Union Find (Disjoint Set Forest)

Union Find 문제 정의, 해결 방법, 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
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

(실습 시간을 절약하기 위해)
선호하는 개발 환경이 있다면 미리 설치해 두세요.
lms의 가상 실습실을 사용한다면 미리 시작해 두세요.

Union Find (Disjoint Set Forest)

Union Find 문제 정의, 해결 방법, 활용도 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 답안 공개)
02. Union Find 문제 정의 및 활용
03. 첫 번째 방법: Quick-Find
개선
04. 두 번째 방법: Quick-Union
개선
05. Quick-Union의 개선
06. 실습: Percolation 문제에 Union-Find의 해(WQU) 적용

(실습 시간을 절약하기 위해)
선호하는 개발 환경이 있다면 미리 설치해 두세요.
lms의 가상 실습실을 사용한다면 미리 시작해 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(화/수)

퀴즈 풀이
개요 복습

수업(화/수)

응용/심화 내용 학습
질의 응답

수업(화/수)

실습 과제 (프로그래밍)
질의 응답

실습과제 수업 중 완료 못한 경우
당일 저녁 or 다음날까지
각자 수행해 제출

~3시간

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

6 그래프와 관련

문제 정의: Union Find (연결 상태 변경 & 확인)

6 정점개수

- N개 객체 주어짐
  - 0 ~ (N-1) 까지 정점(vertex)으로 표현
  - 간선(edge) 없는 상태에서 시작

간선

- 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)

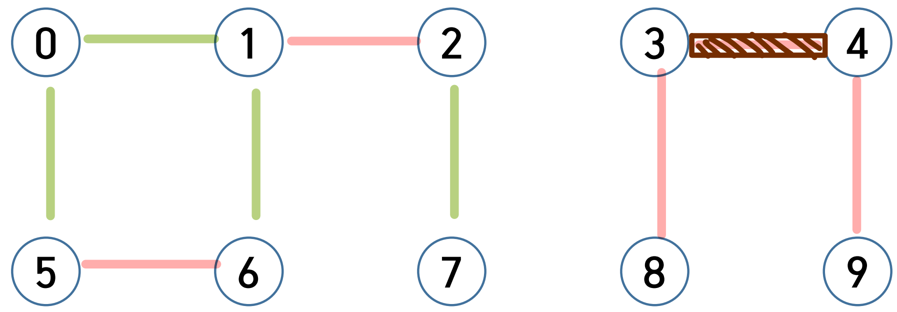
=find 명령

- 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

6 0~9(10개)

- 예제(N=10)



union(4,3)
union(3,8)
union(6,5)
union(9,4)
union(2,1)
connected(0,7) = False
connected(8,9) = True

(정점가는 길 모르지만 간선 여러개 거쳐서 갈 수 O)

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

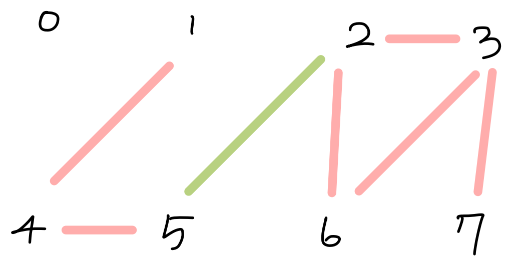
연결 상태가 **동적**으로 계속 변함.
따라서 이전에 했던 답 재활용하기 보다 그때그때 다시 답 확인 필요

문제 정의: Union Find

- 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 절선으로 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

- 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

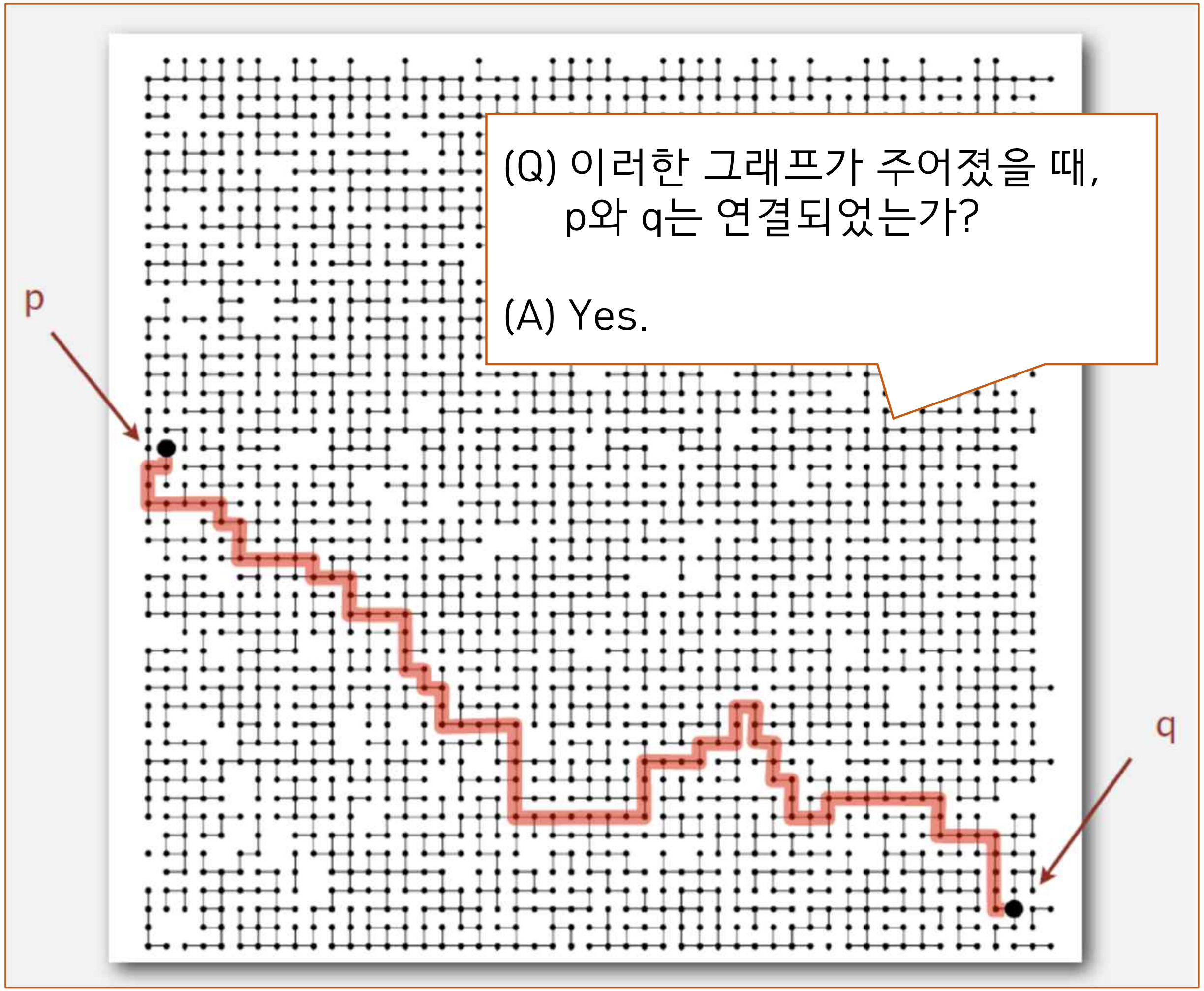
union(5,2)
connected(1,7) = T
connected(0,6) = F

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

Union Find 문제의 해는 어디에 활용되는가?

- 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

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

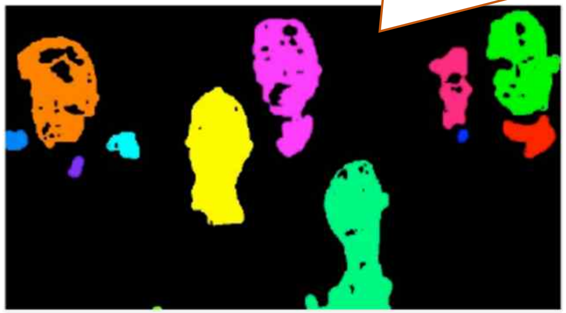
(Q) 컴퓨터망에서 두 장비가 연결되었는지 확인

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

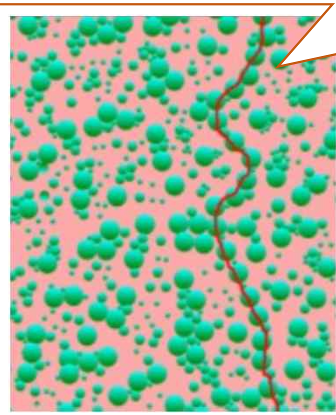
(Q) Kruskal's minimum spanning tree algorithm의 일부로 활용

>>> 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)



Union Find (Disjoint Set Forest)

Union Find 문제 정의, 활용도, 해결 방법 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)

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

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

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

- 어떤 자료구조에 저장해야 할까?

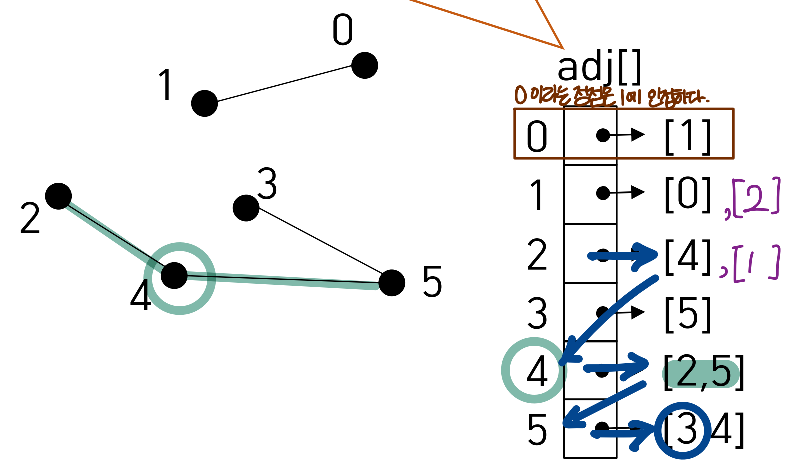
[Q] 일반적으로 그래프 저장에 많이 사용되는

개별 간선 정보 저장하는 오른쪽 방식 생각해 보자.

