

JooHyun – Lee (comkiwer)

# TS좌석배정

Hancom Education Co. Ltd.

# Problem

좌석 배정을 하는 프로그램을 작성한다.

가로 방향으로  $W$ 개, 세로 방향으로  $H$ 개인 직사각형 모양으로 좌석들이 나열되어 있다.

$(1 \leq W \leq 30,000, 1 \leq H \leq 30,000, W * H \leq 30,000)$

각 좌석에는 1부터  $W * H$  까지의 수가 중복 없이 적혀 있다.

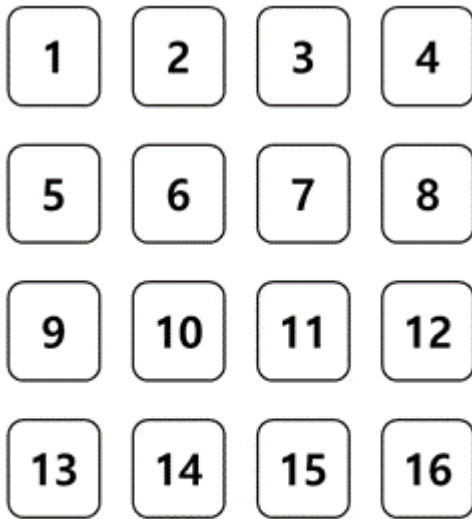
좌석 번호는 상단 좌측 좌석이 1번 좌석이고,  
오른쪽으로 갈수록 좌석 번호가 1씩 증가한다.

오른쪽 끝에 도달하면,  
다음 줄 좌측부터 같은 방법으로 1씩 증가하여  $W * H$  까지 적혀 있다.

[Fig. 1] 은  $W = 4$ ,  $H = 4$  인 경우의 좌석 번호를 나타낸 그림이다.

각 사각형은 좌석을 의미하며 사각형 내에 적힌 수는 해당 좌석의 좌석 번호이다.

좌측 상단 좌석이 1번 좌석이며 우측 하단 좌석이 16 번 좌석이다.

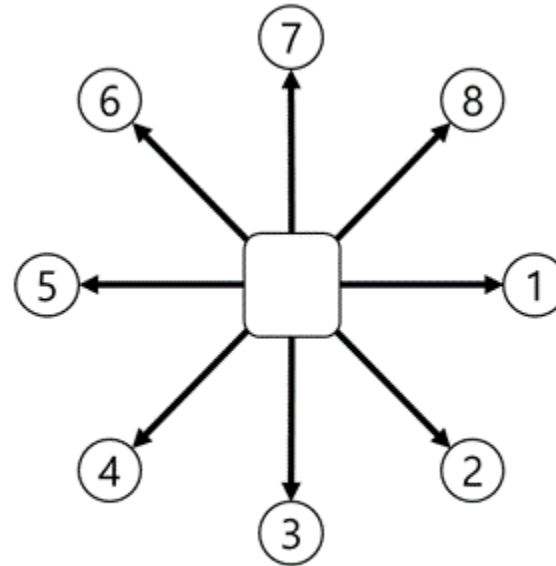


1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

[Fig. 1]

관람객은 입장하는 순서대로 좌석 하나를 선택하며  
아래와 같은 우선 순위로 좌석이 배정된다.

- 1) 선택한 좌석이 비어있는 경우 해당 좌석이 배정된다.
- 2) 선택한 좌석이 비어있지 않은 경우 해당 좌석을 기준으로  
상하좌우대각으로 총 8가지 방향에서  
가장 가까운 비어있는 좌석이 배정된다.
- 3) 만약 가장 가까운 빈 좌석이 2개 이상이라면  
[Fig. 2] 에서 각 방향이 가리키는 수가  
가장 작은 방향의 좌석이 배정된다.
- 4) 만약 8가지 방향에서 더 이상 비어있는  
좌석이 없는 경우 좌석 배정에 실패한다.



