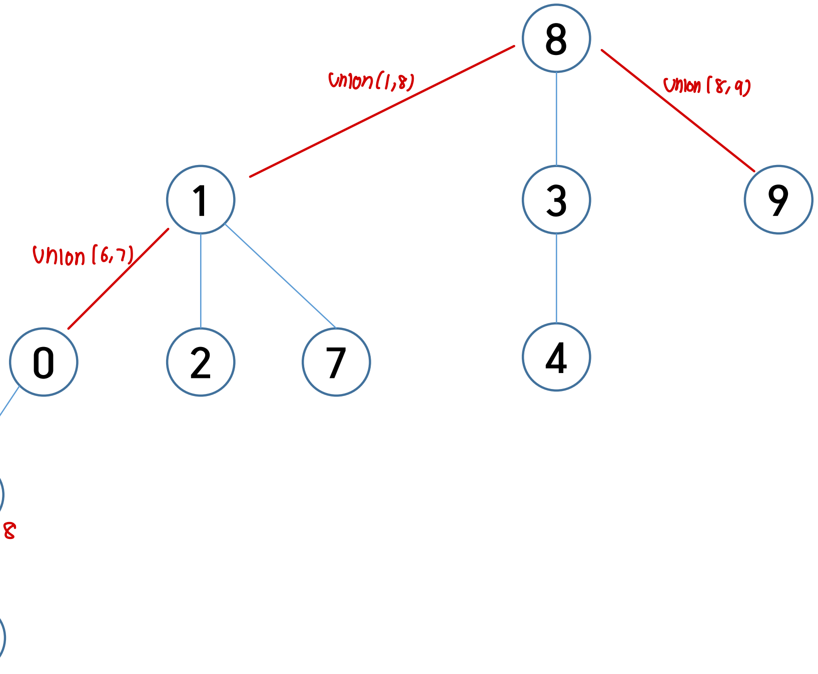
union(7, 1)

union(4, 3)

union(4, 8)

QU(Quick-Union) 수행 예

- connected component를 tree로 봄
- ids[i]: 객체 i의 parent
- 만약 객체 i가 root라면 ids[i] = i
- connected(p, q) == True **if root(p) == root(q)**
- root(i) = ids[ids[ids[··· ids[i] ···]]]
- union(p, q): **root(p)를 root(q) 아래로** 붙이기



root(6) == 0 , root(7) == 1 ⇒ ids[1] = 0

union(6, 7)

root(9) == 9, root(8) == 8 ⇒ ids[9] = 8

union(9, 8)

root(7) == 1, root(3) == 8 ⇒ ids[8] = 1

union(7, 3)

connected(5, 4) True

connected(7, 9) True

[Q] 배열 ids[]에 저장할 값을 써보시오.

index:	0	1	2	3	4	5	6	7	8	9
ids[]	1	8	1	8	3	0	5	1	8	8

parent node 저장

```
N = 10

ids = []
for idx in range(N):   # ids 초기화
    ids.append(idx)

def root(i):
    while i != ids[i]: i = ids[i]
    return i

def connected(p, q):
    return root(p) == root(q)

def union(p, q):
    id1, id2 = root(p), root(q)
    ids[id1] = id2
```

root에 도달할 때까지 parent 따라 올라가기

p와 q가 같은 root 가졌는지 확인

root(p)를 root(q) 아래로 연결

[Q] Quick-Find의 union() 함수에 비해 Quick-Union의 union() 함수는 for loop이 없어 간단해 보인다. 더 빠르다고 할 수 있는가?

이번 시간 첨부파일 QuickUnion.py 참조

```
union(6,5)
print(ids)
union(5,0)
print(ids)
union(2,1)
print(ids)
union(7,1)
print(ids)
union(4,3)
print(ids)
union(4,8)
print(ids)
union(6,7)
print(ids)
union(9,8)
print(ids)
union(7,3)
print(ids)
print(connected(5,4))
print(connected(7,9))
```

Quick-Find와 Quick-Union의 Cost Model 비교 (Worst Case)

Algorithm	Quick-Find	Quick-Union
ids[] 초기화	~N	~N
union	~N	~N
find(connected)	1 (상수시간)	~N

Tree가 flat하므로 find는 빠름. 하지만 flat하게 유지하기 위해 union 시간이 오래 걸림

Tree가 tall해지면(depth 깊어지면) find, union 모두 오래 걸림

```
N = 10

ids = []
for idx in range(N):
    ids.append(idx)

def root(i):
    while i != ids[i]: i = ids[i]
    return i

def connected(p, q):
    return root(p) == root(q)

def union(p, q):
    id1, id2 = root(p), root(q)
    ids[id1] = id2
```

Quick - Find

union : 그룹의 모든 노드 루트 변경 (slow) - O(N)

Find : ids 값만 비교 (Fast) - O(1)

Quick - union

union : 해당하는 root만 합쳐줌 (fast) - O(N)

Find : Parent node를 타고 타고 타고 root 도착시 비교. (slow) - O(N)

Union Find (Disjoint Set Forest)

Union Find 문제 정의, 활용도, 해결 방법 이해

01. 예습자료 & 퀴즈 주요 내용 복습

02. Union Find 문제 정의 및 활용

03. 첫 번째 방법: Quick-Find

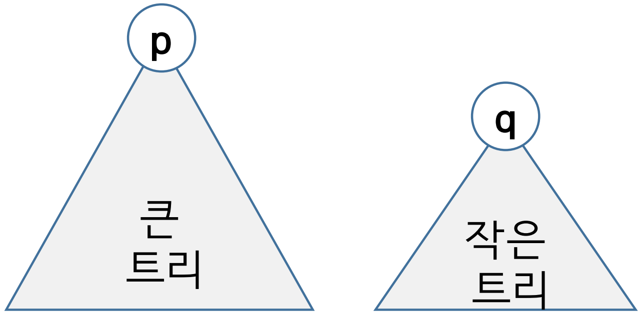
04. 두 번째 방법: Quick-Union

05. Quick-Union의 개선

06. 실습: Percolation 문제에 Union-Find의 해(WQU) 적용

앞으로 배울 방법에서 “트리의 크기”를 결정하는 기준과 의미

- 트리의 크기: 트리에 속한 **객체 수**
- 앞으로 배울 WQU 방법 사용하면 거의 트리의 크기가 depth를 반영하게 되어
- 더 큰 트리일수록 **depth**도 더 깊다고 보면 됨



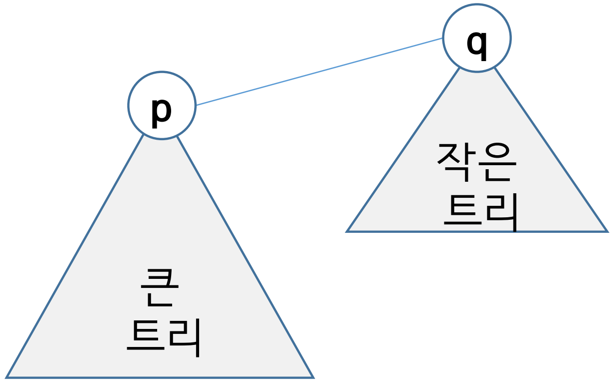
QU(Quick-Union)에서 트리 깊이 제한하기 위한 방법: Weighted QU

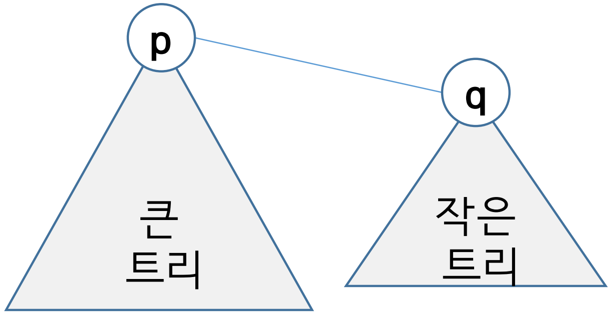
- QU(Quick-Union)
- union(p, q): root(p)를 root(q) 아래 연결

- Weighted QU

∴ Quick-union → weighted Quick-union
무지성으로 root를 이동 → 작은 트리를 큰 트리에 부착.

