Welcome to Codefest 4.0 Spring 2023

"First, solve the problem. Then, write the code."
– John Johnson

Senior Programming Competition

Organized by GDSC
FCIT
University of the Punjab

CamScanner

Dragon Maze

Problem

You are the prince of Dragon Kingdom and your kingdom is in danger of running out of power. You must find power to save your kingdom and its people. An old legend states that power comes from a place known as Dragon Maze. Dragon Maze appears randomly out of nowhere without notice and suddenly disappears without warning. You know where Dragon Maze is now, so it is important you retrieve some power before it disappears.

Dragon Maze is a rectangular maze, an **N** x **M** grid of cells. The top left corner cell of the maze is (0,0) and the bottom right corner is (**N**-1, **M**-1). Each cell making up the maze can be either a dangerous place which you never escape after entering, or a safe place that contains a certain amount of power. The power in a safe cell is automatically gathered once you enter that cell, and can only be gathered once. Starting from a cell, you can walk up/down/left/right to adjacent cells with a single step.

Now you know where the entrance and exit cells are, that they are different, and that they are both safe cells. In order to get out of Dragon Maze before it disappears, you must walk from the entrance cell to the exit cell taking as few steps as possible. If there are multiple choices for the path you could take, you must choose the one on which you collect as much power as possible in order to save your kingdom.

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow.

Each test case starts with a line containing two integers **N** and **M**, which give the size of Dragon Maze as described above. The second line of each test case contains four integers $\mathbf{en_x}$, $\mathbf{en_y}$, $\mathbf{ex_x}$, $\mathbf{ex_y}$, describing the position of entrance cell ($\mathbf{en_x}$, $\mathbf{en_y}$) and exit cell ($\mathbf{ex_x}$, $\mathbf{ex_y}$). Then **N** lines follow and each line has **M** numbers, separated by spaces, describing the **N** x **M** cells of Dragon Maze from top to bottom. Each number for a cell is either -1, which indicates a cell is dangerous, or a positive integer, which indicates a safe cell containing a certain amount of power.

Output

For each test case, output one line containing "Case #x: y", where x is the case number (starting from 1). If it's possible for you to walk from the entrance to the exit, y should be the maximum total amount of power you can collect by taking the fewest steps possible. If you cannot walk from the entrance to the exit, y should be the string "Mission Impossible." (quotes for clarity). Please note that the judge requires an exact match, so any other output like "mission impossible." or "Mission Impossible" (which is missing the trailing period) will be judged incorrect.

Limits

Time limit: 30 seconds per test set.
Memory limit: 1GB.

The amount of power contained in each cell will not exceed 10,000.
$1 \le \mathbf{T} \le 30$.
$0 \le \mathbf{en_x}, \mathbf{ex_x} < \mathbf{N}$.
$0 \le \mathbf{en_y}, \mathbf{ex_y} < \mathbf{M}$.

Test set 1 - Visible

$1 \le \mathbf{N}, \mathbf{M} \le 10$.

CamScanner

Test set 2 - Hidden 2

$1 \le N, M \le 100$.

Sample

Sample Input

```
2
2 3
0 2 1 0
2 -1 5
3 -1 6
4 4
0 2 3 2
-1 1 0 2
1 -1 1 -1
2 -1 -1 1
1 1 1 1
```

Sample Output

```
Case #1: Mission Impossible.
Case #2: 7
```

CamScanner

Hex

3

Problem

This problem was inspired by a board game called Hex, designed independently by Piet Hein and John Nash. It has a similar idea, but does not assume you have played Hex.

This game is played on an **N**x**N** board, where each cell is a hexagon. There are two players: Red side (using red stones) and Blue side (using blue stones). The board starts empty, and the two players take turns placing a stone of their color on a single cell within the overall playing board. Each player can place their stone on any cell not occupied by another stone of any color. There is no requirement that a stone must be placed beside another stone of the same color. The player to start first is determined randomly (with equal probability among the two players).

The upper side and lower sides of the board are marked as red, and the other two sides are marked as blue. The goal of the game is to form a connected path of one player's stones connecting the two sides of the board that have that player's color. The first player to achieve this wins. Note that the four corners are considered connected to both colors.

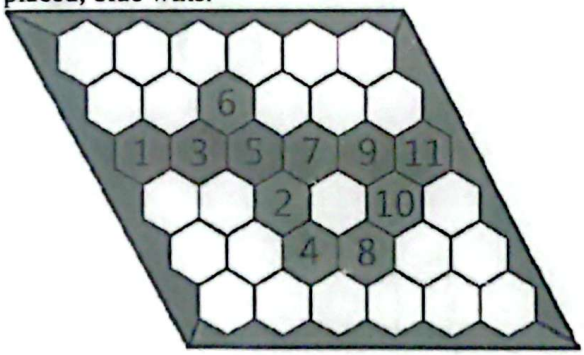
The game ends immediately when one player wins.