[Fig. 2]

**상하좌우대각으로 인접한 좌석 간의 거리는 1이다.**

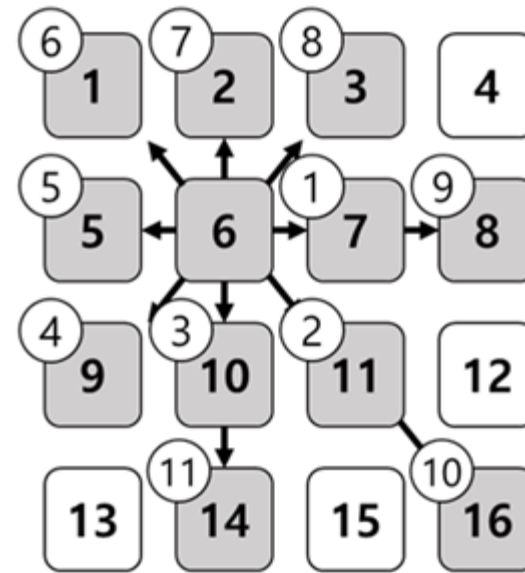
[Fig. 3] 는  $W = 4, H = 4$  에서 관람객이 6번을 선택한 경우 좌석을 배정하는 우선순위를 나타내는 그림이다.

각 좌석의 좌측 상단에 표시된 수가 작을수록 좌석을 배정하는 우선순위가 높다.

6번 좌석이 비어있는 경우 해당 좌석이 배정되며,  
비어있지 않은 경우 8가지 방향의 좌석들 중에서  
좌석의 좌측 상단에 표시된 수가  
가장 작은 비어있는 좌석이 배정된다.

4, 12, 13, 15번 좌석의 경우

6번 좌석을 기준으로 8가지 방향에 놓여있지  
않으므로 해당 좌석들은 배정되지 않는다.



[Fig. 3]

아래 API 설명을 참조하여 각 함수를 구현하라.

```
void init(int W, int H)
```

각 테스트 케이스의 맨 처음에 호출된다.

W와 H는 각각 전체 좌석의 가로, 세로 방향의 좌석 개수이다.

W \* H 값은 30,000을 넘지 않는다.

처음에 모든 좌석은 비어있다.

### *Parameters*

W : 가로 방향의 좌석 개수 ( $1 \leq W \leq 30,000$ )

H : 세로 방향의 좌석 개수 ( $1 \leq H \leq 30,000$ )

```
int selectSeat(int mSeatNum)
```

mSeatNum은 새로 입장한 관람객이 선택한 좌석 번호이다.

아래의 우선순위에 따라 좌석을 배정하고, 배정된 좌석 번호를 반환한다.

배정에 실패한 경우 0을 반환한다.

- 1) 선택한 좌석이 비어있는 경우 해당 좌석이 배정된다.
- 2) 선택한 좌석이 비어있지 않은 경우 해당 좌석을 기준으로  
상하좌우대각으로 총 8가지 방향에서 가장 가까운 비어있는 좌석이 배정된다.
- 3) 만약 가장 가까운 빈 좌석이 2개 이상이라면  
[Fig. 2] 에서 각 방향이 가리키는 수가 가장 작은 방향의 좌석이 배정된다.
- 4) 만약 8가지 방향에서 더 이상 비어있는 좌석이 없는 경우 좌석 배정에 실패한다.

### **Parameters**

mSeatNum : 관람객이 선택한 좌석 번호 ( $1 \leq \text{mSeatNum} \leq W * H$ )

### **Returns**

좌석 배정에 성공한 경우 배정된 좌석 번호, 그렇지 않은 경우 0



### [제약사항]

1. 각 테스트 케이스 시작 시 `init()` 함수가 1회 호출된다.
2. 각 테스트 케이스에서 `selectSeat()` 함수의 총 호출 수는 전체 좌석 수 이하이다.
3. 모든 테스트 케이스에서 전체 좌석 수는 30,000 개를 넘지 않는다.

# Problem analysis

아래는 각 함수 호출 이후 좌석 배정 상태를 나타낸다.

각 사각형은 좌석을 의미하며 사각형 내에 적힌 수는 해당 좌석의 좌석 번호이다.

노란색으로 표시된 좌석은 해당 함수 호출 순서에 배정된 좌석을 나타내며,

주황색으로 표시된 좌석은 관람객이 선택한 좌석이 비어 있지 않음을 나타낸다.

회색으로 표시된 좌석은 이미 배정된 좌석을 나타낸다.

```
[#1] init(4, 4)
```

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

입장한 관람객이 1번 좌석을 선택하였고  
해당 좌석이 비어 있으므로 1번 좌석이 배정된다.

[#2] selectSeat(1)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

입장한 관람객이 1번 좌석을 선택하였으나  
#2에서 배정된 좌석으로 비어 있지 않아 2번 좌석이 배정된다.