- **union(p, q): 작은 트리의 root를 큰 트리의 root 아래 연결**
- 이를 위해 tree의 size도 기록 (tree에 속한 객체 수)





[Q] Union 후 Tree의 최대 depth가 몇 증가하는가?

[Q] Union 후 Tree의 최대 depth가 몇 증가하는가?

[Q] Union 후 Tree의 최대 depth가 증가하는 경우는 어떤 경우인가?

Weighted QU(Quick-Union) 수행 예

- connected component를 tree로 봄
- ids[i]: 객체 i의 parent
- 만약 객체 i가 root라면 ids[i] = i
- connected(p,q) == True if root(p) == root(q)
- root(i) = ids[ids[ids[… ids[i] …]]]
- **union(p, q): 작은 트리의 root를 큰 트리의 root 아래 연결**
- **트리의 크기는 객체 수**

두 트리 크기 같다면 root(p)를 root(q) 아래에 연결

N=10

union(6, 5)

union(5, 0)

union(2, 1)

union(7, 1)

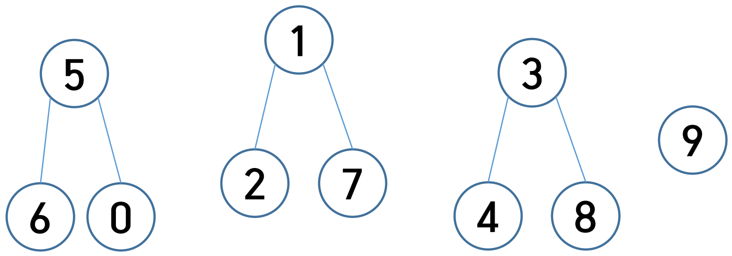
union(4, 3)

union(4, 8)

Weighted QU(Quick-Union) 수행 예

- connected component를 tree로 봄
- ids[i]: 객체 i의 parent
- 만약 객체 i가 root라면 ids[i] = i
- connected(p,q) == True if root(p) == root(q)
- root(i) = ids[ids[ids[… ids[i] …]]]
- **union(p, q): 작은 트리의 root를 큰 트리의 root 아래 연결**
- **트리의 크기는 객체 수**

두 트리 크기 같다면 root(p)를 root(q) 아래에 연결



union(6, 7)

union(9, 8)

union(7, 3)

connected(5, 4)

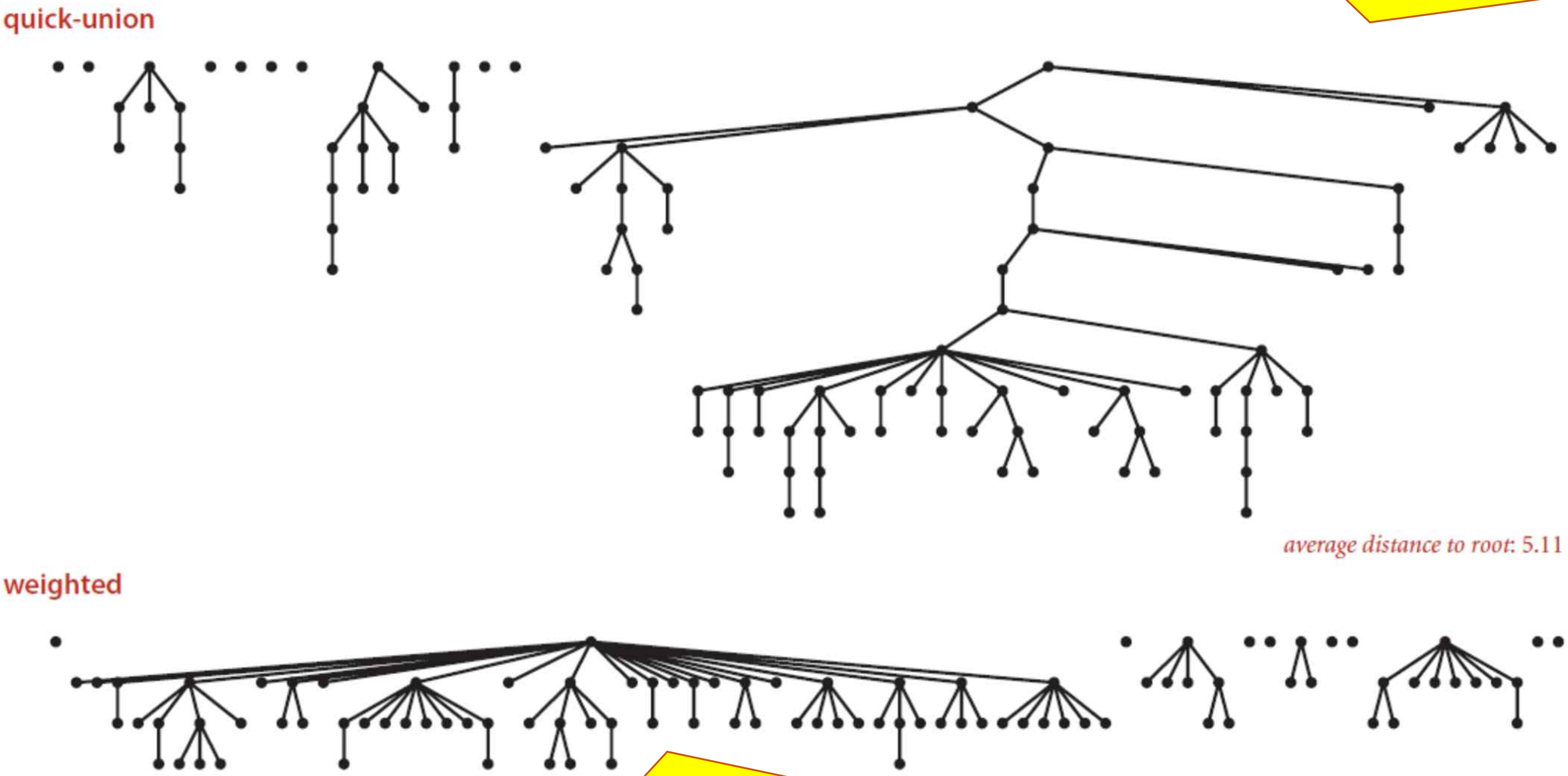
connected(7, 9)

[Q] 이 트리의 최대 깊이는 얼마인가? QU를 사용한 경우와 비교해 보시오.