(N x N 배열보다 compact)

union, connected는 빠른가?

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



· union(1,2) : 자료구조는 "두 군데" 바꿔야 함.
두군데 바꾸는데 상수시간(~1) 걸림

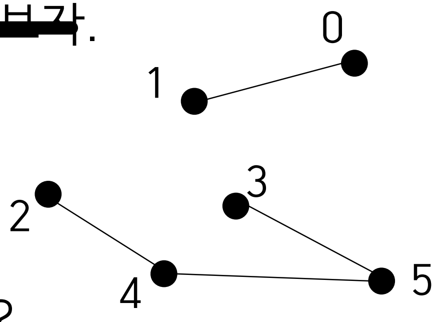
· connected(2,3) : ~E >> n
(간선개수) (정점개수)

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

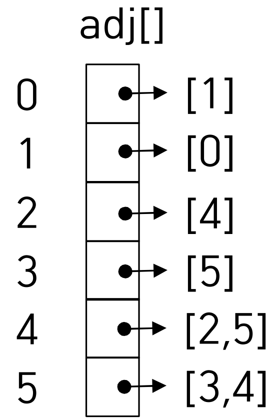
- ~~그래프 **연결 상태** 저장해야 함~~

~~[Q] 개별 간선 정보 저장하는 오른쪽의 기본 방식 생각해 보자.~~

~~union, connected는 빠른가?~~



왼쪽 그래프의 개별 간선 정보를 저장한 예



[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", …)

Quick Find

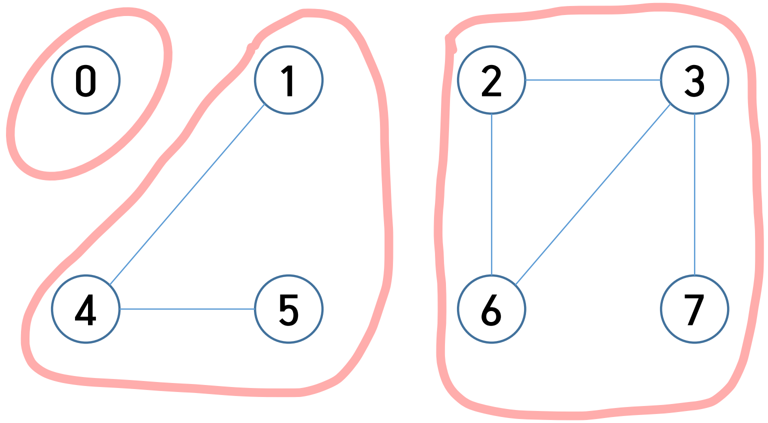
첫 번째 해결책: 각 객체가 어느 연결된 덩어리(connected component)에 속하는지 계속 기록하고 업데이트

- 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? {0}, {1, 4, 5}, {2, 3, 6, 7}

{4, 5, 1}
[Q] ~~{4, 5}~~는 connected component인가? No

{2, 3, 6, 7}
[Q] ~~{2, 3, 6}~~은 connected component인가? No

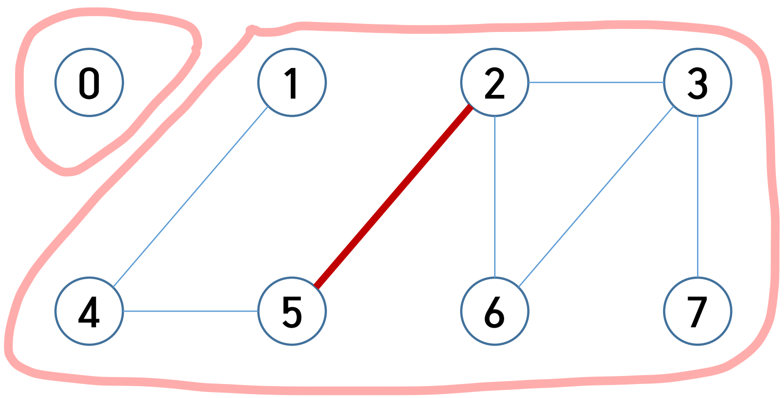
첫 번째 해결책: 각 객체가 어느 연결된 덩어리(connected component)에 속하는지 계속 기록하고 업데이트

- 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}, {1,4,5,2,6,3,7}

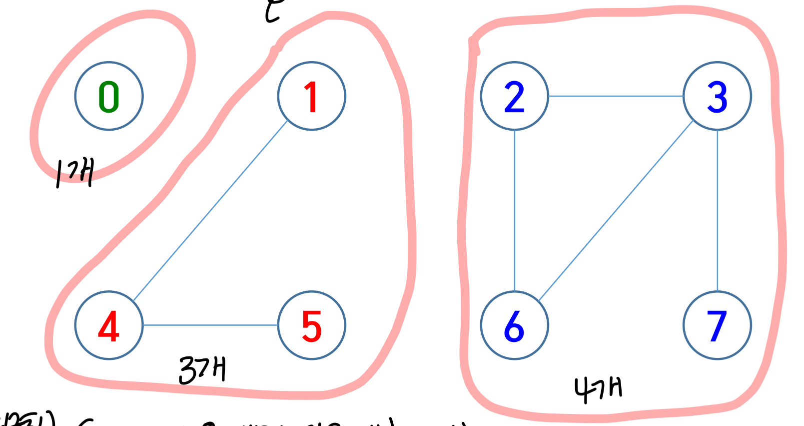
[Q] {1, 4, 5}는 connected component인가? NO

[Q] {2, 3, 6, 7}은 connected component인가? NO

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

e 0 번네가 속한 ID를 인덱스에 저장



우리규칙) Component 요소개수가 적은 애부터 0부여.

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

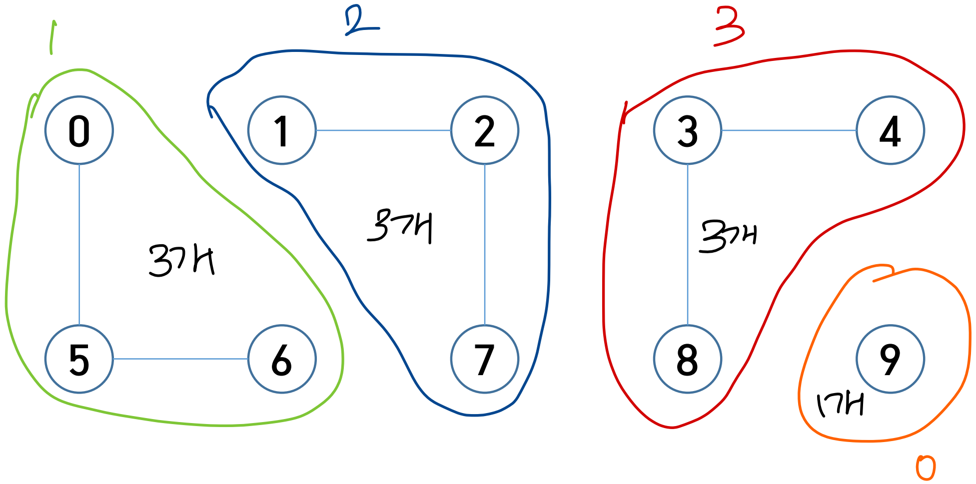
같은 ID를 갖고있으면

연결되어있음을 알수 O

→ Connected를 상수시간으로 변경 : Quick Find

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

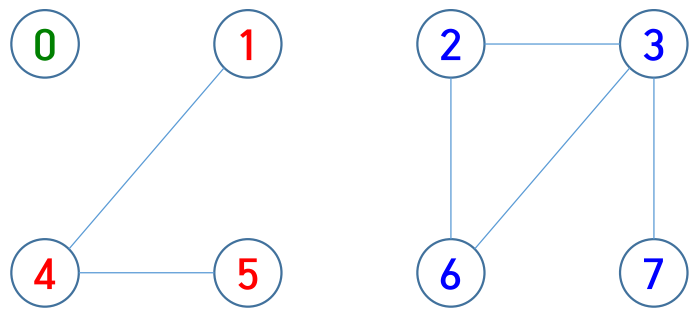


index:	0	1	2	3	4	5	6	7	8	9
ids[]	1	2	2	3	3	1	1	2	3	0

[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

(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

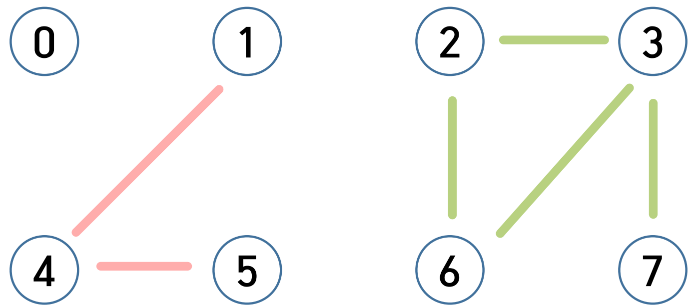
첫 번째 해결책(Quick-Find): 각 객체가 어느 연결된 덩어리(connected component)에 속하는지 크기 N의 배열에 기록 & 업데이트

- $\boxed{\text{ids[i] = i}}$로 초기화

  는 시작할 때 각자가 컴포넌트니까 각자 번호를 컴포넌트 ID로 부여함.
- 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

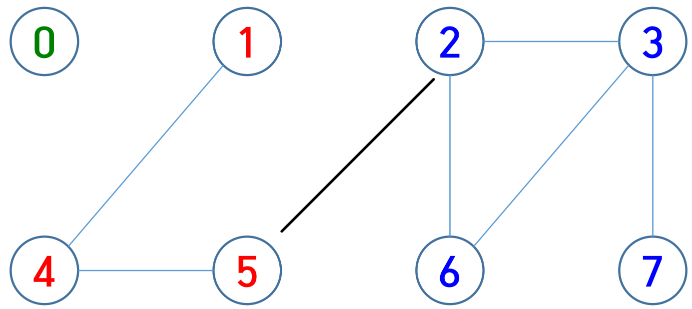
[Q] a, b의 id 중 더 작은 값 사용한다고
가정하고 ids[]의 변화 과정을 써보자.



