[22.01.21] 딱지 게임

TS 딱지

김 태 현

문 제

2명의 플레이어가 N*N 크기의 판에 번갈아 가며 정사각형 모양의 딱지를 내려 놓는다.
내려 놓은 딱지와 연결된 모든 딱지들이 해당 플레이어의 소유로 변한다.
한 턴에 { 1번 플레이어 add() , 2번 플레이어 add() , get() }이 수행 된다.

1. `void init(int N, int M)`

N: 게임 판 한 변의 길이
M: 딱지의 한 변의 길이 최대 값

2. `int add(int row, int col, int size, int pid)`

pid 플레이어가 (row, col) 위치에 한 변의 길이가 size인 딱지를 놓는다.

3. `int get(int row, int col)`

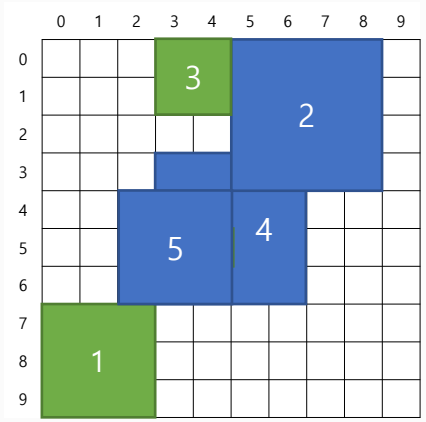
(row, col) 격자를 덮고 있는 딱지를 소유한 플레이어를 반환 (1 or 2)
딱지가 없으면 0 반환

- 10 <= N <= 100,000,000
- 1 <= M <= 10,000,000
- (0, 0) ~ (N-1, N-1)
- N은 10의 배수
- M은 N/10
- 각 테스트 케이스 최대 10,000턴
- 딱지 20,000개

Naïve

add(), 놓은 딱지와 연결된 모든 딱지들을 하나의 그룹으로 만든다.

get(), (row, col) 위치에 존재하는 딱지의 pid를 반환한다.



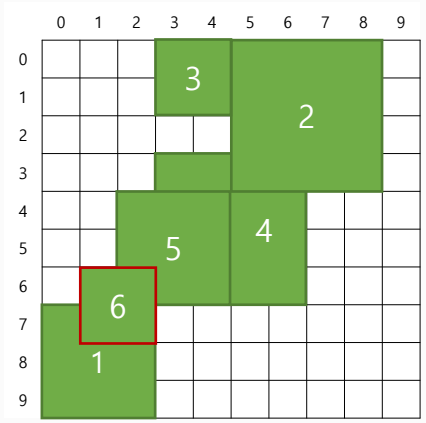
group	pid	cnt
1	1	1
2	2	3
3	1	1

딱지 번호

1

2 4 5

3



group	pid	cnt
1	1	0
2	1	5
3	1	1

2 4 5 1 6

3

add()

1) 그룹에 겹치는 딱지가 있는지 확인
2) 있으면 그룹 합침

O((딱지 검색 개수 + 그룹 이동 비용) * 호출 횟수)
= O(20,000 * 20,000)

get()

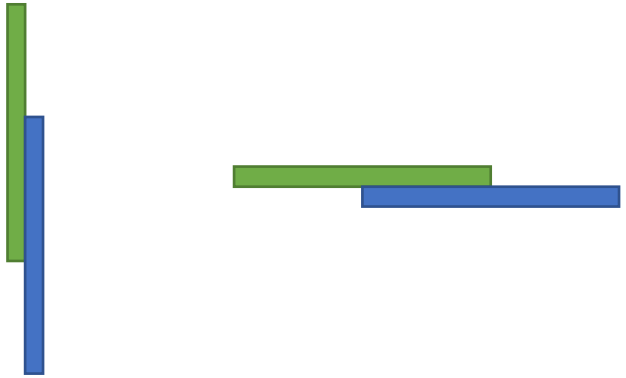
O(딱지 검색 개수 * 호출 횟수)
= O(20,000 * 10,000)

```
struct Group {
    int pid, cnt;
    <ddakji list>
}G[20,003]
```