[Q] 배열 ids[]에 저장할 값을 써보시오.

index:	0	1	2	3	4	5	6	7	8	9
ids[]										

"Union Find" 문제를 "Disjoint Set Forest"라고도 하는 이유



모든 객체가 root에 가까움을 유의해 보세요.

Quick-union and weighted quick-union (100 sites, 88 `union()` operations)

QU(Quick-Union)

```
N = 10

ids = []
for idx in range(N):
    ids.append(idx)



def root(i):
    while i != ids[i]: i = ids[i]
    return i

def connected(p, q):
    return root(p) == root(q)

def union(p, q):
    id1, id2 = root(p), root(q)
    ids[id1] = id2
```

Weighted QU

```
N = 10

ids = []
size = []    # size[i]: size of tree rooted at i
for idx in range(N):
    ids.append(idx)
    size.append(1)

def root(i):
    while i != ids[i]: i = ids[i]
    return i

def connected(p, q):
    return root(p) == root(q)

def union(p, q):
    id1, id2 = root(p), root(q)
    if id1 == id2: return
    if size[id1] <= size[id2]:
        ids[id1] = id2
        size[id2] += size[id1]
    else:
        ids[id2] = id1
        size[id1] += size[id2]
```

각자가 속한 트리크기 저장

각 객체 자신을 root로 하는 tree의 크기 저장하는 배열

p가 속한 트리의 사이즈가 작은 경우

q가 속한 트리의 사이즈가 작은 경우

속도가 빠른 방법일수록 저장공간을 더 사용하는 경우 많으나, WQU는 여전히 N에 비례한 공간 사용

이번 시간 첨부파일 WeightedQuickUnion.py 참조

Weighted QU

```
N = 10

ids = []
size = []   # size[i]: size of tree rooted at i
for idx in range(N):
    ids.append(idx)
    size.append(1)

def root(i):
    while i != ids[i]: i = ids[i]
    return i

def connected(p, q):
    return root(p) == root(q)

def union(p, q):
    id1, id2 = root(p), root(q)
    if id1 == id2: return
    if size[id1] <= size[id2]:
        ids[id1] = id2
        size[id2] += size[id1]
    else:
        ids[id2] = id1
        size[id1] += size[id2]
```

각 객체 자신을 root로 하는 tree의 크기 저장하는 배열

p가 속한 트리의 사이즈가 작은 경우

q가 속한 트리의 사이즈가 작은 경우

[Q] union 하면서 size를 갱신할 때, 새로운 root의 size만 갱신하는 이유는 무엇인가? 이 트리에 속한 다른 노드의 size는 갱신하지 않아도 괜찮나?

QU: 최대 깊이 ~N
Weighted QU(Quick-Union): 어떤 객체 x의 깊이도 $\leq \log_2(N)$ 으로 제한됨

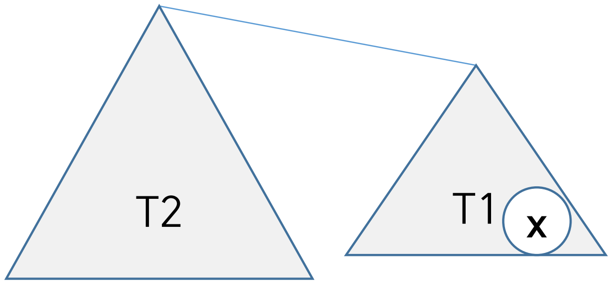
- 증명:
- N개 객체 중 임의의 정점을 x라 하자.
- x의 깊이가 +1 될 때는 x가 속한 트리가 (작아서) 더 큰 트리에 연결될 때
- 이 때 x가 속한 tree의 크기는 최소 2배가 됨 (그림 참조)

크기 : 자기 tree node 개수

작은거 크기 : N
큰거 크기 : $N+k$
2N+k : 최소 2배

- 그런데 이렇게 크기가 2배가 되는 것은 많아봐야 $\log_2(N)$회 → 펴지니까?
- x의 깊이가 k번 +1된다고 가정하면,
- x가 속한 트리의 크기는 최소 2^k
- 전체 그래프에 N개의 객체만 있으므로 $2^k \leq N$
- 따라서 $k \leq \log_2(N)$

트리 내 임의 정점 x 입장에서 볼 때, 깊이 증가 횟수는 $\log_2(N)$ 넘을 수 없음 보임



Quick-Find, Quick-Union, Weighted QU의 Cost Model 비교

Algorithm	Quick-Find	Quick-Union	Weighted QU
ids[] 초기화	~N	~N	~N
union	~N	~N	~$\log_2$(N)
find(connected)	1 (상수시간)	~N	~$\log_2$(N)

Tree가 flat하므로 find는 빠름. 하지만 flat하게 유지하기 위해 union 시간이 오래 걸림

Tree가 tall해지면 (depth 깊어지면) find, union 모두 오래 걸림

root에 도달하는 시간을 $\log_2$(N)으로 제한
따라서 find, union 모두 $\log_2$(N)으로 제한

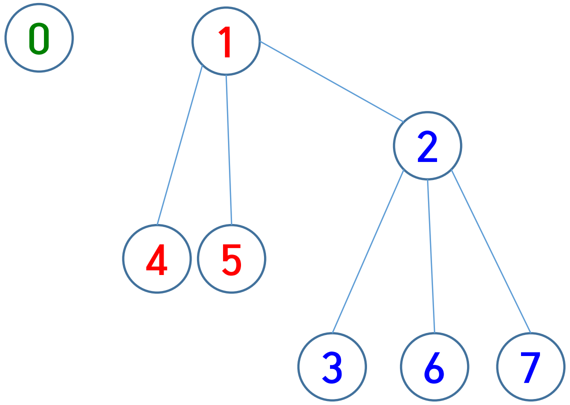
[Q] WQU와 QF를 비교하면 어느 쪽이 더 빠른가?

예: 10^9개 객체에 대해 10^9번의 union 수행?

```
ids = []
   []
    in range(N):
   .append(idx)
   e.append(1)

   t(i):
    while i != ids[i]: i = ids[i]
    return i

def connected(p, q):
    return root(p) == root(q)

def union(p, q):
    id1, id2 = root(p), root(q)
    if id1 == id2: return
    if size[id1] <= size[id2]:
        ids[id1] = id2
        size[id2] += size[id1]
    else:
        ids[id2] = id1
        size[id1] += size[id2]
```