Given a game state, help someone new to the game determine the status of a game board. Say one of the following:

- "**Impossible**": If it was impossible for two players to follow the rules and to have arrived at that game state.
- "**Red wins**": If the player playing the red stones has won.
- "**Blue wins**": If the player playing the blue stones has won.
- "**Nobody wins**": If nobody has yet won the game. Note that a game of Hex can't end without a winner!

Note that in any impossible state, the only correct answer is "Impossible", even if red or blue has formed a connected path of stones linking the opposing sides of the board marked by his or her colors.

Here's a an example game on a 6x6 gameboard where blue won. Blue was the first player to move, and placed a blue stone at cell marked as 1. Then Red placed at cell 2, then blue at cell 3, etc. After the 11th stone is placed, blue wins.



Input

The first line of input gives the number of test cases, **T**. **T** test cases follow. Each test case start with the size of the side of the board, **N**. This is followed by a board of **N** rows and **N** columns consisting of only 'B', 'R' and '.' characters. 'B' indicates a cell occupied by blue stone, 'R' indicates a cell occupied by red stone, and '.' indicates an empty cell.

Output

CamScanner

For each test case, output one line containing "Case #x: y", where x is the case number (starting from 1) and y is the status of the game board. It can be "Impossible", "Blue wins", "Red wins" or "Nobody wins" (excluding the quotes). Note that the judge is case-sensitive, so answers of "impossible", "blue wins", "red wins" and "nobody wins" will be judged incorrect.

Limits

Time limit: 30 seconds per test set.
Memory limit: 1GB.

$1 \le T \le 100$.

Test set 1 - Visible

$1 \le N \le 10$.

Test set 2 - Hidden

$1 \le N \le 100$.

Sample

Sample Input

```
7
1
.
1
B
1
R
2
BR
BB
4
BBBB
BBB.
RRR.
RRRR
4
.BBB.
.BBB
RRR
RRRR
6
......
..R...
BBBBBB
..R.R.
..RR..
......
```

Sample Output

```
Case #1: Nobody wins
Case #2: Blue wins
Case #3: Red wins
Case #4: Impossible
Case #5: Blue wins
Case #6: Impossible
Case #7: Blue wins
```

Ignore all my comments

Problem

Good programmers write fabulous comments. Igor is a programmer and he likes the old C-style comments in `/* ... */` blocks. For him, it would be ideal if he could use this style as a uniform comment format for all programming languages or even documents, for example Python, Haskell or HTML/XML documents.

Making this happen doesn't seem too difficult to Igor. What he will need is a comment pre-processor that removes all the comment blocks in `/*`, followed by comment text, and by another `*/`. Then the processed text can be handed over to the compiler/document renderer to which it belongs—whatever it is.

Igor's pre-processor isn't quite that simple, though. Here are some cool things it does:

- The comments the pre-processor reads can be nested the same way brackets are nested in most programming languages. It's possible to have comments inside comments. For example, the following code block has an outer level of comments that should be removed by the comment pre-processor. The block contains two inner comments.

```
printf("Hello /* a comment /* a comment inside comment */
        inside /* another comment inside comment */
        string */ world");
```

After the pre-process step, it becomes:

```
printf("Hello  world");
```

- Igor recognizes comments can appear anywhere in the text, including inside a string `"/*...*/"`, a constant number `12/*...*/34` or even in a character escape `\/*...*/n`

Or more formally:

```
text:
  text-piece
  text-piece remaining-text
text-piece:
  char-sequence-without-/*
  empty-string
remaining-text:
  comment-block text

comment-block:
  /* comment-content */
comment-content:
  comment-piece
  comment-piece remaining-comment
comment-piece:
  char-sequence-without-/*-or-*/
  empty-string
remaining-comment:
  comment-block comment-content

char:
  letters
  digits
  punctuations
  whitespaces
```

CamScanner

Our pre-processor, given a **text**, removes all **comment-block** instances as specified.

Notes

- Igor only needs to remove the comment in one pass. He doesn't remove additional comment blocks created as a result of the removal of any comment block. For example:

```
//*no recursion*/* file header */
```

should generate:

```
/* file header */
```

- The * character in any /* or /*cannot be re-used in another /* or */. For example the following does **NOT** form a proper comment block

```
/*/
```

Input

A text document with comment blocks in /* and */. The input file is valid. It follows the specification of **text** in the problem statement. The input file always terminates with a newline symbol.

Output

We only have one test case for this problem. First we need to output the following line.

```
Case #1:
```

Then, print the document with all comments removed, in the way specified in the problem statements. Don't remove any spaces or empty lines outside comments.

Limits

Time limit: 30 seconds per test set.
Memory limit: 1GB.

The input program contains only:

- Letters: a-z, A-Z,
- Digits: 0-9
- Punctuation: ~ ! @ # % ^ & * ( ) - + = : ; " ' < > , . ? | / \ { } [ ] _
- Whitespace characters: space, newline

Test set 1 - Visible

The small input contains a program of less than 2k bytes.

Test set 2 - Hidden

The large input contains a program of less than 100k bytes.

