Chessland Attacks

Objective
Give practice with dynamic memory.
Give practice with array lists.

Story
In a nearby land there is quite the border dispute between many royal families. Each one has a champion chess piece (a rook) on a large 2D grid. The royal families would like to know how much peril their rook currently faces. Rooks can only attack other pieces in their rank or file (row/column). Additionally, a rook can only attack another rook if there is no other piece(s) directly in between the two.

Unfortunately, the board is so large it is hard to eyeball whether 2 rooks can attack each other. You offer to solve the rook threat question in exchange for some coins.

Problem
You will be given the rank and file of many rooks. Your objective is to determine how many and which rooks threaten each piece.

Input
Input will begin with a line containing 1 integers, ***n*** ($1 \leq n \leq 100{,}000$), representing the number of rooks to process. The first line will be followed by ***n*** more lines. The ***i***-th following line contains two integers, ***r*** and ***c*** ($1 \leq r, c \leq 1{,}000{,}000{,}000$), representing the rank and file of the ***i***-th rook. The i-th rook will have the ID of i.

No two rooks will be in the same spot. The number of unique ranks and files will be at most 10,000.

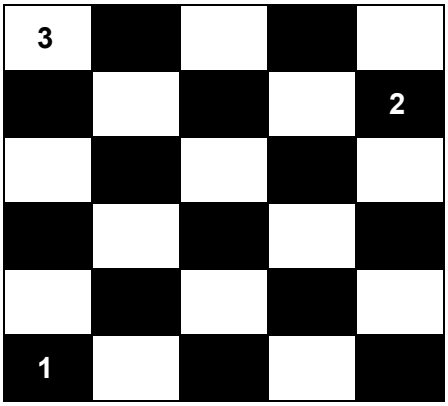
Output
Output should contain ***n*** lines. The i-th line will start with an integer t, representing the number of rooks that can attack the i-th rook. Following t should be t integers on the same line. Each integer will represent the ID of the rook that can attack the current rook. Each rook ID should occur only once in the line. All integers in the line should be separated by spaces. Order of the IDs does not matter.

Sample Input	Sample Output
3 1 1 5 5 6 1	1 3 0 1 1
4 1 1 2 1 4 1 2 2	1 2 3 3 1 4 1 2 1 2

Explanation

Case 1

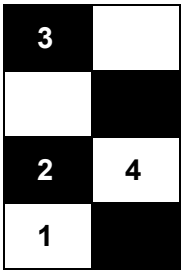
There are 3 rooks. The grid looks like the following,



1 and 3 can mutually attack each other. 2 cannot attack anyone.

Case 2

There are 3 rooks. The grid looks like the following,



1 and 2 can mutually attack each other. 3 and 2 can mutually attacke each other. 4 and 2 can mutually attack each other.

Hints

Array list of Array lists: I recommend using a dynamically sized array for each row where only the ids and locations of the rooks are stored inside. Trying to create arrays based on the size of the board is not feasible. Also recommend using an arraylist to hold all the rows, and use some integer to know which row corresponds to each spot of the array list.

The struct can look like the following,

```
struct rankArrayList {
   int size, cap;
   struct rank * ranks;
}
```

Rank struct: You can make a struct that stores the information for each row. The struct can look like the following

```
struct rank {
   int location;
   int num_pieces;
   int capacity;
   struct piece * array;
};
```

Piece struct: You can make a struct that stores the information for each piece. The struct can look like the following

```
struct piece {
   int rank, file, id;
};
```

Doubling: It's good practice to double the size of the array list when increasing capacity. It ensures that the number of doublings occurs logarithmically with respect to the number of elements.

Avoid Needless Checks: Storing pieces by their rank or file allows your program to quickly check the important pieces for any possible threats (remember rooks can only attack in the same row or column).

Storing Pieces Twice: It is a good idea to not only store the pieces by their rank but to also store the pieces by their file to void needlessly checking which files a piece can attack.

Function Recommendations: I recommend making separate array list functions for the array list of ranks and the rank themselves. The following prototypes are recommended

```
void addPieceToBoard(struct rankArrayList * board, struct piece *
newPiece);
void expandBoard(struct rankArrayList * board);
void cleanBoard(struct rankArrayList * board);
struct rankArrayList * createBoard();

void addPieceToRank(struct rank * rank, struct piece * newPiece);
void expandRank(struct rank * curRank);
void cleanRank(struct rank * curRank);
struct rank * createRank();
```

Grading Criteria

- Read/Write from/to **standard** input/output (e.g. scanf/printf and no FILE *)
    - 10 points
- Good comments, whitespace, and variable names
    - 15 points
- Only check the pieces in the same row/column to increase performance.
    - 10 points
- Store each piece in the program until all input has been read
    - 5 points
- Check the four directions for each piece.
    - 10 points
- Programs will be tested on 10 cases
    - 5 points each

Programs with extra output will be docked 10 points.

No points will be awarded to programs that do not compile using "gcc -std=gnu11 -lm".

Sometime a requested technique will be given, and solutions without the requested technique will have their maximum points total reduced. For this problem use dynamic memory (calloc, malloc, realloc, free). ***Without this programs will earn at most 50 points!***

Any case that causes a program to return a non-zero return code will be treated as wrong. Additionally, any case that takes longer than the maximum allowed time (the max of {5 times my solutions time, 10 seconds}) will also be treated as wrong.

No partial credit will be awarded for an incorrect case.