[#3] selectSeat(1)



[#4] selectSeat(5)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

[#5] selectSeat(9)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

[#6] selectSeat(11)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

입장한 관람객이 1번 좌석을 선택하였으나 #2에서 배정된 좌석으로 비어 있지 않다.  
좌석 배정의 우선순위에 따라 6번 좌석이 배정된다.

[#7] selectSeat(1)



입장한 관람객이 6번 좌석을 선택하였으나 #7에서 배정된 좌석으로 비어있지 않다.  
좌석 배정의 우선순위에 따라 7번 좌석이 배정된다.

[#8] selectSeat(6)





[#9] selectSeat(6)



[#10] selectSeat(7)



[#11] selectSeat(7)



[#12] selectSeat(7)



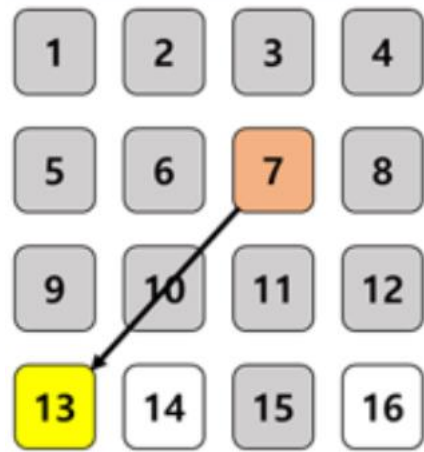
[#13] selectSeat(7)



[#14] selectSeat (7)



[#15] selectSeat (7)

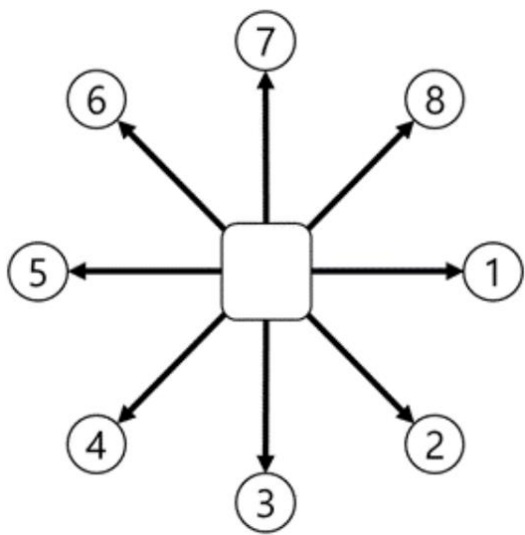


입장한 관람객이 7번 좌석을 선택하였으나 #8에서 배정된 좌석으로 비어있지 않다.  
8가지 방향에 더 이상 비어있는 좌석이 없어 좌석 배정에 실패한다.

[#16] selectSeat (7)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

- 실질적으로 작성할 함수는 좌석을 배정하는 `selectSeat(int sn)` 함수뿐이다.
- 좌석 배정의 우선순위는 다음과 같다.
  1. 선택한 좌석이 비어 있다면 좌석을 배정한다.
  2. 선택한 좌석이 비어 있지 않은 경우 주변 8방향으로 가장 가까운 빈 자리를 배정한다.  
가장 가까운 빈 자리가 2개 이상인 경우 아래 그림의 우선순위를 따른다.



- 좌석이 배치된 크기는  
가로방향으로  $W(1 \sim 30,000)$ 개,  
세로방향으로  $H(1 \sim 30,000)$ 개가 될 수 있는데  
좌석수  $W \times H$ 의 한계도 30,000이하이다.
- $W \times H$ 개의 배열을 정적배열로 선언할 수 없으므로 다른 방법을 생각해야 한다.  
하나의 대안으로 동적으로 선언하는 것 등을 생각할 수 있다.
- 좌석수가 30,000 이라고 할 때,  
`selectSeat()` 함수 호출 역시 30,000 이 될 수 있다.

- 선택한 좌석이 비어 있는 경우에는 처리가 간단하다.  
하지만 비어 있지 않은 경우 우선순위에 따라 찾아야 한다.  
단순히 BFS(DFS)등으로 찾는다고 할 때 시간 복잡도는 어떻게 될 까?
- 오른쪽 그림을 예로 보자. ( $W=2$ ,  $H = 15,000$ ,  $\text{selectSeat}()$ 호출수  $W*H$ )
  - ✓  $\text{selectSeat}(\text{별색자리})$ 만 15002번 호출한 그림이다.  
시간 복잡도는 15,000의 제곱이다.
  - ✓ 이후 14,998번  $\text{selectSeat}(\text{별색자리})$ 를 호출하면  
시간 복잡도는 역시 15,000제곱이 된다.
  - ✓ 15,000의 제곱은 225,000,000이므로  
50개의 TC를 수행하면 시간초과에 걸릴 수 있다.
  - ✓ 어떻게 해야 할 까?