Sample

Sample Input

```
//*no recursion*/* file header
**********/***********
```

Sample Output

```
Case #1:
/* file header
```

CamScanner

```
* Sample input program *
**********/************
*/
int spawn_workers(int worker_count) {
  /* The block below is supposed to
spawn 100 workers.
     But it creates many more.
     Commented until I figure out why.
  for (int i = 0; i < worker_count;
++i) {
    if(!fork()) {
      /* This is the worker. Start
working. */
      do_work();
    }
  }
  */
  return 0; /* successfully spawned 100
workers */
}

int main() {
  printf("Hello /*a comment inside
string*/ world");
  int worker_count = 0/*octal
number*/144;
  if (spawn_workers(worker_count) != 0)
{
    exit(-1);
  }
  return 0;
}
```

```
************************
*/
int spawn_workers(int worker_count) {

  return 0;
}

int main() {
  printf("Hello  world");
  int worker_count = 0144;
  if (spawn_workers(worker_count) != 0)
{
    exit(-1);
  }
  return 0;
}
```

7

CamScanner

Meet and party

8

Problem

Little Sin lives in a Manhattan-grid city, a 2D plane where people can only go north, west, south or east along the grid. The distance from (x1, y1) to (x2, y2) is |x1 - x2| + |y1 - y2|.

Little Sin really likes to party and is hoping to host a house party in Manhattan this Sunday. Little Sin has collected a list of people who will attend, and now needs to decide at whose home she will host the party.

Little Sin invited all of the people in several rectangular areas, and all of those people have said yes. A rectangular area is denoted as (x1, y1, x2, y2), where x1 ≤ x2, y1 ≤ y2. People who live in a rectangular area fill all integral points inside it. So there are a total of (x2 - x1 + 1) * (y2 - y1 + 1) people in the rectangular area (x1, y1, x2, y2).

Little Sin knows the coordinates of those rectangular areas. She wants the party to be hosted at the home of one of the people who is attending, but she also doesn't want everyone else to have to travel very far: she wants to minimize the sum of all distances from all attendees' houses to the party. Can you help her?

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each test case starts with a line containing a single integer: the number of rectangular areas, **B**. **B** lines follow. Each line contains 4 integers: x1, y1, x2, y2, denoting the coordinates of a rectangular area of people Little Sin has invited to her party.

Output

For each test case, output one line containing "Case #t: x y d", where t is the case number (starting from 1) and (x, y) is the coordinates of the person whose home the party should be hosted. If there are multiple positions with the same minimum total distance, choose the one with the smallest x. If there are still multiple positions, choose the one with the smallest y. The value d is the sum of the distances from all attendees' houses to the point (x, y).

Limits

Memory limit: 1GB.

$1 \le \mathbf{T} \le 10$.
$|x1|, |y1|, |x2|, |y2| \le 10^9$.
$x1 \le x2, y1 \le y2$.
The rectangular areas within a test case don't intersect.

Test set 1 - Visible

Time limit: 30 seconds.
$1 \le \mathbf{B} \le 100$.
$1 \le$ **Total number of people in each test case** ≤ 1000.

Test set 2 - Hidden

Time limit: 60 seconds.
$1 \le \mathbf{B} \le 1000$.
$1 \le$ **Total number of people in each test case** ≤ 1000000.

CamScanner

Sample

Sample Input

```
2
1
0 0 2 2
3
-1 2 -1 2
0 0 0 0
1 3 1 3
```

Sample Output

```
Case #1: 1 1 12
Case #2: -1 2 6
```

CamScanner

Sudoku Checker

Problem

Sudoku is a popular single player game. The objective is to fill a 9x9 matrix with digits so that each column, each row, and all 9 non-overlapping 3x3 sub-matrices contain all of the digits from 1 through 9. Each 9x9 matrix is partially completed at the start of game play and typically has a unique solution.

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

10

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

→ 45

Given a completed N^2xN^2 Sudoku matrix, your task is to determine whether it is a *valid* solution. A *valid* solution must satisfy the following criteria:

- Each row contains each number from **1** to N^2, once each.
- Each column contains each number from **1** to N^2, once each.
- Divide the N^2xN^2 matrix into N^2 non-overlapping NxN sub-matrices. Each sub-matrix contains each number from **1** to N^2, once each.

You don't need to worry about the uniqueness of the problem. Just check if the given matrix is a valid solution.

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each test case starts with an integer **N**. The next N^2 lines describe a completed Sudoku solution, with each line contains exactly N^2 integers. All input integers are positive and less than 1000.

Output

CamScanner

For each test case, output one line containing "Case #x: y", where x is the case number (starting from 1) and y is "Yes" (quotes for clarity only) if it is a valid solution, or "No" (quotes for clarity only) if it is invalid. Note that the judge is case-sensitive, so answers of "yes" and "no" will not be accepted.

Limits

Time limit: 30 seconds per test set.
Memory limit: 1GB.

$1 \leq \mathbf{T} \leq 100$.

Test set 1 - Visible

$\mathbf{N} = 3$.

Test set 2 - Hidden

$3 \leq \mathbf{N} \leq 6$.

Sample

Sample Input

```
3
3
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
3
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
3
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 999 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

Sample Output

```
Case #1: Yes
Case #2: No
Case #3: No
```

CamScanner