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)			"			1	"	
union(2,3)			"	2		"		
union(6,2)			"				2	"
union(3,6)	(안바꿔도 됨)							
union(3,7)								2
connected(1,7)								

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)			1	1			1	1
connected(1,7)	TRUE							
connected(0,6)	False							
union(0,3)	0	0	0	0	0	0	0	0

→ 전체를 한 번 돌아야 함

[Q] 이 방법의 단점은 무엇이라고 생각하는가? (유의: 값을 변경할 부분만 아니라 변경하지 않는 부분도 빠짐없이 확인 필요) connect 시간은 줄였지만. union 시간이 늘어남.

	Union	Connected
QF	~N	~1

```
N = 8

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

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
```

0	1	2	3	4	5	6	7

두 값 a, b 중
(더 작은 값, 더 큰 값)
반환

작은게 1번ID   큰게 2번ID

```
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))
```

```
def union(a, b):
    id1, id2 = minmax(ids[a], ids[b])
    for i, _ in enumerate(ids):
        if ids[a] == ids[b] · ids[i] = id1
```

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
```

튜플/리스트로 만들어짐

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

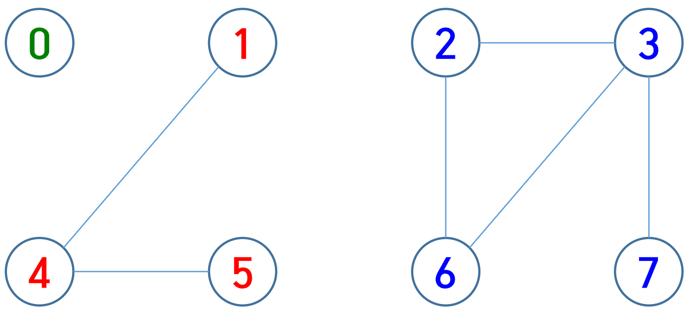
Union Find (Disjoint Set Forest)

Union Find 문제 정의, 활용도, 해결 방법 이해

01. 퀴즈 풀이 & 에습 내용 복습 (이번 주 #1~3차 문제 공개)
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

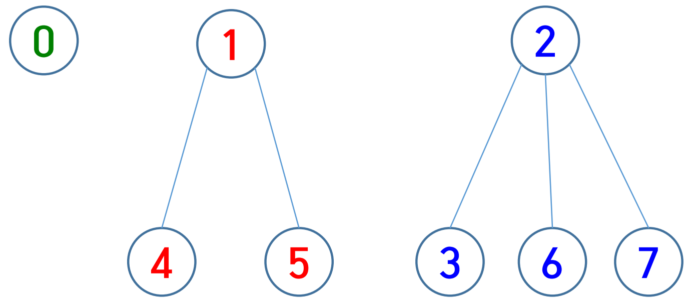
QF(Quick-Find)에서 사용하던 구조의 다른 해석

- 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

- 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)**



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

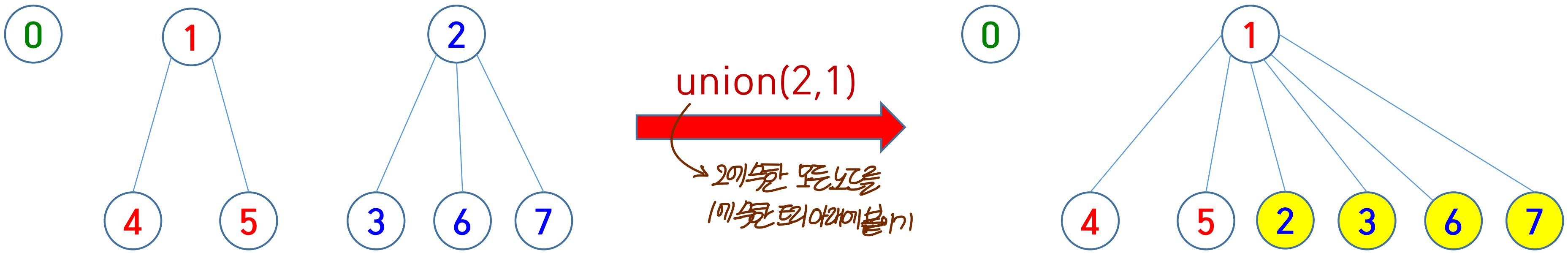
parent

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



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

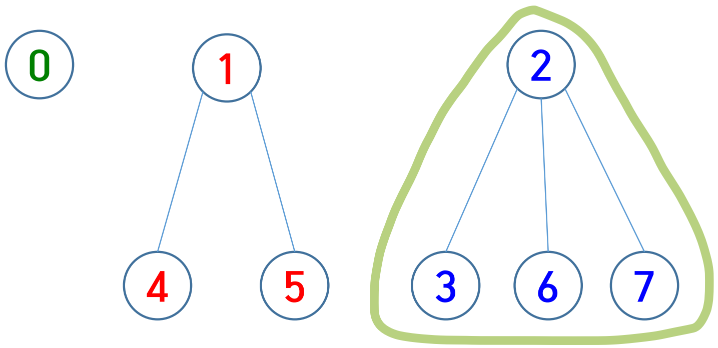
→ Union이 하는 일이 多

→ 루트만 루트에 붙일까?

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

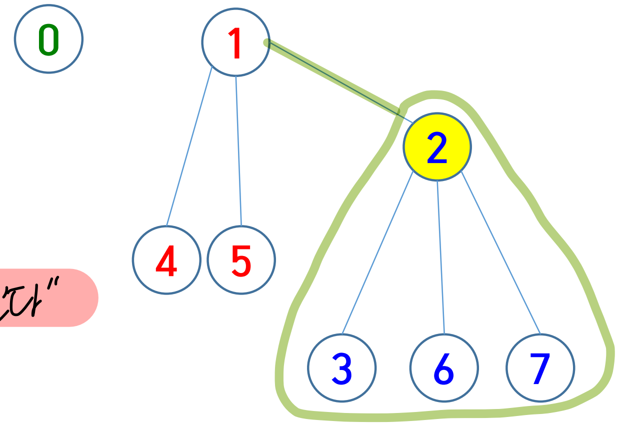


index:	0	1	2	3	4	5	6	7
ids[]	0	1	2	2	1	1	2	2

루트들만비교

union(2,1)

"루트를 루트에 붙인다"



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] …]]]**

[Q] connected(1,2)에 답하는 과정을 보이시오.

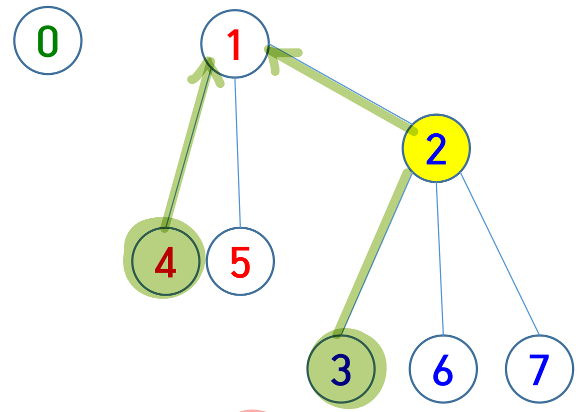
[Q] connected(3,4)에 답하는 과정을 보이시오.

: 3과 4의 루트를 찾아서 비교

(3의루트는 2 , 4의루트는 1 이고,

걔네는 둘다 1이므로 T)

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



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

∴연결O

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)

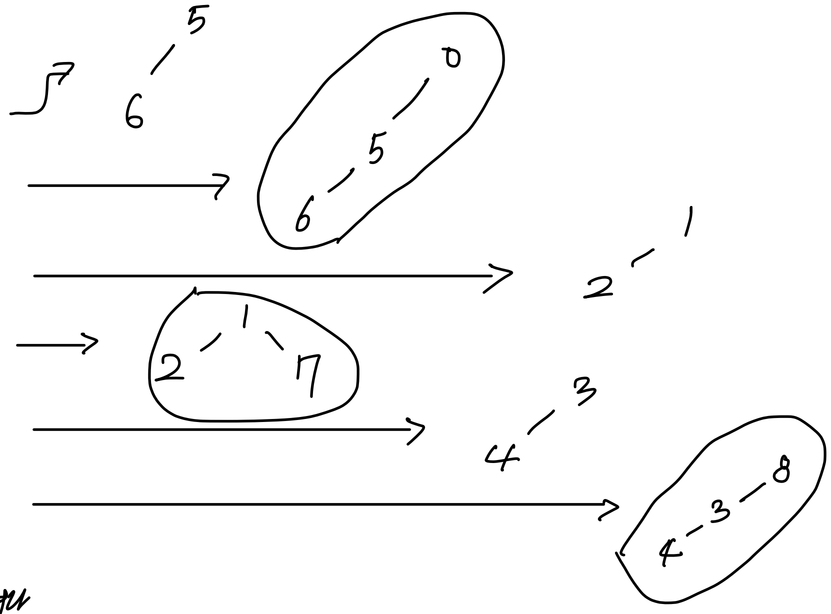
union(2,1)

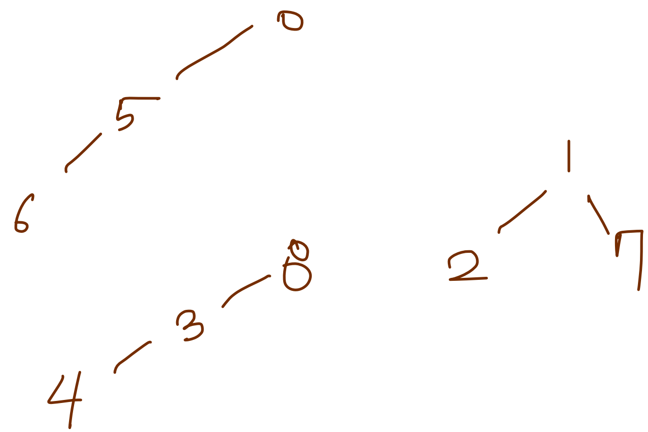
union(7,1)

union(4,3)

union(4,8)

4의루트(3)를
8의루트에 붙여서





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 옮겨 붙이기

$\overbrace{\text{6의 루트}}^{0}$를 $\overbrace{\text{7의 루트}}^{1}$에 붙여라

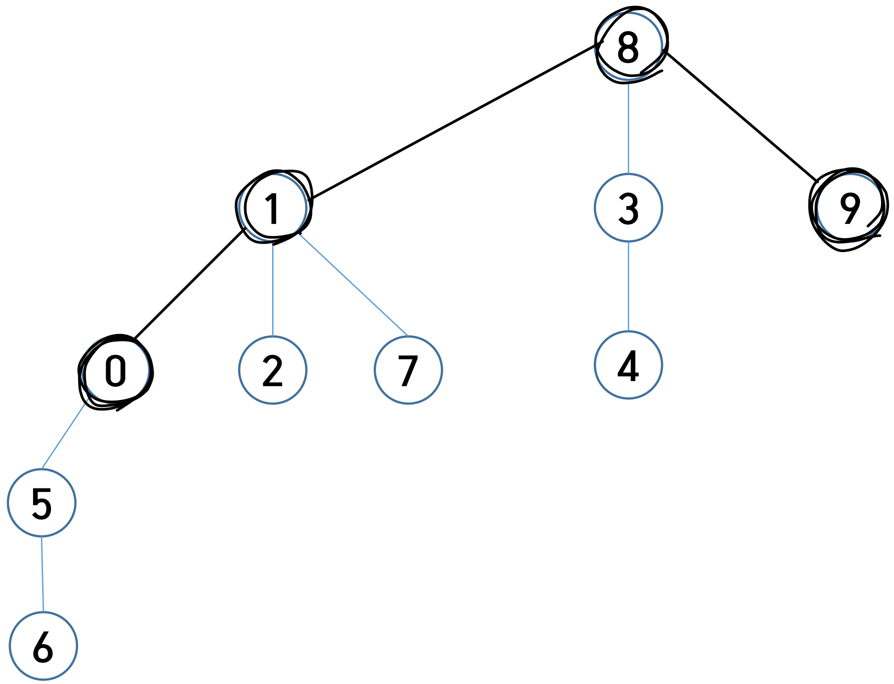
union(6,7)

union(9,8)

union(7,3)

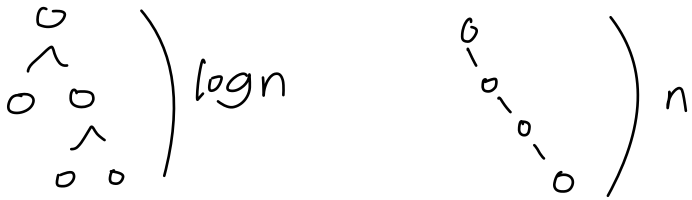
connected(5,4) : T

connected(7,9) : T



[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