정리: 문제 풀이에 필요한 정보만 저장 & 문제 풀이에 적합한 구조로 생각

- 그래프 연결 상태 저장 위해 일반적인 방식 (N x N 배열 혹은 adjacency-list에 연결 상태 저장) 대신 문제에 적합한 더 최적화된 자료구조 사용
- 서로 연결된 객체들을 묶어 'connected component' 혹은 'tree' 형태 구조로 생각
- 1차원 배열에 저장
- **자료구조**는 좋은 **알고리즘** 만드는데 중요한 역할



index:	0	1	2	3	4	5	6	7
ids[]	0	1	1	2	1	1	2	2

정리: 알고리즘 설계 과정

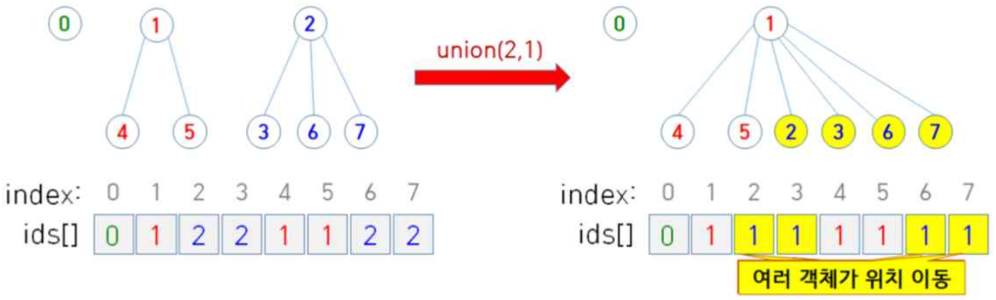
- 첫 알고리즘 고안
- 성능 예측 and 부족한 부분 있으면 이유 파악
- 문제점 해결 위한 방법 고안
- 위 단계 반복

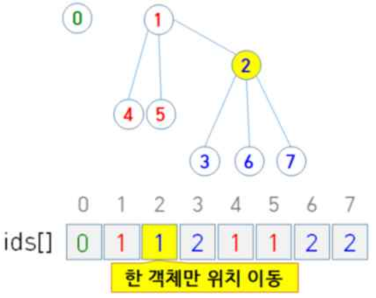
Algorithm	Quick-Find	Quick-Union	Weighted QU
ids[] 초기화	~N	~N	**~N**
union	~N	~N	~$\log_2(N)$
find(connected)	1 (상수시간)	~N	~$\log_2(N)$

Tree가 flat하므로 find는 빠름. 하지만 flat하게 유지하기 위해 **union 시 여러 객체를 옮겨야 하므로** 시간이 오래 걸림

union 시 한 객체만 옮김
Tree가 tall해지면
(depth 깊어지면)
find, union 모두 오래 걸림

작은 트리를 큰 트리 아래 연결함으로써 depth를 $\log_2(N)$으로 제한!





Union Find (Disjoint Set Forest)

Union Find 문제 정의, 활용도, 해결 방법 이해

01. 예습자료 & 퀴즈 주요 내용 복습

02. Union Find 문제 정의 및 활용

03. 첫 번째 방법: Quick-Find

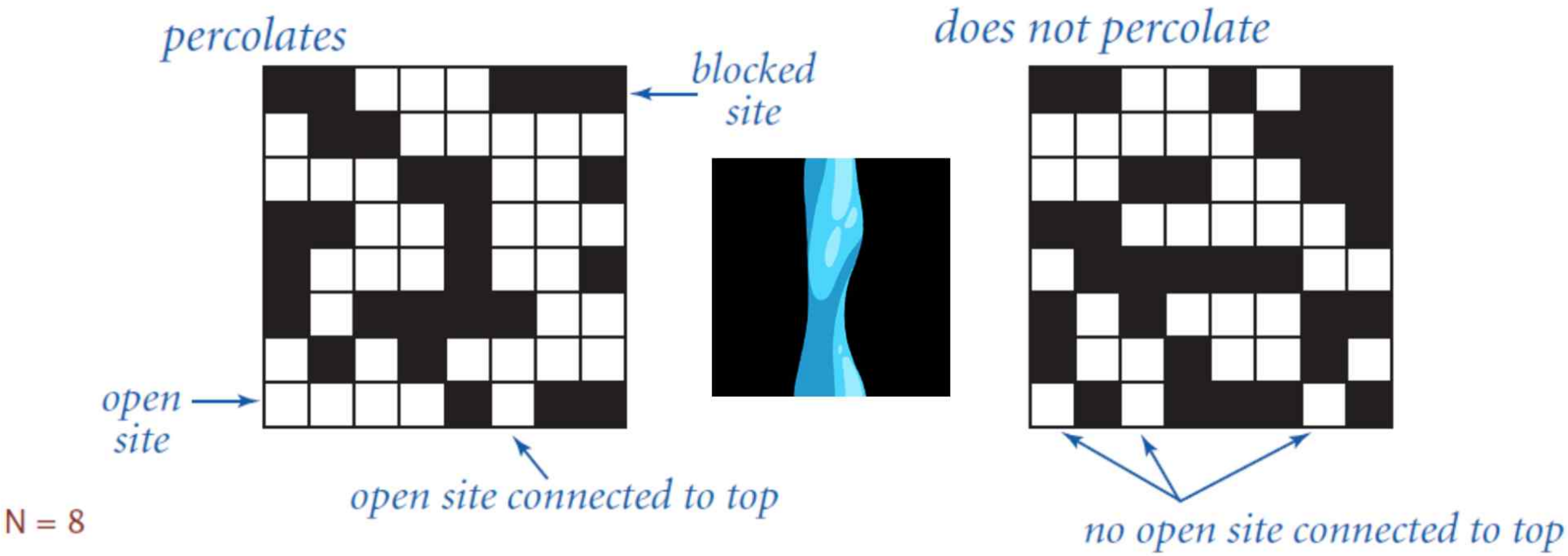
04. 두 번째 방법: Quick-Union

05. Quick-Union의 개선

06. 실습: Percolation 문제에 Union-Find의 해(WQU) 적용

N × N 격자가 "Percolate": 윗줄 → 아랫줄 가는 경로 존재

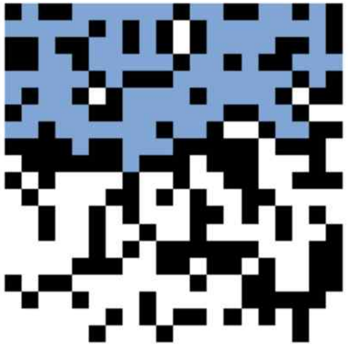
- **N × N** 개의 객체가 **격자**를 이룸 (그림 참조)
- 각 객체는 두 상태(**열림, 닫힘**) 중 하나를 가질 수 있으며
- 가장 윗줄이 가장 아랫줄에 연결되었다면 (인접한 **열린 격자 통해 이동**) 이 격자는 **percolate** 한다고 함



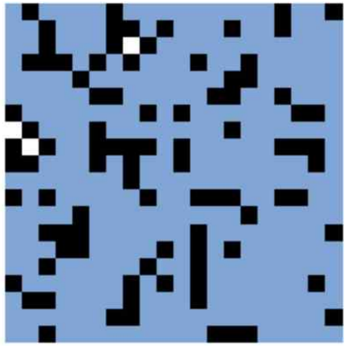
Percolation 문제 정의: 열린 격자 비율이 어느 값 이상일때, 거의 항상 percolate?

- **p: 열린 격자의 비율(퍼센티지)**
- 다음 질문에 답하고자 함: p가 클수록 percolate할 가능성 높아질 텐데, 평균적으로 어떤 p값 이상일 때 거의 항상 percolate하는가?

