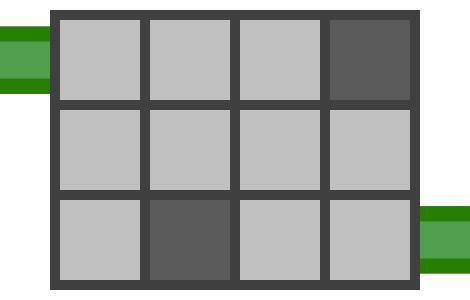
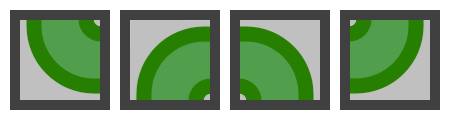
A. **Let It Flow**

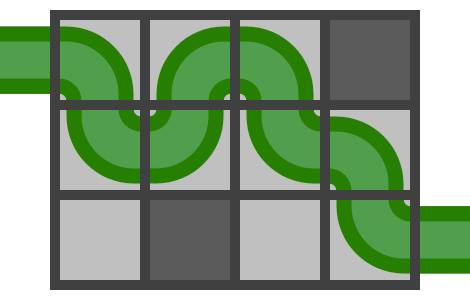
You've been hired for a boring plumbing installation job. You'll be installing pipes into a house which can be modeled as a grid with 3 rows and **N** columns. The *j*th cell in the *i*th row of the grid is described by the character **Gi,j**, and is either empty (if **Gi,j** = .) or is blocked by a wall (if **Gi,j** = #). There's already a pipe incoming into the left edge of the top-left cell, and another pipe leaving from the right edge of the bottom-right cell. For example, the house might initially look as follows:



Your job is to install one or more additional pipes in empty cells throughout the house, such that water can successfully flow through them from the top-left pipe all the way to the bottom-right one. You have access to a whole lot of pipes, but unfortunately they're all of a single type — elbow-shaped. When you install such a pipe in a cell, it allows water to flow in from one edge of the cell, make a 90-degree turn either clockwise or counter-clockwise, and flow out from another edge of the cell. Each pipe may be installed in any of the following four rotations:



Pipes may only be installed into empty cells, and no cell may contain multiple pipes. So as to not waste equipment, each pipe installed must end up actually contributing to the flow of water -- in other words, you may not install a pipe if it could be removed without disrupting the flow of water from the top-left pipe to the bottom-right one. For example, the following diagram illustrates the only valid set of pipes which could be installed into the house shown above:



To make the job less boring, you're interested in counting the number of different valid sets of pipes which you might choose to install. As this number may be large, you only want to compute its value modulo 1,000,000,007. Two sets of pipes are considered to be different if one of them includes a pipe in a cell which is left empty in the other, or if at least one pipe is installed in a different rotation between them.

**Input**

Input begins with an integer **T**, the number of houses. For each house, there is first a line containing the integer **N**. Then, 3 lines follow, each containing a string of length **N** containing only the characters . and #. The *j*th character of the *i*th line is **Gi,j**.

**Output**

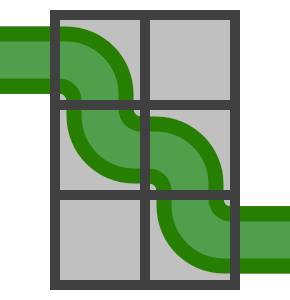
For the *i*th house, print a line containing "Case #*i*: " followed by the number of different valid sets of pipes which could be installed in the *i*th house (modulo 1,000,000,007).

**Constraints**

1 ≤ **T** ≤ 7 (T = 7 is only in the sample test, in other tests T = 1)  
1 ≤ **N** ≤ 1,000

**Explanation of Sample**

In the first case, pipes can be installed only as follows:



The third case is explained in the problem statement above.

Sample input

7

2

..

..

..

2

#.

..

.#

4

...#

....

.#..

4

..##

....

.#..

5

.#...

.....

.....

8

....#...

........

#.......

70

......................................................................

......................................................................

......................................................................

Sample output

Case #1: 1

Case #2: 0

Case #3: 1

Case #4: 0

Case #5: 0

Case #6: 4

Case #7: 179869065

B. **Ethan Traverses a Tree**

Ethan is doing his second programming assignment: implementing pre-order tree traversal.

Ethan has a binary tree with **N** nodes (numbered 1 to **N**), rooted at node 1. Each node *i*'s left child is node **Ai** (with **Ai** = 0 indicating no left child), and similarly its right child is **Bi** (with **Bi** = 0 indicating no right child). Each node *i* is also assigned an integral label **Li**.

Given such a tree, Ethan must compute its [pre-order traversal](https://en.wikipedia.org/wiki/Tree_traversal#Pre-order_(NLR)) (expressed as a sequence of node labels). The pre-order traversal of a tree involves taking its root node, then concatenating the pre-order traversal of the root's left sub-tree (if any), and then concatenating the pre-order traversal of the root's right sub-tree (if any).

Ethan has attempted to solve this problem, but unfortunately he got his computer science terms mixed up, and now his algorithm finds the tree's [post-order traversal](https://en.wikipedia.org/wiki/Tree_traversal#Post-order_(LRN)) instead! The post-order traversal of a tree involves taking the post-order traversal of the root's left sub-tree (if any), and then concatenating the post-order traversal of the root's right sub-tree (if any), and finally concatenating the root node at the end.