<ddakji list>
vector : 전체 탐색 빠름
list : 그룹 이동 빠름

정사각형 겹침 판단

x, y축 독립적으로 판단
: x축, y축 둘다 겹쳐야함



겹치는 경우

(s1 <= s2 && s2 <= e1) || (s1 <= e2 && e2 <= e1)

겹치지 않는 경우

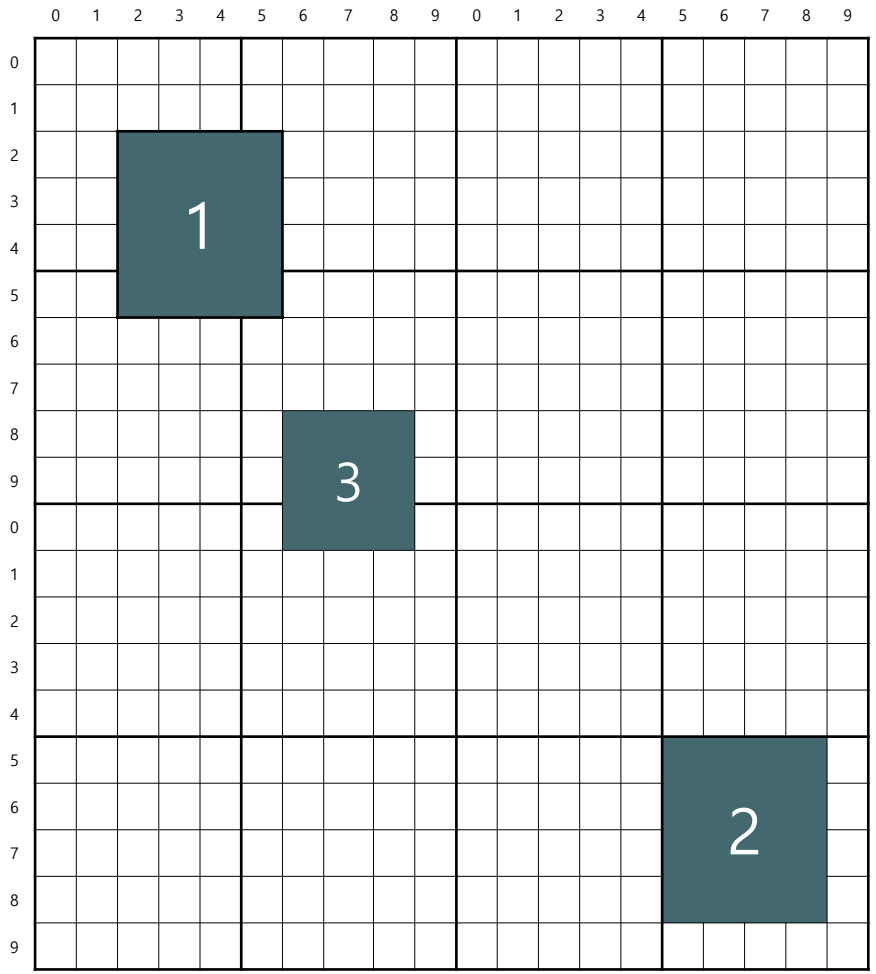
e1 < s2 || e2 < s1

최적화 포인트

1. 딱지 검색 개수 줄이기
2. 그룹 이동 비용 줄이기

최적화 포인트

1. 딱지 검색 개수 줄이기



2번 딱지를 놓고 아주 멀리 있는 1번 딱지와 비교할 필요가 없다.
겹칠 가능성이 있는 딱지들만 검색할 수 있도록 분류해서 저장

N 은 10 의 배수로만 주어진다.
첫 번째 테스트 케이스를 제외하고, M 은 N / 10 으로 주어진다.