```
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
```

루트도달전까지

p의루트 q의루트

p의루트 q의루트

p루트 q루트

0	1	~~2~~ 4	3	4	5	6	7	8	9

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?

```
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))
```

	Union	Connected
QU	~d (깊이)	~d

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
```

Union Find (Disjoint Set Forest)

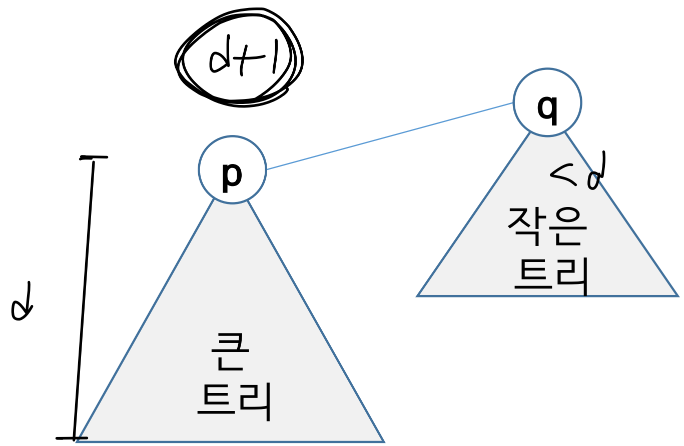
Union Find 문제 정의, 활용도, 해결 방법 이해

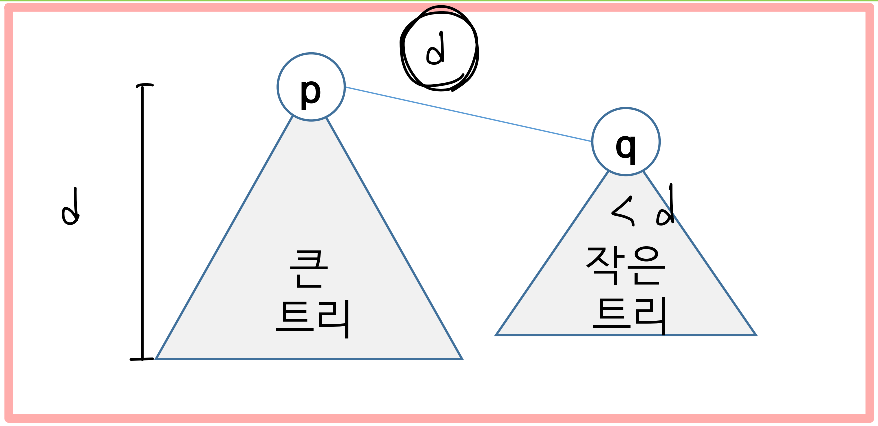
01. 퀴즈 풀이 & 에습 내용 복습 (이번 주 #1~3차 문제 공개)
02. Union Find 문제 정의 및 활용
03. 첫 번째 방법: Quick-Find
04. 두 번째 방법: Quick-Union
05. Quick-Union의 개선
06. 실습: Percolation 문제에 Union-Find의 해(WQU) 적용

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





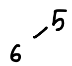
[Q] Union 후 Tree의 최대 depth가 몇 증가하는가? d+1

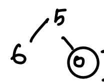
[Q] Union 후 Tree의 최대 depth가 몇 증가하는가? d

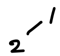
Weighted QU(Quick-Union) 수행 예

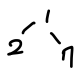
- connected component를 tree로 봄
- ids[i]: 객체 i의 parent
- 만약 객체 i가 root라면 ids[i] = i
- connected(p,q) == True if root(p) == root(q)
- root(i) = ids[ids[ids[… ids[i] …]]]
- **union(p,q): 작은 트리의 root를 큰 트리의 root 아래 연결**
- **트리의 크기는 객체 수**

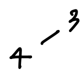
N=10

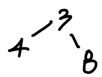
union(6,5) — 

union(5,0) —  개수가 작은 애를 개수가 많은 애 자식으로

 — union(2,1)

 — union(7,1)

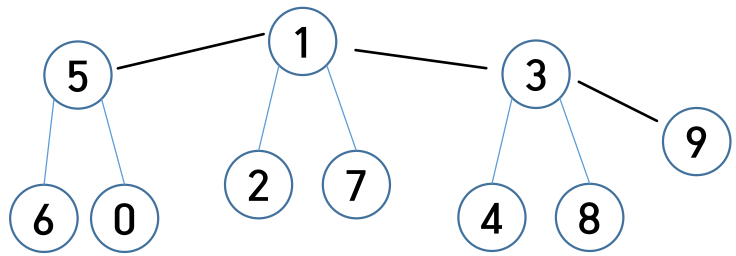
union(4,3) — 

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**

루트끼리 붙여



depth가 2까지 줄어듦
(루트까지의 간선개수)

union(6,7)

union(9,8)

union(7,3)