15,000

*	*
*	*
*	
*	
...	...
*	
*	
*	

# Solution sketch



- 동적배열을 잡아 좌석 배치도를 작성하는 방법도 있지만 우리는 2차원 좌표를 1차원 좌표로, 1차원 좌표를 2차원으로 변환하며 처리하는 방법을 사용해 보자.
- 이를 위하여 좌석번호에서 1을 빼서 **0\_base**로 바꾸어 처리한다.
- 차원간 좌표 변환은 아래와 같다. ( $w = 5, h = 3$  인 예)

**r** : 2차원 행 좌표

**c** : 2차원 열좌표

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14

**k** : 일차원좌표

- 2차원 좌표를 1차원 좌표로  
 $k = r * W + c$
- 1차원 좌표를 2차원 좌표로  
 $r = k / W$   
 $c = k \% W$

- 각 좌석을 다음과 같은 클래스로 정의해 보자.  
각 좌석에 방향별로 인접한 좌석의 번호를 저장한다면  
2차원 테이블이 없어도 탐색이 가능하다.

```
struct Seat {  
    int r, c, id; // 2차원 좌표 행(r) 열(c), 좌석번호 id=r*W+c  
    int flag;     // 빈좌석(0), 배정됨(1)  
    int adj[8];   // 방향별 인접한 노드의 id (1차원좌표)  
    void init(int nid) { // 좌석번호를 1차원 좌표 nid로 받아 초기화 하기  
        id = nid, r = id / W, c = id % W, flag = 0;  
        for (rint i = 0; i < 8; ++i) {  
            int nr = r + dr[i], nc = c + dc[i];  
            if (nr < 0 || nr >= H || nc < 0 || nc >= W)  
                adj[i] = -1; // 경계를 벗어난 경우  
            else  
                adj[i] = nr * W + nc; // 이웃한 좌석의 좌석 번호 id  
        }  
    }  
}srr[LM];
```

**c** : 2차원 열좌  
**r** : 2차원 행 좌표

	0	1	2
0	0	1	2
1	5	6	7
2	10	11	12

- 현재 좌석이 이미 배정된 경우 인접 좌석 번호가 저장된 `adj[]`를 따라 이동하면 원하는 방향의 빈 좌석 또는 경계를 찾을 수 있다.

```
struct Seat {  
    int r, c, id; // 2차원 좌표 행(r) 열(c), 좌석번호  $id=r*W+c$   
    int flag; // 빈좌석(0), 배점됨(1)  
    int adj[8]; // 방향별 인접한 노드의 id (1차원좌표)
```

- 그런데 문제 분석에서와 같이 `selectSeat()` 호출시마다 아주 긴 경로를 매번 탐색한다면 시간적 낭비가 많을 것으로 예상된다.



- 그런데 한번 탐색한 결과를 경로에 있는 각 자리에 업데이트 한다면 중복탐색을 효과적으로 줄일 수 있다.
- 예를 들어 아래 모든 좌석이 배정된 상태라고 하자.



- 이제 0번 좌석을 호출한 경우 각 좌석별 adj[0]을 다음과 같이 업데이트 할 수 있다.



- 만일 빈 자리를 찾은 경우라면 빈 자리 번호로 업데이트 하면 된다.



- 이방법은 union-find와 같은 원리로 작동하므로  
시간 복잡도는  $O(\log^*N : \text{로그-스타}N)$ 이다. (반복 로그라고도 한다.)  
이는 거의 상수시간으로 취급된다.

- 구현방법으로는 재귀도 가능할 것이나  
재귀의 깊이가 30,000까지 될 수 있으므로  
스택 1MB제한에 걸릴 위험이 있다.

