Pylons (8pts, 23pts)

Problem

Our Battlestarcraft Algorithmica ship is being chased through space by persistent robots called Pylons! We have just teleported to a new galaxy to try to shake them off of our tail, and we want to stay here for as long as possible so we can buy time to plan our next move... but we do not want to get caught!

This galaxy is a flat grid of **R** rows and **C** columns; the rows are numbered from 1 to **R**from top to bottom, and the columns are numbered from 1 to **C** from left to right. We can choose which cell to start in, and we must continue to jump between cells until we have visited each cell in the galaxy *exactly* once. That is, we can never revisit a cell, including our starting cell.

We do not want to make it too easy for the Pylons to guess where we will go next. Each time we jump from our current cell, we must choose a destination cell that does not share a row, column, or diagonal with that current cell. Let (i, j) denote the cell in the i-th row and j-th column; then a jump from a current cell (r, c) to a destination cell (r', c') is invalid if and only if any of these is true:

- r = r'
- c = c'
- r - c = r' - c'
- r + c = r' + c'

Can you help us find an order in which to visit each of the **R** × **C** cells, such that the move between any pair of consecutive cells in the sequence is valid? Or is it impossible for us to escape from the Pylons?

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each consists of one line containing two integers **R** and **C**: the numbers of rows and columns in this galaxy.

Output

For each test case, output one line containing `Case #x: y`, where `y` is a string of uppercase letters: either `POSSIBLE` or `IMPOSSIBLE`, according to whether it is possible to fulfill the conditions in the problem statement. Then, if it is possible, output **R** × **C**more lines. The i-th of these lines represents the i-th cell you will visit (counting starting from 1), and should contain two integers r_i and c_i: the row and column of that cell. Note that the first of these lines represents your chosen starting cell.

Limits

Time limit: 20 seconds per test set.
Memory limit: 1GB.

Test set 1 (Visible)

T = 16.
2 ≤ **R** ≤ 5.
2 ≤ **C** ≤ 5.

Test set 2 (Hidden)

1 ≤ **T** ≤ 100.
2 ≤ **R** ≤ 20.
2 ≤ **C** ≤ 20.

Sample

Input	Output
2 2 2 2 5	Case #1: IMPOSSIBLE Case #2: POSSIBLE 2 3 1 1 2 4 1 2 2 5 1 3 2 1 1 5 2 2 1 4

In Sample Case #1, no matter which starting cell we choose, we have nowhere to jump, since all of the remaining cells share a row, column, or diagonal with our starting cell.

In Sample Case #2, we have chosen the cell in row 2, column 3 as our starting cell. Notice that it is fine for our final cell to share a row, column, or diagonal with our starting cell. The following diagram shows the order in which the cells are visited:

2 4 6 10 8
7 9 1 3 5

Analysis

Test set 1

There are a few impossible cases for this problem. If we experiment with some small grids, we can find that in addition to the 2 x 2 grid given as a sample case, the 2 x 3, 2 x 4, and 3 x 3 cases have no solution. (Because of symmetry, the 3 x 2 and 4 x 2 cases are also impossible.)

In the 2 x 3 grid, we can notice that the middle cell of the top row shares a row, column, or diagonal with every other cell; the same is true of the central cell of the 3 x 3 grid. In each case, we cannot go from that cell to any other cells, or vice versa, so the grid is unsolvable.

In the 2 x 4 grid, we can try starting in the second cell of the top row; then we find that we are forced into a series of three moves that eventually take us to the third cell of the bottom row,

from which there are no other moves; we can never move out of that set of four cells that we visited. Since the same is true of the other set of four cells, there is no solution.

The other cases in test set 1 are all solvable, though. One strategy is to make mostly "knight moves" — two unit cells in one direction, and one unit cell in the other. We may not be able to solve a grid using only knight moves — see this article for more information — so we can sprinkle in other legal moves as well. For instance, we can solve the 3 x 4 case by visiting the cells in the following order:

```
02 05 10 07 09 12 01 04 06 03 08 11
```

And here is a solution for the 3 x 5 case, which makes many steps that are wider than knight moves:

```
04 14 09 12 07 06 11 01 15 03 02 08 13 10 05
```

Because of symmetry, the only remaining cases are 2 x 5, 4 x 4, and 4 x 5, and we can solve these by hand if we choose, or we can use a brute force algorithm.

Test set 2

There are multiple strategies for dealing with test set 2. One class of approach is *constructive*. For example, we can devise general solutions for 2 x N and 3 x N grids, for arbitrary N, and then divide up the grid into horizontal strips of height 2, plus one more strip of height 3 if needed. Unfortunately, this can be tricky to get right. We need to make sure that the solutions to the subproblems do not violate the rules — the last move in one subproblem cannot be in the same row, column, or diagonal as the first move in another. Moreover, we might struggle to come up with our general 2 x N and 3 x N solutions.

The Code Jam team's first successful constructive solution included multiple cases for 2 x N (depending on whether N is odd, 0 mod 4, or 2 mod 4), and multiple cases for 3 x N (which entailed adding pairs of columns to the left and right of our hardcoded 3 x 4 and 3 x 5 solutions, bouncing between those columns to avoid breaking rules). It also made each such solution start near the left edge of each strip and end near the right edge, to avoid diagonal interactions among subproblems / strips.

Is there an easier way? Let's take a step back. The problem imposes some constraints, but one can observe that it is not too difficult to solve the larger test set 1 cases by hand. This suggests that there are many possible solutions, and we might expect even more as the grid's dimensions get larger. (We could use Ore's theorem to establish the existence of at least one solution for sufficiently large grids.) So, our intuition might suggest at this point that the best approach is some kind of brute force.

We might consider using backtracking solutions. One possible concern is that these solutions, whether they are breadth-first or depth-first, proceed in an orderly fashion that might be more likely to leave us with tough "endgames" in which the constraints are impossible. For example, if our solution somehow solves all but the bottom row of the grid, then all hope for the universe is lost!

We can try these solutions anyway, or we can rely on our occasional friend, randomness! We can pick a random starting cell, repeatedly choose valid moves uniformly at random from the space of all allowed moves from our current cell, and, if we run out of available moves, give up and start over. For any case except for the impossible ones mentioned above, this approach finds a solution very quickly.

Many problems cannot be approached with random or brute-force solutions, but identifying the problems that can be (in Code Jam or the real world) is a useful skill!

There's even a greedy algorithm

We are aware of other approaches. For example, we know that there is at least one implementation of the following idea that solves every possible test in the test set 1 and 2 limits: repeatedly greedily select an unvisited cell that has the largest count of unvisited "neighbors", where we define a cell's neighbors as those cells that share a row, column, or diagonal with that cell. (We can either rule out specific impossible cases beforehand, since there are not very many, or infer impossibility by comparing the number of unvisited cells with the largest count of unvisited neighbors.)

Although we do not provide a proof here, intuitively, it is advantageous to prevent any one row, column, or diagonal from having too many unvisited cells, relative to other rows, columns, and diagonals. For example, if we are close to the end of our journey, and a majority of our remaining unvisited cells are in the same column, we are doomed.

Given the relatively small number of possible test cases for this problem, it is not too hard to check all of them to verify that a solution works, and that may be much easier than proving correctness! That is a rare luxury to have for a Code Jam problem!