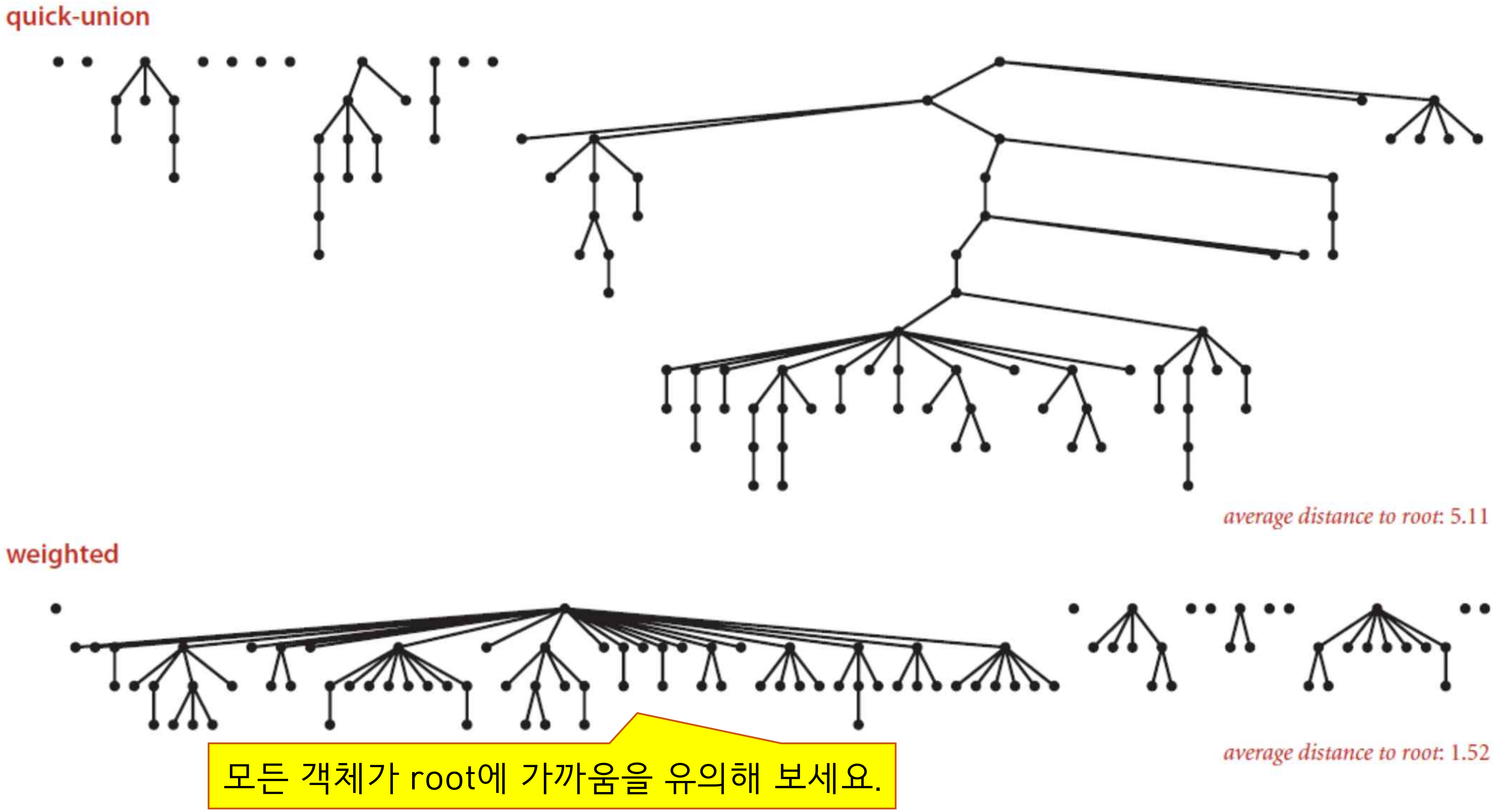
connected(5,4) = T

connected(7,9) = T

[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[]	5	1	1	1	3	1	5	1	3	3



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

QU(Quick-Union)

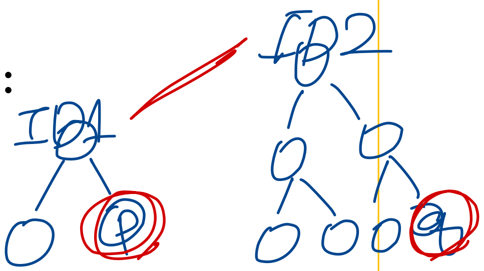
```python
N = 10

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
```



ids[id1]값을 id2로 바꿔주기

Weighted QU

```python
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 객체를 root로 하는 tree의 크기 저장하는 배열

p가 속한 트리의 사이즈가 작은 경우

q가 속한 트리의 사이즈가 작은 경우

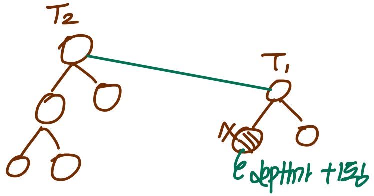
크기가 작은 애를 바꿔주기

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

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

- 증명:
- N개 객체 중 임의의 정점을 x라 하자.

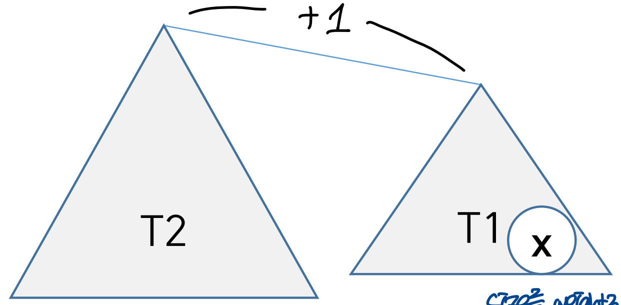
vertex



- 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)

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

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



size를 weight로 생각

root 거리	x가 속한 tree size(노드수)
0	1
+1	×2 (최소두배)
+k	2^k

$k \le \log_2 N$

	Union	connected
WQU	~logN	~logN

비교해야할 애들의 갯수 등

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