게임 판을 M*M 크기의 10*10 개 그룹으로 분류하여 딱지가 포함되는 그룹에 등록
최대 4개에 등록

평균 검색해야 하는 딱지 개수 = 총 개수 / 그룹 개수 * 4 = 20,000 / 100 * 4 = 800

add() 검색 비용 = O(800 * 4 * 20,000)
get() 검색 비용 = O(800 * 10,000)
그룹 이동 비용은 그대로.

group	딱지
(0,0)	1
(0,1)	1
(1,0)	1
(1,1)	1, 3
(2,1)	3
(3,3)	2

vector or array

최적화 포인트 2. 그룹 이동 비용 줄이기

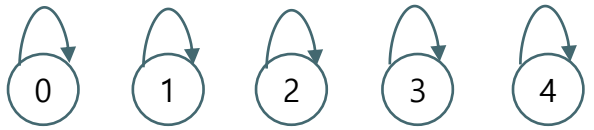
딱히 방법이 없으므로 주어진 Union Find 코드 활용 고려

Union Find

```
const int LM = 20003;
int root[LM], rank[LM];

// initialize
for (int i = 0; i < LM; i++) {
    root[i] = i;
    rank[i] = 0;
}
```

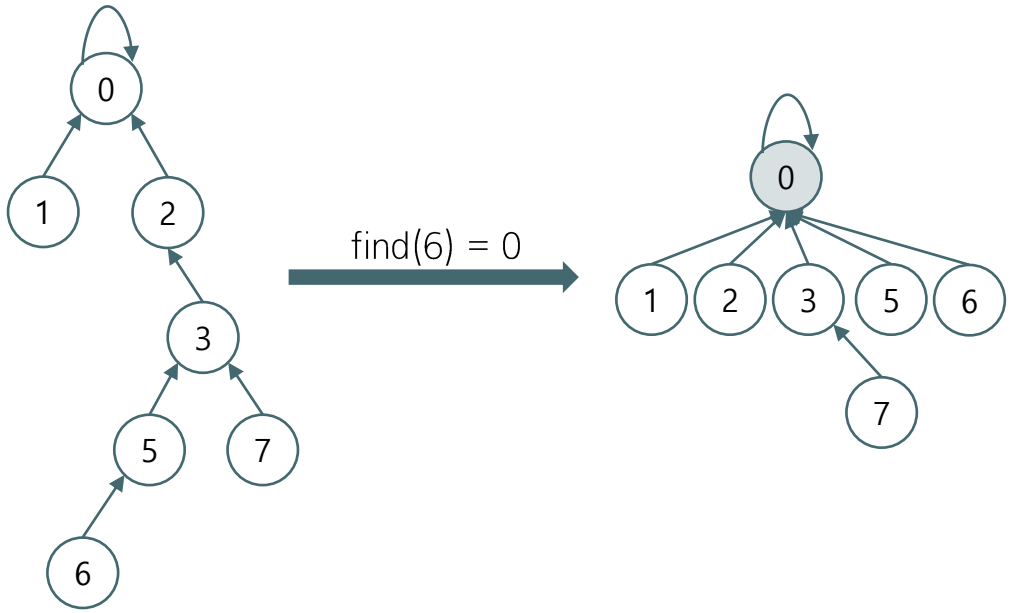
```
int find(int x) {
    if (root[x] == x) return x;
    return root[x] = find(root[x]);
}

void union(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;

    if (rank[x] < rank[y]) {
        root[x] = y;
    }
    else {
        root[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
    }
}
```