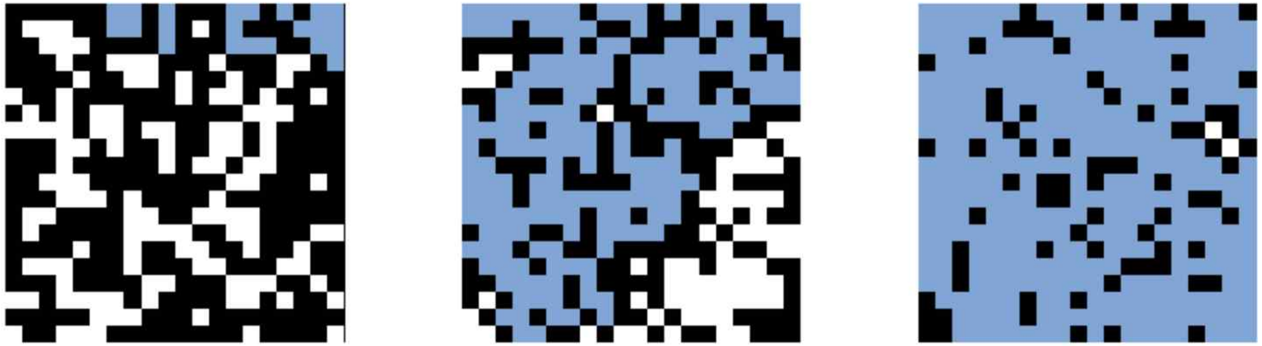
p low (0.4)
does not percolate

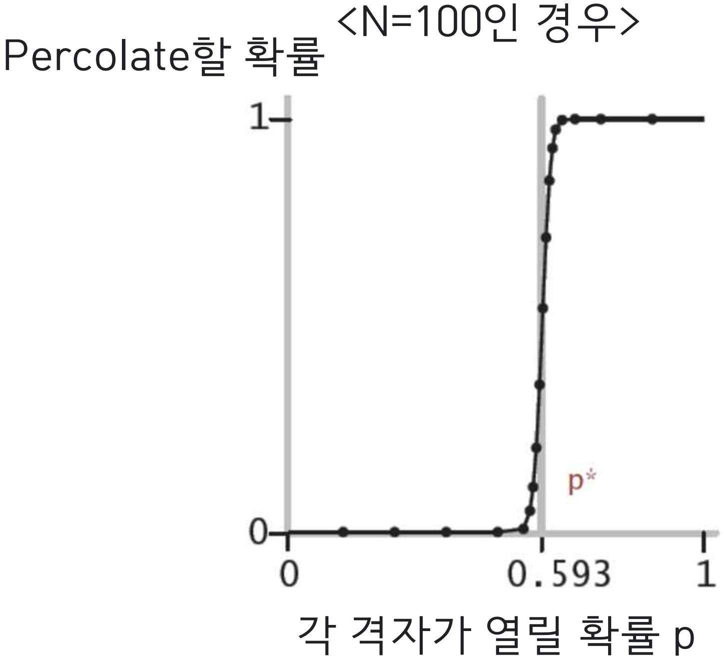


p medium (0.6)
percolates?



p high (0.8)
percolates

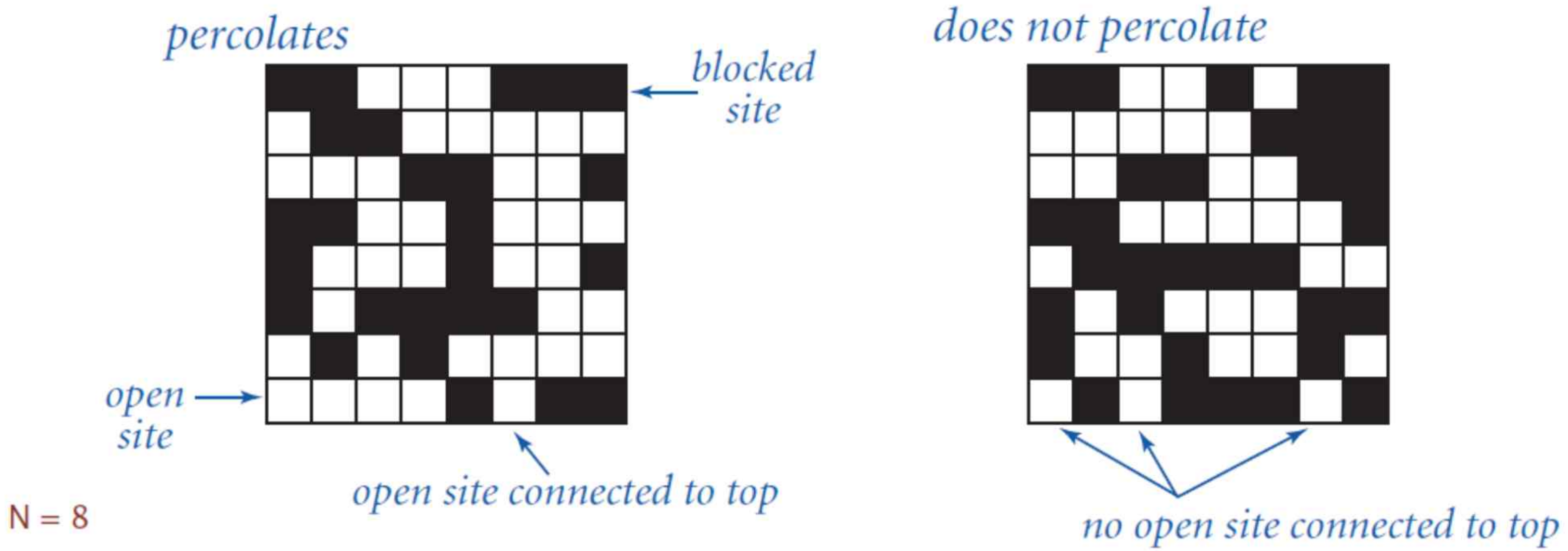




이 임계치를 수학으로 알아내기 어려워 **컴퓨터 시뮬레이션** 수행해 알아내며, 이를 우리가 수행해 봄

Percolation 문제 활용 예

- (비전도체인) 평면에 전도체(금속)를 뿌려 위→아래 방향으로 전기가 흐르도록 하려면 최소한 평면의 몇 퍼센트 정도에 금속을 배포해야 할까?
- (물이 흐르지 않는) 재료의 일부에 미세한 구멍을 내어 위→아래 방향으로 물이 흐르도록 하려면 최소한 몇 퍼센트 정도에 구멍을 내야 할까?
- 전기(혹은 물)을 가스나 SNS 사용자 간의 연결 상태 등 다른 다양한 경우로 바꾼 경우 모두 Percolation 문제에 대응되며, 유사한 방법으로 풀이 가능