```
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]
```

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

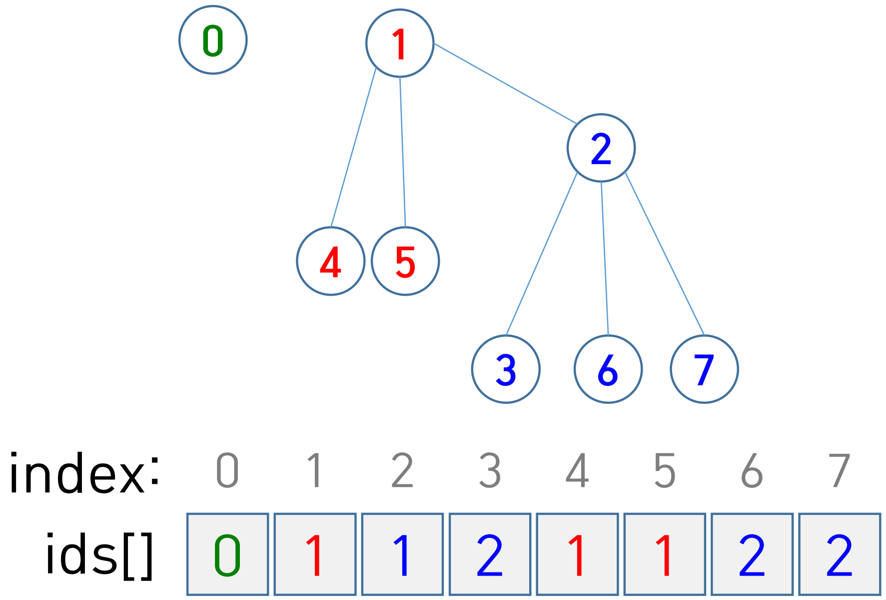
[Q] WQU와 QF를 비교하면 어느 쪽이 더 빠른가?

예: 10^9개 객체에 대해 10^9번의 union 수행?

→ 10^9와 $\log_2(10^9)$을 비교
→ 압도적으로 빨라짐

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



정리: 알고리즘 설계 과정

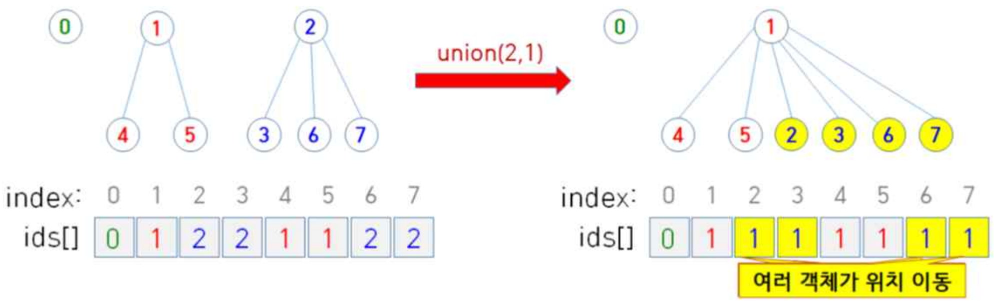
- 첫 알고리즘 고안
- 성능 예측 and 부족한 부분 있으면 이유 파악
- 문제점 해결 위한 방법 고안
- 위 단계 반복

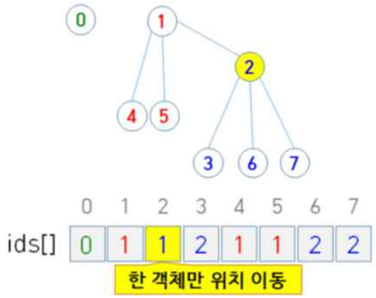
Algorithm	Quick-Find	Quick-Union	Weighted QU
ids[] 초기화	~N	~N	**~N**
union	~N	~N	~$\log_2$(N)
find(connected)	1 (상수시간)	~N	~$\log_2$(N)

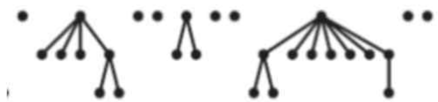
Tree가 flat하므로 find는 빠름. 하지만 flat하게 유지하기 위해 **union 시 여러 객체를 옮겨야 하므로** 시간이 오래 걸림

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!







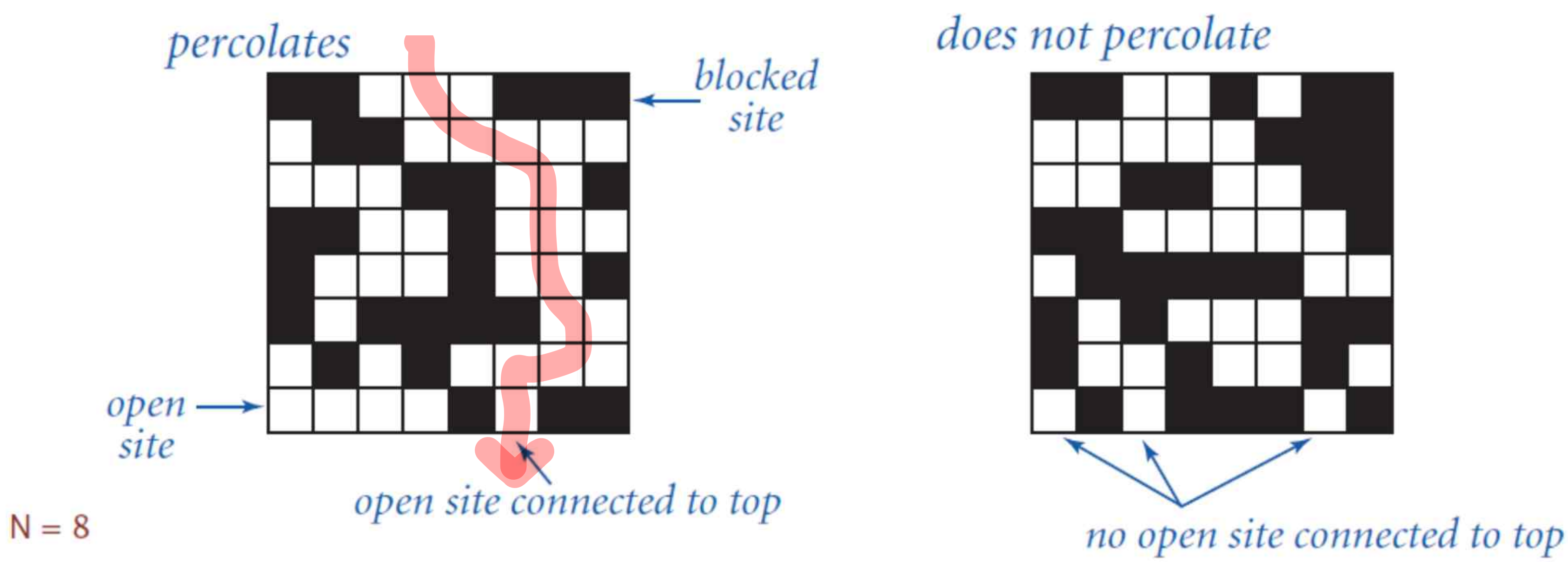
Union Find (Disjoint Set Forest)

Union Find 문제 정의, 활용도, 해결 방법 이해

01. 퀴즈 풀이 & 예습 내용 복습 (이번 주 #1~3차 문제 공개)
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

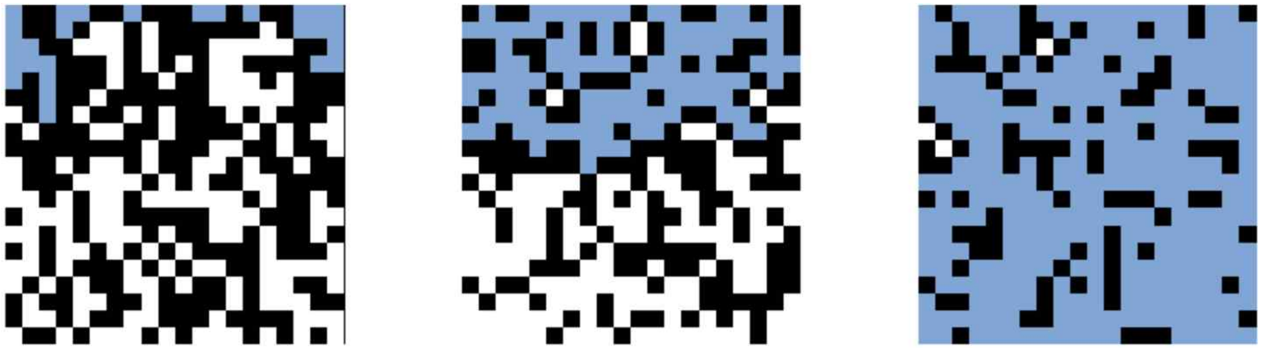
N × N 격자가 "Percolate": 윗줄 → 아랫줄 가는 경로 존재

- 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 (**열린 격자 통해 이동 가능**) 이 격자는 **percolate** 한다고 함



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

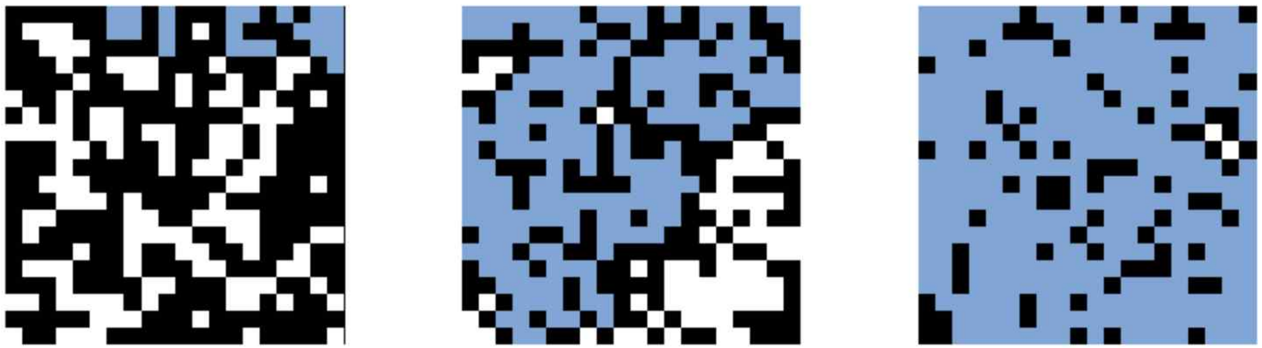
- **p: 열린 격자의 비율(퍼센티지)**
- 다음 질문에 답하고자 함: p가 클수록 percolate할 가능성 높아질 텐데, 평균적으로 어떤 p값 이상일 때 거의 항상 percolate하는가?

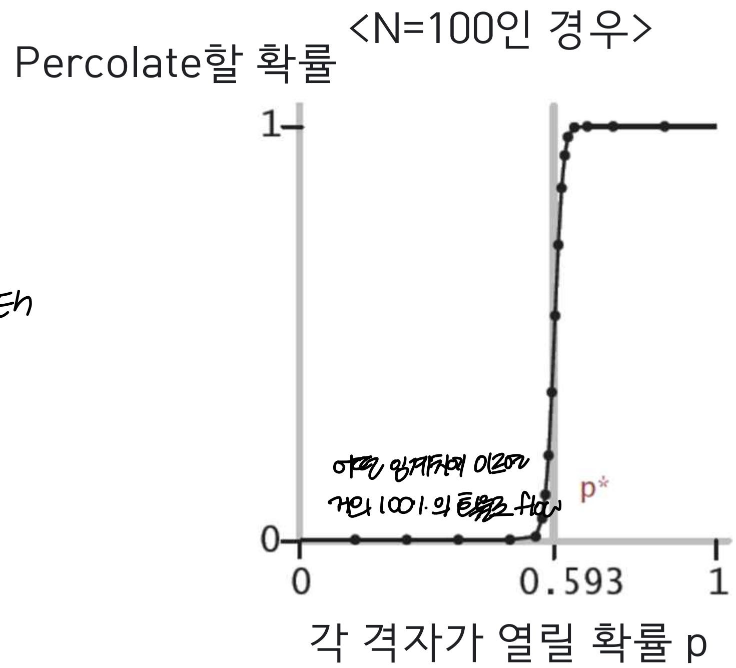


p low (0.4) 40% 연상태
does not percolate

p medium (0.6) 60% 연상태
percolates?

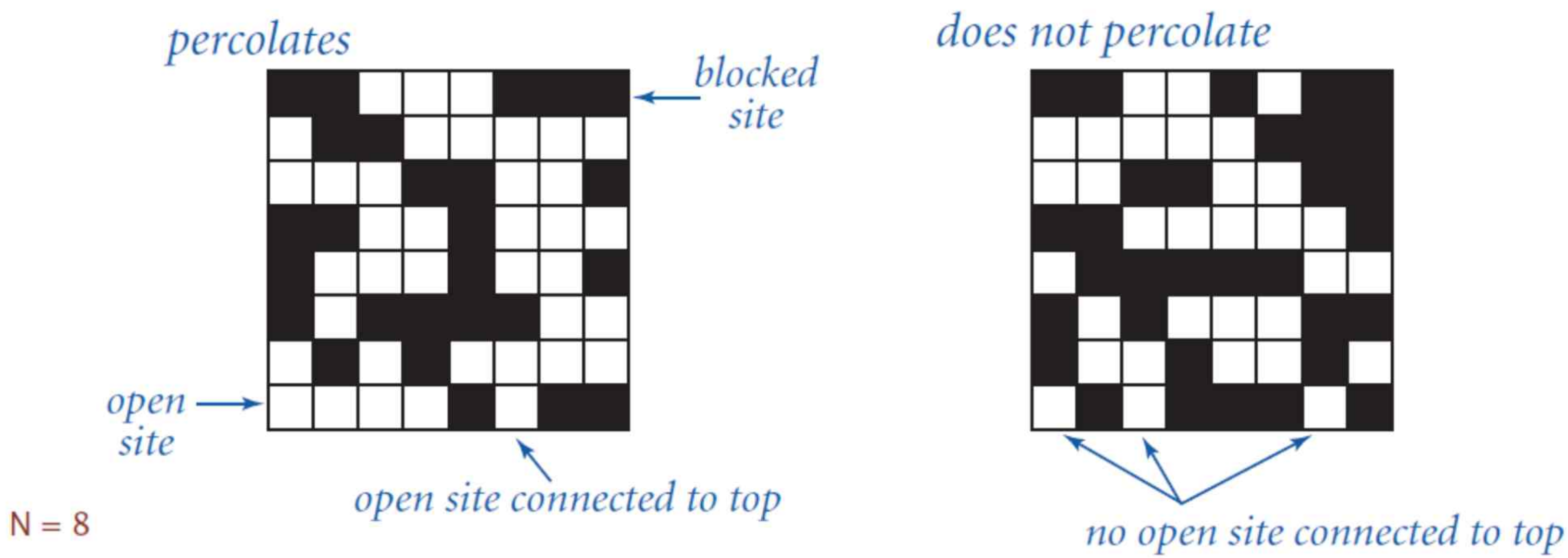
p high (0.8) 80% 연 상태
percolates





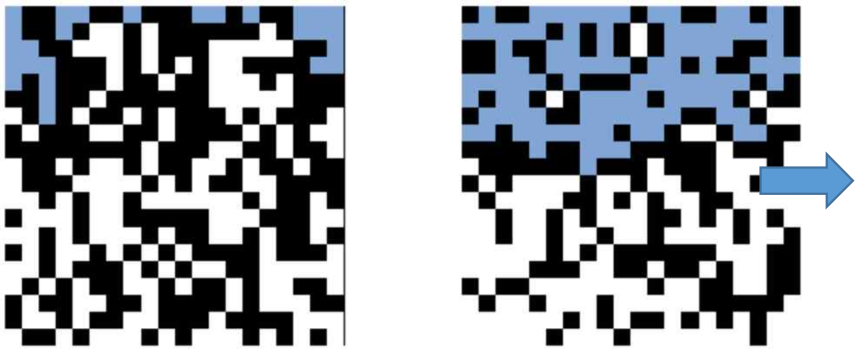
Percolation 문제 활용 예

- (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

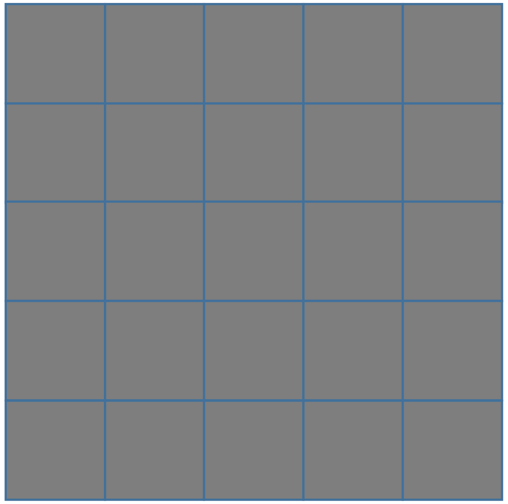


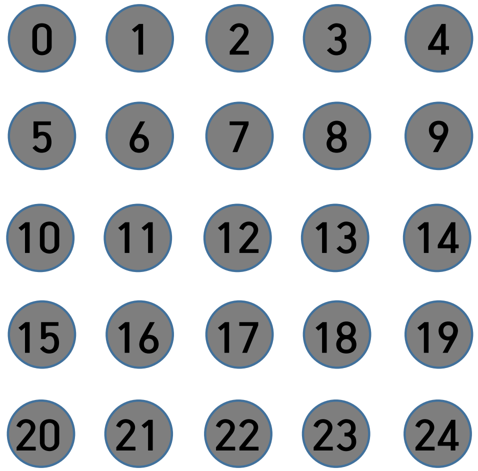
하나씩 임의로 선정해 열다가 percolate하는
순간 열린 격자의 비율 구하기
이를 반복한 후 수집한 값의 평균 내기

Percolation 문제 해결을 위한 simulation 방법