	0	1	2	3	4
root	0	1	2	3	4
rank	0	0	0	0	0



Union Find

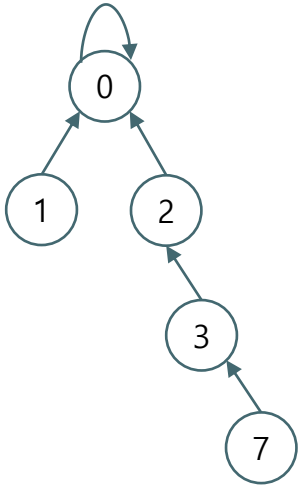
```
const int LM = 20003;
int root[LM], rank[LM];

// initialize
for (int i = 0; i < LM; i++) {
    root[i] = i;
    rank[i] = 0;
}

int find(int x) {
    if (root[x] == x) return x;
    return root[x] = find(root[x]);
}

void union(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;

    if (rank[x] < rank[y]) {
        root[x] = y;
    }
    else {
        root[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
    }
}
```

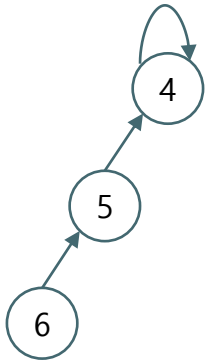


Union Find

```
const int LM = 20003;
int root[LM], rank[LM];

// initialize
for (int i = 0; i < LM; i++) {
    root[i] = i;
    rank[i] = 0;
}

int find(int x) {
    if (root[x] == x) return x;
    return root[x] = find(root[x]);
}

void union(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;

    if (rank[x] < rank[y]) {
        root[x] = y;
    }
    else {
        root[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
    }
}
```

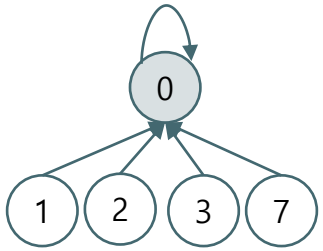
union(7, 6)

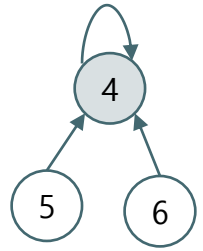




x = find(7) = 0

y = find(6) = 4

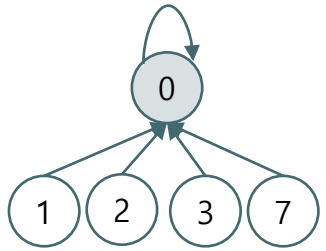


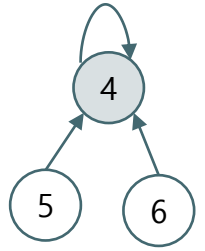


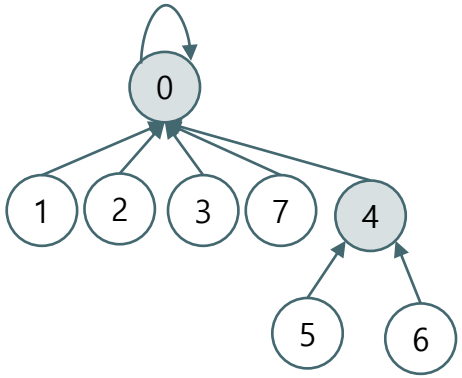
Union Find

```
const int LM = 20003;
int root[LM], rank[LM];

// initialize
for (int i = 0; i < LM; i++) {
    root[i] = i;
    rank[i] = 0;
}

int find(int x) {
    if (root[x] == x) return x;
    return root[x] = find(root[x]);
}

void union(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;

    if (rank[x] < rank[y]) {
        root[x] = y;
    }
    else {
        root[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
    }
}
```

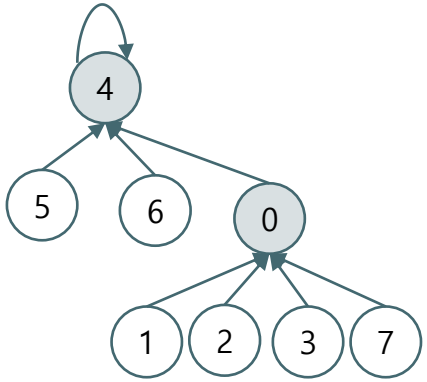
union(7, 6)