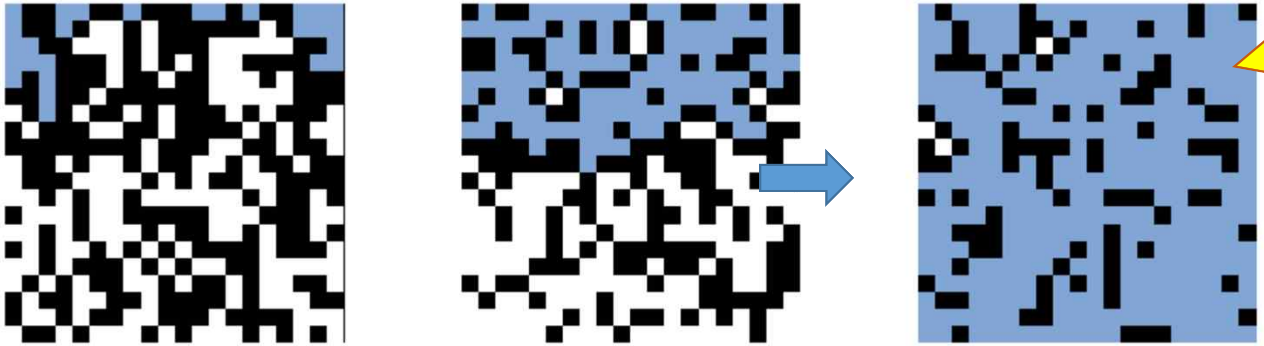
Percolation 문제 해결을 위한 simulation 방법 개요

- N × N 개의 객체를 **닫힌 상태로 초기화**

- **닫힌 객체 중 하나를 (임의로 선정해) 열린 상태로 바꾸고** percolate 하는지 확인
- **위 → 아래로 percolate 할 때까지 반복**

- percolate 할 때 열려 있는 객체의 비율(=열린 객체 수 / (N × N))을 p의 예측치로 사용

- 위와 같은 시뮬레이션을 여러 회 반복해 p의 평균 혹은 신뢰 구간 구하기



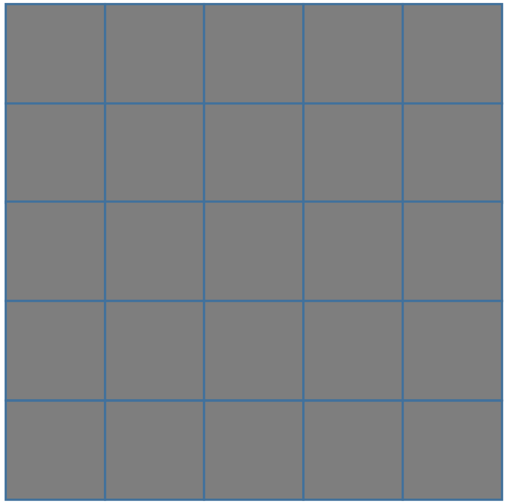
하나씩 임의로 선정해 열다가 percolate하는 순간 열린 격자의 비율 구하기
이를 반복한 후 수집한 값의 평균 내기

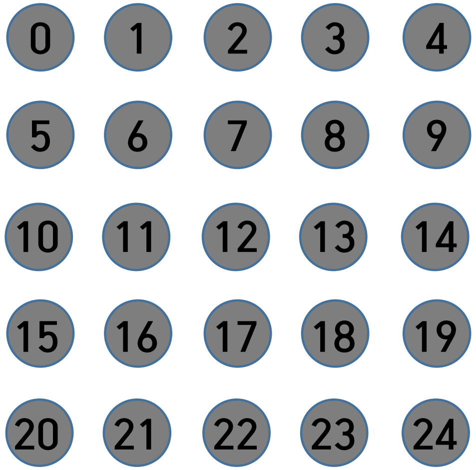
[Q] 왜 Union Find 문제 상황에 들어맞는지 생각해 보자.

Percolation 문제 해결을 위한 simulation 방법

- N을 입력으로 받고
- N × N 개의 객체를 **닫힌 상태로 초기화**

<N=5인 예>

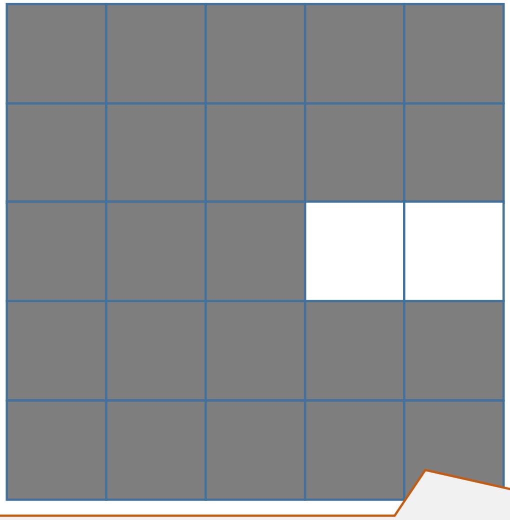




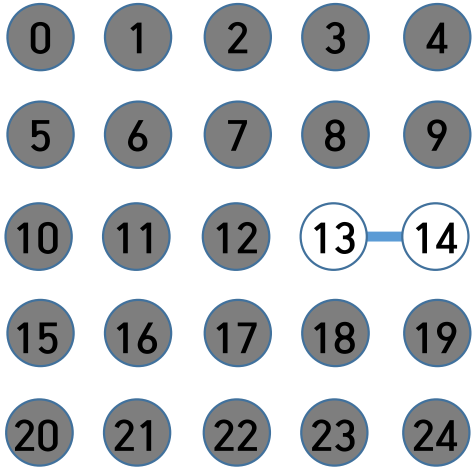
Percolation 문제 해결을 위한 simulation 방법

- N을 입력으로 받고
- N × N 개의 객체를 닫힌 상태로 초기화
- **닫힌 객체 중 하나를 (임의로 선정해) 열린 상태로 바꾸고, 위→아래로 percolate할 때까지 반복**
  - **인접한 두 객체가 모두 열렸다면 서로 연결(union)**되어 같은 connected component에 속한다고 보면 됨
  - 한 객체를 열때마다 **인접한 4곳(up, down, left, right)** 열렸는지 확인해 열린 객체와 모두 연결

<N=5인 예>



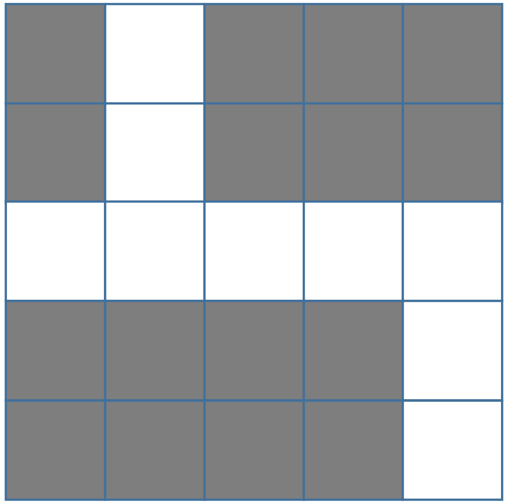
[Q] 객체 12, 6, 10, 11을 차례로 열었을 때
오른쪽 그래프의 연결 상태는 어떻게 변하나?