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

<N=5인 예>

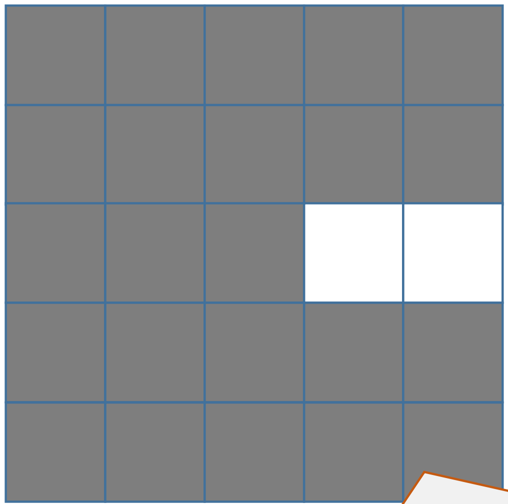




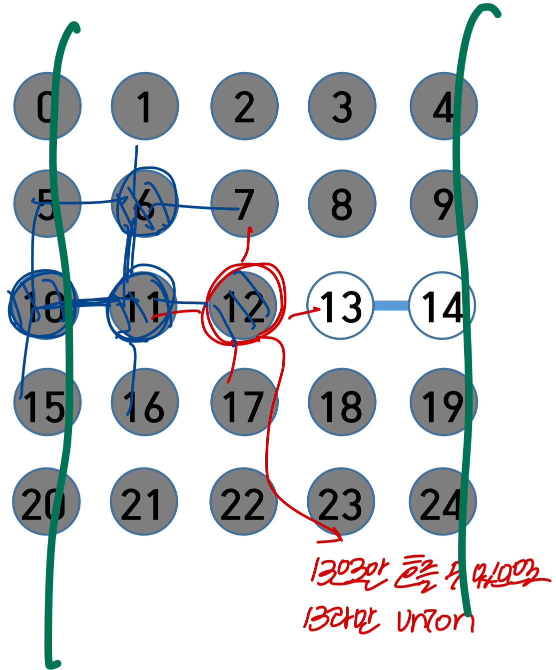
Percolation 문제 해결을 위한 simulation 방법

- 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 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>



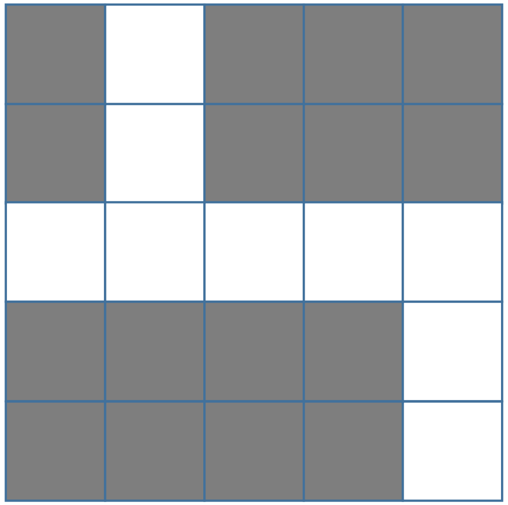
[Q] 객체 12, 6, 10, 11을 차례로 열었을 때
오른쪽 그래프의 연결 상태는 어떻게 변하나?

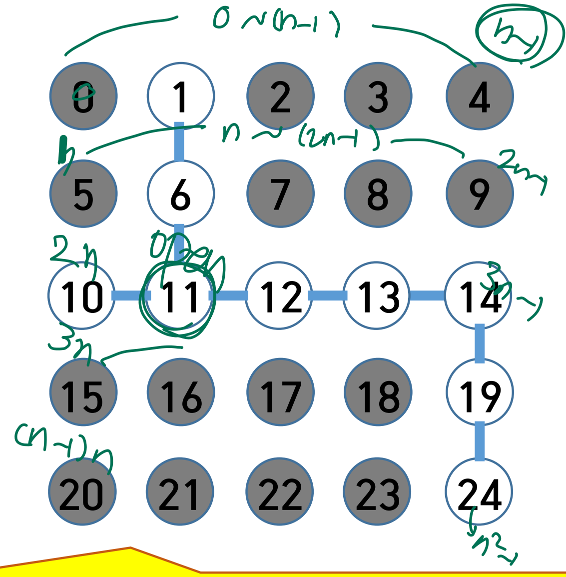


Percolation 문제 해결을 위한 simulation 방법

- 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>





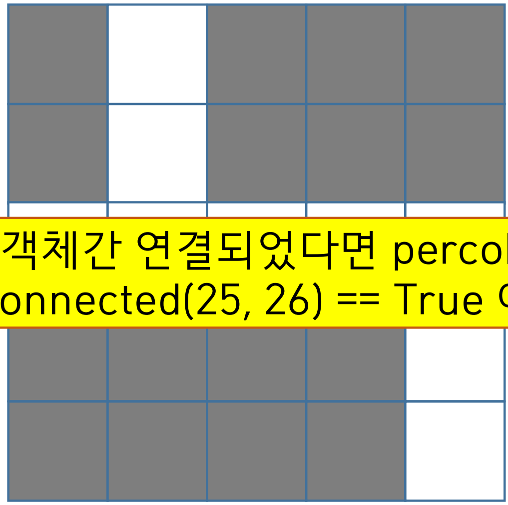
윗줄 N개 객체와 아랫줄 N개 객체의 모든 가능한 쌍 간에 connected 확인하기는 번거로움

Percolation 문제 해결을 위한 simulation 방법

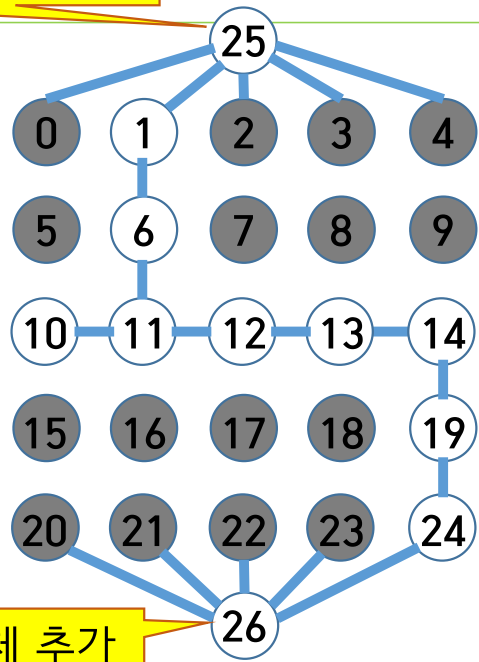
- N을 입력으로 받고
- N × N 개의 객체를 닫힌 상태로 초기화
- **닫힌 객체 중 하나를 (임의로 선정해) 열린 상태로 바꾸고, 위→아래로 percolate할 때까지 반복**
  - **인접한 열린 객체는 서로 연결(union)**되어 같은 connected component에 속한다고 보면 됨
  - 한 객체를 열때마다 **인접한 4곳(u** … 두 연결

윗줄 N개 객체 모두와 연결된 가상 객체 추가

<N=5인 예>



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

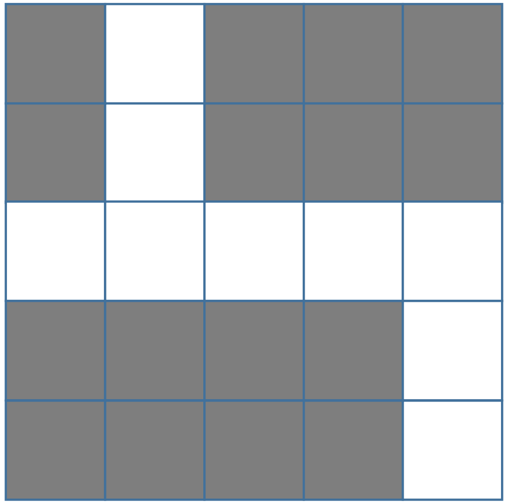


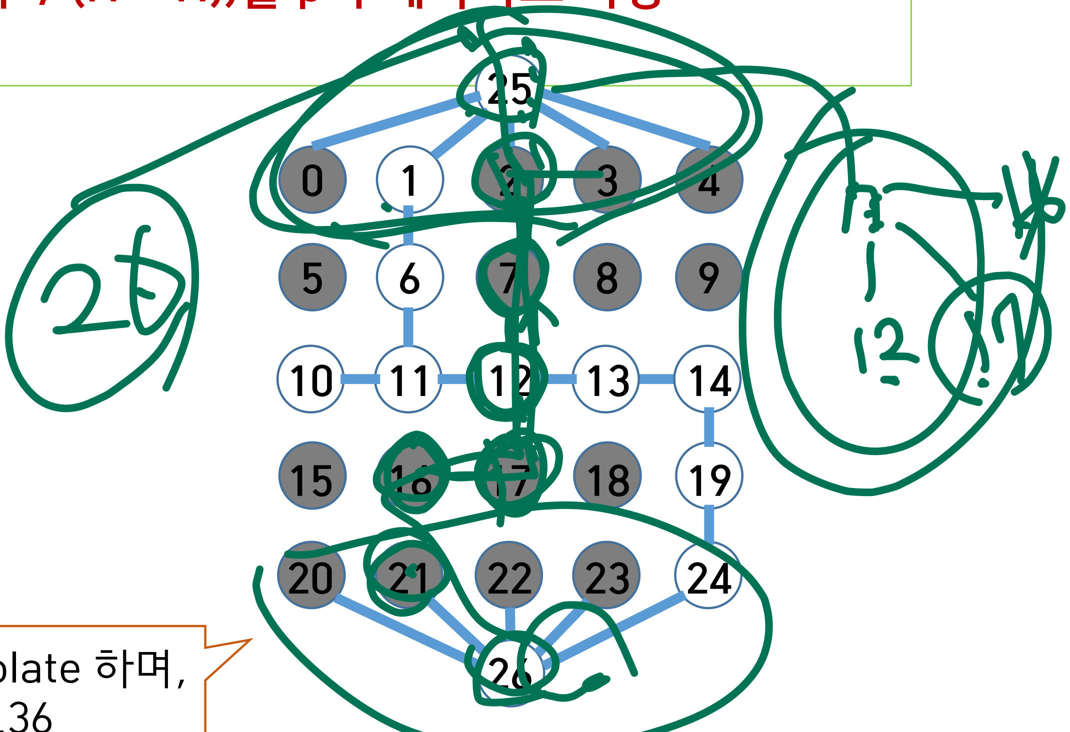
아랫줄 N개 객체 모두와 연결된 가상 객체 추가

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 평균 혹은 신뢰 구간 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
95% confidence interval = [0.5912745988, 0.5947124012]

프로그램 입출력 조건

- 정수 n과 t를 입력으로 받는 함수 정의 ($1 \le n \le 200$, $2 \le t \le 10^5$)
  - def simulate(n, t):
- 위 함수는 n × n 격자에 대해 t회 시뮬레이션 반복한 후
- Percolate할 때 열린 객체 비율의 평균, 표준편차, 95% 신뢰구간을 아래 예제와 같은 형식으로 출력
  - t회 예측치가 x1, x2, …, xt라 할 때,
  - 평균 = (x1+x2+…+xt) / t   # statistics.mean() 사용해 계산
  - 표준편차^2 = {(x1−평균)^2 + (x2−평균)^2 + … (xT−평균)^2} / (t−1) # statistics.stdev() 사용해 계산
  - 95% 신뢰구간 = [평균 - 1.96 * 표준편차 / √t , 평균 + 1.96 * 표준편차 / √t] # math.sqrt() 사용해 제곱근 계산
  - 위 값은 모두 소수점 아래 10자리로 출력 (format string ".10f"에 해당)
- 이 중 평균과 표준편차를 반환
- Weighted Quick Union 방법 사용해 구현해야 하며
- 성능: 이어지는 각 예제에 대해 10초 이내에 출력이 나오면 됨