or

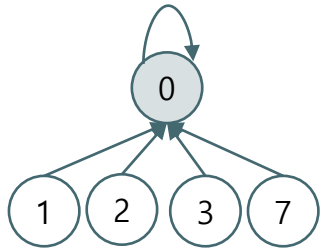


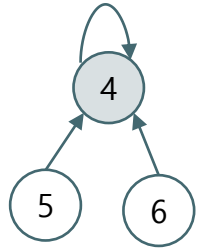
rank는 모르겠지만 루트가 다르면 하나로 연결 해주는 코드임을 확인

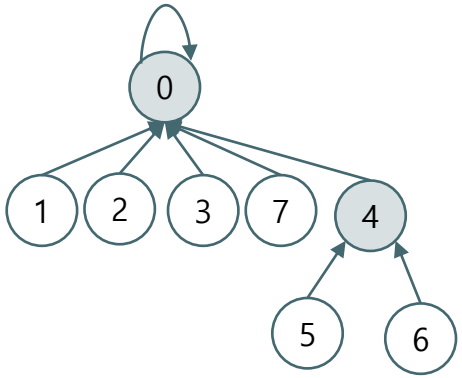
Union Find

```
const int LM = 20003;
int root[LM], rank[LM];

// initialize
for (int i = 0; i < LM; i++) {
    root[i] = i;
    rank[i] = 0;
}

int find(int x) {
    if (root[x] == x) return x;
    return root[x] = find(root[x]);
}

void union(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;

    if (rank[x] < rank[y]) {
        root[x] = y;
    }
    else {
        root[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
    }
}
```

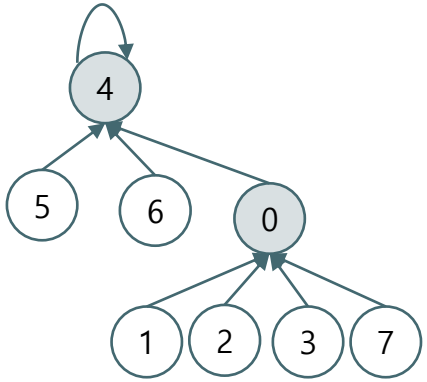
union(7, 6)







or



rank는 모르겠지만 루트가 다르면 하나로 연결 해주는 코드임을 확인

Union Find

Disjoint Set

```
const int LM = 20003;
int root[LM], rank[LM];

// initialize
for (int i = 0; i < LM; i++) {
    root[i] = i;
    rank[i] = 0;
}

int find(int x) {
    if (root[x] == x) return x;
    return root[x] = find(root[x]);
}

void union(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;

    if (rank[x] < rank[y]) {
        root[x] = y;
    }
    else {
        root[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
    }
}
```

집합을 그래프로 관리하며

1. 두 노드가 한 개의 집합에 존재하는지 판별
2. 두 노드가 속한 집합을 한 개로 합침

시간 복잡도 worst O(N) => 최적화 O(상수)

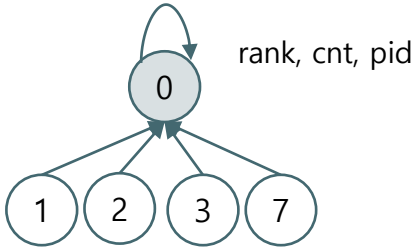
1. Union by Rank
2. Path Compression

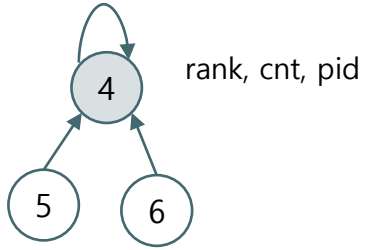
Union Find 활용 그룹 이동

놓은 딱지와 겹치는 딱지의 그룹을 union()으로 합친다.

그 과정에서 각 그룹의 개수와 소유 플레이어, 플레이어 별 총 소유 개수를 추가적으로 관리한다.

각 그룹의 root에 유효한 정보들이 들어간다. (rank, cnt, pid)





player	1	2
cnt		


감사합니다