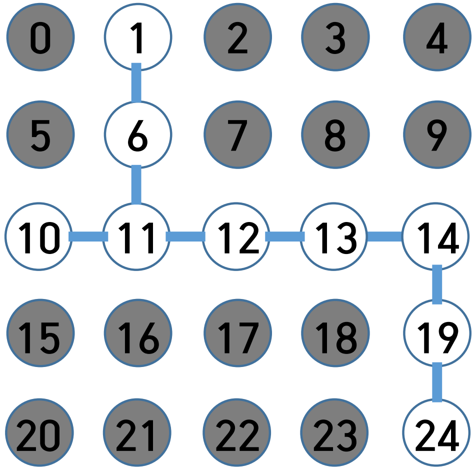


Percolation 문제 해결을 위한 simulation 방법

- N을 입력으로 받고
- N × N 개의 객체를 닫힌 상태로 초기화
- **닫힌 객체 중 하나를 (임의로 선정해) 열린 상태로 바꾸고, 위→아래로 percolate할 때까지 반복**
  - **인접한 열린 객체는 서로 연결(union)**되어 같은 connected component에 속한다고 보면 됨
  - 한 객체를 열때마다 **인접한 4곳(up, down, left, right)** 열렸는지 확인해 열린 객체와 모두 연결

<N=5인 예>



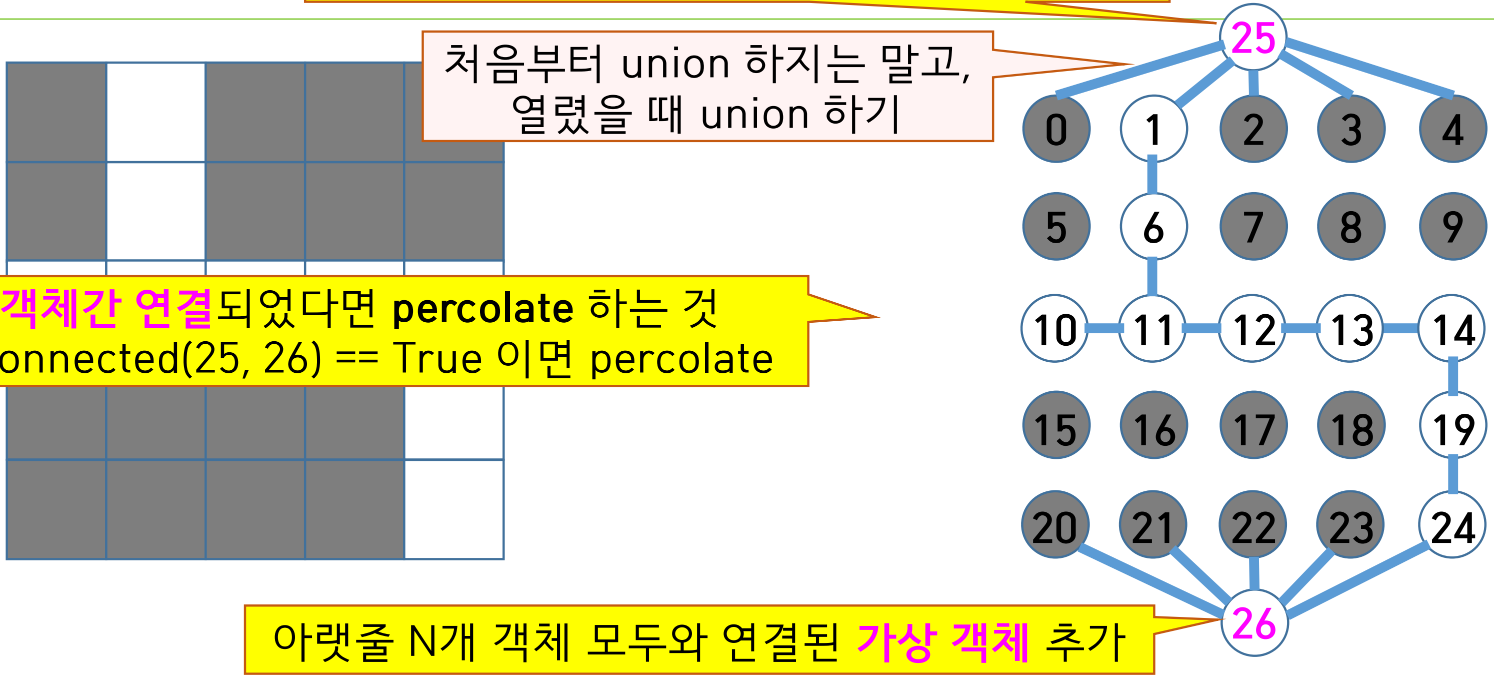


윗줄 N개 객체와 아랫줄 N개 객체의 모든 가능한 쌍 간에 connected 확인하기는 번거로움

Percolation 문제 해결을 위한 simulation 방법

- N을 입력으로 받고
- N × N 개의 객체를 닫힌 상태로 초기화
- **닫힌 객체 중 하나를 (임의로 선정해) 열린 상태로 바꾸고, 위→아래로 percolate할 때까지 반복**
  - **인접한 열린 객체는 서로 연결(union)**되어 같은 connected component에 속한다고 보면 됨
  - 한 객체를 열때마다 **인접한 4곳(u** 두 연결

<N=5인 예>



윗줄 N개 객체 모두와 연결된 **가상 객체** 추가

처음부터 union 하지는 말고,
열렸을 때 union 하기

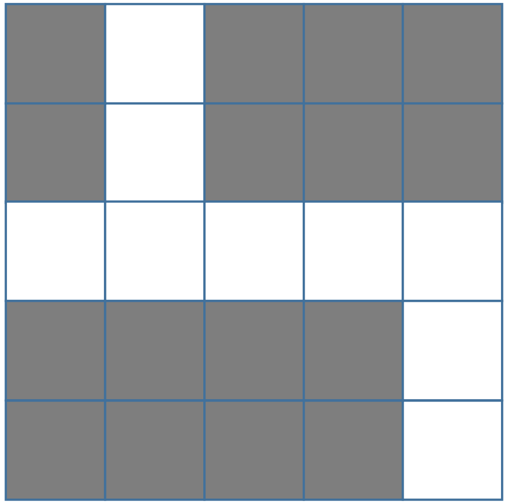
추가한 **두 가상 객체간 연결**되었다면 **percolate** 하는 것
오른쪽 예에서는 connected(25, 26) == True 이면 percolate

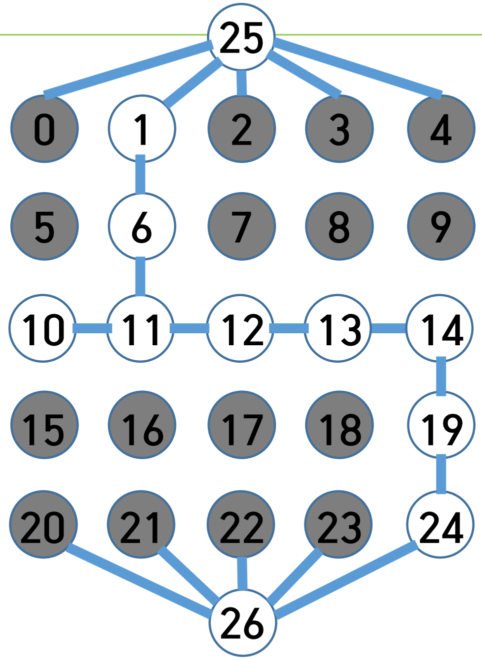
아랫줄 N개 객체 모두와 연결된 **가상 객체** 추가

Percolation 문제 해결을 위한 simulation 방법

- N을 입력으로 받고
- N × N 개의 객체를 닫힌 상태로 초기화
- 닫힌 객체 중 하나를 (임의로 선정해) 열린 상태로 바꾸고, 위→아래로 percolate할 때까지 반복
- **percolate 할 때 열려 있는 객체의 비율(=열린 객체 수 / (N × N))을 p의 예측치로 사용**
    - 열린 객체 수에 2개의 가상 객체는 포함하지 않는 것에 유의