```
>>> simulate(200,100)
mean                    = 0.5922960000
stdev                   = 0.0085377805
95% confidence interval = [0.5906225950, 0.5939694050]
```

$\mu - \frac{1.96 \times \sigma}{\sqrt{t}}$

프로그램 입출력 조건

```
>>> print(simulate(200,100))
mean                    = 0.5922960000
stdev                   = 0.0085377805
95% confidence interval = [0.5906225950, 0.5939694050]
(0.592296, 0.008537780478979858)
```

mean, stdev 반환하므로 이와 같이 출력됨

```
>>> simulate(200,100)
mean                    = 0.5920527500
stdev                   = 0.0090788103
95% confidence interval = [0.5902733032, 0.5938321968]
```

```
>>> simulate(2,10000)
mean                    = 0.6658250000
stdev                   = 0.1181512393
95% confidence interval = [0.6635092357, 0.6681407643]
```

```
>>> simulate(2,100000)
mean                    = 0.6663600000
stdev                   = 0.1179596946
95% confidence interval = [0.6656288782, 0.6670911218]
```

그 외 프로그램 구현 조건

- 작성한 코드는 파일 이름 Percolate.py에 저장
- 수업 자료와 함께 제공된 코드 필요하다면 내용 복사해서 작성한 코드 일부로 사용 가능 (예: WQU, QU, QF)
- 최종 결과물로는 Percolate.py 파일 하나만 제출하며, 이 파일만으로 코드가 동작해야 함

- import는 statistics, math, random만 할 수 있음

스마트 출결

수업 전

문서 형태
개요/이론 학습

수업 전

개요 관련 퀴즈 수행
스스로 학습 & 시행착오

40~50분

수업(화/수)

퀴즈 풀이
개요 복습

수업(화/수)

응용/심화 내용 학습
질의 응답

수업(화/수)

실습 과제 (프로그래밍)
질의 응답

~3시간

실습과제 수업 중 완료 못한 경우
당일 저녁 or 다음날까지
각자 수행해 제출

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

12:00까지 실습 & 질의응답

- 작성한 코드는 lms > 강의 콘텐츠 > 오늘 수업 > 실습 과제 제출함에 제출

- 시간 내 제출 못한 경우 내일 11:59pm까지 제출 마감
- 마감 시간 후에는 제출 불가하므로 그때까지 작성한 코드 꼭 제출하세요.