찾아 갈때는 별도의 배열에 방문한 좌석번호를 저장하고  
탐색이 끝난 후 배열을 순회하며 업데이트 할 수 있다.

```
for (i = 0; i < 8; ++i) {  
    nid = sn;  
    for (j = 0; nid > -1 && srr[nid].flag; ++j) {  
        A[j] = nid;           // 방문한 id를 목록에 저장  
        nid = srr[nid].adj[i];  
        if (nid < 0) break;  
    }  
    for (--j; j >= 0; --j) { // 돌아오며 인접 id 갱신  
        srr[A[j]].adj[i] = nid;  
    }  
    ...  
}
```

- 함수 별 할 일을 정리해 보자.

```
void init(int W, int H)
```

- 좌석의 가로 크기 세로 크기 등을 초기화 한다.

```
::W = W, ::H = H, N = W * H;
```

- 1차원 0\_base기준으로 Seat클래스를 이용하여 각 좌석을 초기화 한다.

```
for (rint i = 0; i < N; ++i)  
    srr[i].init(i);
```

```
int selectSeat(int sn)
```

- sn자리가 빈 자리인 경우 바로 처리한다.

```
    if (srr[--sn].flag == 0) {    // sn 자리가 비어있는 경우
        srr[sn].flag = 1;        // 자리가 배정되었음을 표시
        return srr[sn].id + 1;
    }
```

- 그렇지 않은 경우 각 방향으로 빈자리를 찾아 본다.

- ✓ 우선순위에 따라 8방향 탐색을 진행하면서 가장 가까운 빈자리를 찾는다.
- ✓ 자리를 탐색할 때, 방문한 각 노드의 adj[]를 업데이트 한다.
- ✓ 찾은 경우 배정표시를 하고 자리 번호의 (행번호+열번호+1) 를 반환한다.  
(0\_base로 작업하였으므로 +1하여 반환하는 것에 유의한다.)
- ✓ 못 찾은 경우 0을 반환한다.



## [Summay]

- 함수의 개수가 적고 문제가 단순한 경우  
idea를 필요로 하는 경우가 많다.  
(idea생각해내는 것이 그만큼 어렵다는 반증)
- 실제 시험에서는 union-find아이디어를 통한  
adj[]업데이트를 수행하지 않고 set, map을 사용하거나  
부분 최적화(너비1 또는 높이1인 경우 예외처리 등) 하는 것으로  
합격한 예도 있다.

아이디어도 중요하지만 구현의 완성이 더 중요하다.

# Code example

# Code example

## TS좌석배정

```
#define rint register int
const int LM = 30010;
int W, H, N;
int dr[] = {0, 1, 1, 1, 0, -1, -1, -1};
int dc[] = {1, 1, 0, -1, -1, -1, 0, 1};
struct Seat {
    int r, c, id; // 좌표, id=r*W+c
    int flag;      // 빈좌석(0), 배정됨(1)
    int adj[8];    // 방향별 인접한 노드의 id
    void init(int nid) {
        id = nid, r = id / W, c = id % W, flag = 0;
        for (rint i = 0; i < 8; ++i) {
            int nr = r + dr[i], nc = c + dc[i];
            if (nr < 0 || nr >= H || nc < 0 || nc >= W)
                adj[i] = -1; // 경계를 벗어난 경우
            else
                adj[i] = nr * W + nc; // 이웃한 id
        }
    }
}srr[LM];
```

# Code example

## TS좌석배정

```
inline int abs(int a) { return a < 0 ? -a : a; }
inline int max(int a, int b) { return a > b ? a : b; }
void init(int W, int H){
    ::W = W, ::H = H, N = W * H;
    for (rint i = 0; i < N; ++i) srr[i].init(i);
}

int A[LM]; // 한방향으로 방문한 id목록
int selectSeat(int sn){
    if (srr[--sn].flag == 0) { // sn 자리가 비어있는 경우
        srr[sn].flag = 1;
        return srr[sn].id + 1;
    }
    int retDist = LM, retID = -1; // 기본 결과값
    rint r = sn / W, c = sn % W; // sn의 행, 열 좌표
    rint i, j, nid;
```

# Code example

## TS좌석배정

```
for (i = 0; i < 8; ++i) {
    nid = sn;
    for (j = 0; nid > -1 && srr[nid].flag; ++j) {
        A[j] = nid;           // 방문한 id를 목록에 저장
        nid = srr[nid].adj[i];
        if (nid < 0) break;
    }
    for (--j; j >= 0; --j) { // 돌아오며 인접 id 갱신
        srr[A[j]].adj[i] = nid;
    }
    if (nid < 0) continue;   // 경계를 벗어난 경우
    int dist = max(abs(r - srr[nid].r), abs(c - srr[nid].c));
    if (dist < retDist)      // 더 가까운 노드라면 선택하기
        retDist = dist, retID = nid;
}
if (retID < 0) return 0;    // 빈 자리를 못 찾은 경우
srr[retID].flag = 1;        // 자리가 배정되었음을 표시
return retID + 1;
}
```

**Thank you.**