<N=5인 예>





connected(25, 26)==True 이므로 percolate 하며, 이 때 열려 있는 객체의 비율 = 9 / 25 = 0.36

Percolation 문제 해결을 위한 simulation 방법

- N을 입력으로 받고
- N × N 개의 객체를 닫힌 상태로 초기화
- 닫힌 객체 중 하나를 (임의로 선정해) 열린 상태로 바꾸고, 위→아래로 percolate할 때까지 반복
- percolate 할 때 열려 있는 객체의 비율(=열린 객체 수 / (N × N))을 p의 예측치로 사용
- **위와 같은 시뮬레이션을 T회 반복해 p의 평균과 표준편차 구하기**

```
simulation #1: percolate 할 때 열려 있는 객체의 비율 = 9/25 = 0.36
simulation #2: percolate 할 때 열려 있는 객체의 비율 = 20/25 =0.8
…
simulation #10,000: percolate 할 때 열려 있는 객체의 비율 = 14/25 = 0.56

따라서 열려 있는 객체 비율의
mean                     = 0.5929934999
stdev                     = 0.0087699042
```

프로그램 구현 조건

- 격자 크기와 시뮬레이션 횟수가 주어졌을 때, percolate하는 객체 비율의 평균과 표준편차 찾는 함수 구현

  def simulate(**n**, **t**):

- 입력 **n**, **t** : 1≤n≤200, 2≤t≤10^5) 범위의 정수로
- **n** × **n** 격자에 대해 **t**회 시뮬레이션 반복하여야 함을 의미

- 반환 값: 입력과 같이 시뮬레이션 하여 percolate 할 때 열린 객체 비율의 평균과 표준편차를 반환
  - t회 예측치가 x1, x2, ⋯, xt라 할 때
  - 평균 = (x1+x2+⋯+xt) / t   # statistics.mean() 사용해 계산
  - 표준편차 = √[{(x1-평균)^2 + (x2-평균)^2 + ⋯ (xT-평균)^2} / (t-1)] # statistics.stdev() 사용해 계산
  - 위 값은 소수점 아래 절삭 등은 하지 않고 계산한 결과 그대로 반환
  - “return 평균, 표준편차” 하면 두 값을 함께 2-tuple로 반환할 수 있음

- 이번 시간에 제공한 코드 Percolate.py에 위 함수 추가해 제출

```
>>> print(simulate(200, 100))
(0.592296, 0.008537780478979858)
```

simulate() 함수가 mean, stdev를 반환하므로
이를 print() 하면 왼쪽과 같이 출력됨

프로그램 구현 조건

- 최종 결과물로 Percolate.py 파일 하나만 제출하며, 이 파일만으로 코드가 동작해야 함
- import는 원래 Percolate.py 파일에서 하던 패키지 외에는 추가로 할 수 없음 (statistics, math, random, timeit)
- 평균과 표준편차를 구할 때는 반드시 statistics.mean()와 statistics.stdev() 함수를 사용해야 함
- Percolate.py 내에 이미 구현되어 있는 코드는 채점에 사용되는 코드이니 제거하거나 수정하지 말 것
- import, simulateQU(), simulateQF(), __main__ 아래 코드

simulate() 함수의 입출력 예

```
>>> print(simulate(200, 100))
(0.592296, 0.008537780478979858)
```

simulate() 함수가 mean, stdev를 반환하므로
이를 print() 하면 왼쪽과 같이 출력됨

```
>>> print(simulate(2, 10000))
(0.668475, 0.11720195392910683)
```

```
>>> print(simulate(2, 100000))
(0.6666575, 0.11785495997906034)
```

그 외 유의사항

- 수업 자료와 함께 제공된 코드 필요하다면 내용 복사해서 작성한 코드 일부로 사용 가능
- 예: WQU, QU, QF

- __main__ 아래에 simulate 함수의 채점에 사용되는 테스트 케이스가 있으며
- 정확도와 성능을 검증합니다.
- 각 케이스에 대해 P/F 혹은 Pass/Fail이 출력되니 검증에 활용하세요.
- **성능 테스트**는 간혹 fail할 수는 있지만 (갑작스러운 통신량 증가, background 어플리케이션 실행 등 여러 이유로 순간적으로 느려질 수 있음) **거의 항상 pass여야 통과**로 봅니다.
- 성능 테스트를 모두 통과하려면 이번 시간에 배운 가장 빠른 방법을 사용해 구현하세요.

그 외 유의사항

- n*n개 격자 중 닫힌 격자 하나를 임의로 선정해 열기 위해서는
- random.shuffle() 함수를 활용할 수 있습니다. 아래 예제를 참조하세요.

```
indicesToOpen = [i for i in range(n*n)]     # n*n개의 index를 담은 리스트 생성
random.shuffle(indicesToOpen)               # 리스트에 담긴 index들을 무작위로 섞음
# 이제 indicesToOpen에 저장된 index를 하나씩 차례로 읽으며 open하면
# 아직 open하지 않은 격자 중 하나를 임의로 선정해 열게 됨
```

실습 과제 (프로그래밍) 유의사항

- **개별 평가**이므로 코드는 꼭 각자 **직접 작성**해 주세요.
- 매주 제출한 코드에 대해 2개 분반 함께 유사도 검사를 합니다.

실습 문제 풀이 & 질의응답

- 작성한 코드는 lms > 강의실 > 오늘 수업 > 실습 과제 제출함에 제출
- **종료 시간 이전 풀이를 완료한 경우 튜터의 검사 받고 출석 확인 후 퇴실** 가능

- 시간 내 제출 못한 경우 **실습 종료 시간에 출석 확인**하고 퇴실
- 과제는 **내일 11:59pm까지 제출**
- **제출하면 기본 점수** 있으므로 그때까지 작성한 코드를 **꼭 제출**하세요.
- **마감 시간 후에는 제출 불가**합니다.

- 실습 과제 **채점 관련 질문은 TA에게** 해주세요: **튜터 소개**

- **다음 시간도 수업 전날까지 예습 & 퀴즈 완료**해 주세요.

공지사항: lms에 과제물을 올릴 때 기존에 올린 파일과 같은 이름의 파일 올리면 -1, -2 등이 붙는데, 이때문에 감점되지는 않습니다. 또한 수업 자료도 변경되면 마찬가지로 숫자가 붙어 있으니 변경 여부 확인해 보세요 (첫 버전에서 약간씩 수정이 있습니다. 특히 실습 과제 요건 변경 있는지 꼭 확인).