Since you were mean to Ethan on his first assignment, you'd like to cheer him up by making his algorithm work out after all. Though the tree's shape must stay as is, you can choose a set of labels **L1..N** for its nodes such that Ethan's algorithm will still produce the correct answer — in other words, such that the sequence of node labels in the tree's pre-order traversal is equal to the sequence of node labels in its post-order traversal. Your only two restrictions are that each node label must be between 1 and **K** (inclusive), and that every integer between 1 and **K** (inclusive) must be used as the label of at least one node. You'd like to find any way of validly labelling the nodes, or determine that no way exists.

**Input**

Input begins with an integer **T**, the number of trees. For each tree, there is first a line containing the space-separated integers **N** and **K**. Then, **N** lines follow. The *i*th of these lines contains the space-separated integers **Ai** and **Bi**.

**Output**

For the *i*th tree, print a line containing "Case #*i*: " followed by your chosen node labels **L1..N** separated by spaces, or "Impossible" if there's no valid way to label the nodes.

**Constraints**

1 ≤ **T** ≤ 6 (T = 6 is only in the sample test, in other tests T = 1)  
1 ≤ **K** ≤ **N** ≤ 2,000   
0 ≤ **Ai**, **Bi** ≤ **N**

Every tree is guaranteed to be a valid binary tree rooted at node 1.

**Explanation of Sample**

In the first case, if **L** = [1, 1], then both the pre-order and post-order label sequences will be [1, 1].

In the second case, for each label between 1 and **K** to be present, you must choose either **L** = [1, 2] or **L** = [2, 1], both of which would result in the pre-order and post-order label sequences differing. For example, if **L** = [1, 2], then the pre-order sequence will be [1, 2] while the post-order sequence will be [2, 1].

In the third case, if **L** = [2, 2, 1], then the pre-order and post-order label sequences will both be [2, 1, 2].

Note that other outputs for example cases 3 to 5 would also be accepted.

Sample input

6

2 1

0 2

0 0

2 2

0 2

0 0

3 2

0 3

0 0

2 0

9 4

2 3

4 0

0 5

6 0

0 7

8 0

0 9

0 0

0 0

15 4

8 0

0 0

0 9

0 0

15 6

0 0

4 0

2 13

0 0

14 12

5 0

3 0

10 11

0 0

0 7

15 3

8 11

0 0

0 9

0 0

15 6

0 0

4 0

2 13

0 0

14 12

5 0

3 0

10 0

0 0

0 7

Sample output

Case #1: 1 1

Case #2: Impossible

Case #3: 2 2 1

Case #4: 2 4 3 1 3 4 3 2 2

Case #5: 1 1 2 3 4 1 2 3 1 3 1 3 2 3 1

Case #6: Impossible

C. **Platform Parkour**

You're about to put on an exciting show at your local circus — a parkour demonstration! **N** platforms with adjustable heights have been set up in a row, and are numbered from 1 to **N** in order from left to right. The initial height of platform *i* is **Hi** metres.

When the show starts, **M** parkourists will take the stage. The *i*th parkourist will start at platform **Ai**, with the goal of reaching a different platform **Bi**. If **Bi** > **Ai**, they'll repeatedly jump to the next platform to their right until they reach **Bi**. If **Bi** < **Ai**, they'll instead repeatedly jump to the next platform to their left until they reach **Bi**. All of the parkourists will complete their routes simultaneously (but don't worry, they've been trained well to not impede one another).

Not all parkourists are equally talented, and there are limits on how far up or down they can jump between successive platforms. The *i*th parkourist's maximum upwards and downwards jump heights are **Ui** and **Di**, respectively. This means that they're only able to move directly from platform *x* to some adjacent platform *y* if **Hx** - **Di** ≤ **Hy** ≤ **Hx** + **Ui**, where **Hx** and **Hy** are the current heights of platforms *x* and *y*, respectively.

With the show about to begin, a disastrous flaw has just occurred to you — it may not be possible for all of the parkourists to actually complete their routes with the existing arrangement of platforms! If so, you will need to quickly adjust some of the platforms' heights first. The height of each platform may be adjusted upwards or downwards at a rate of 1 metre per second, to any non-negative real-valued height of your choice, and multiple platforms may be adjusted simultaneously. As such, if the initial height of platform *i* is **Hi** and its final height is **Pi**, then the total time required to make your chosen height adjustments will be max{|**Hi** - **Pi**|} over *i*=1..**N**.

Determine the minimum amount of time required to set up the platforms such that all **M** parkourists will then be able to complete their required routes. Note that you may not perform further height adjustments once the show starts. The platform heights must all remain constant while all **M** parkourists complete their routes.

In order to reduce the size of the input data, you're given **H1** and **H2**. **H3..N** may then be generated as follows using given constants **W**, **X**, **Y**, and **Z** (please watch out for integer overflow during this process):

**Hi** = (**W** \* **Hi-2** + **X** \* **Hi-1** + **Y**) % **Z** (for *i*=3..**N**)

**Input**

Input begins with an integer **T**, the number of shows. For each show, there is first a line containing the space-separated integers **N** and **M**. The next line contains the space-separated integers **H1**, **H2**, **W**, **X**, **Y**, and **Z**. Then, **M** lines follow. The *i*th of these lines contains the space-separated integers **Ai**, **Bi**, **Ui**, and **Di**.

**Output**

For the *i*th show, print a line containing "Case #*i*: " followed by 1 real number, the minimum amount of time required to set up the platforms (in seconds). Absolute and relative errors of up to 10-6 will be ignored.

**Constraints**

1 ≤ **T** ≤ 5 (T = 5 is only in the sample test, in other tests T = 1)  
2 ≤ **N** ≤ 200,000   
1 ≤ **M** ≤ 20   
0 ≤ **Hi** < **Z**   
0 ≤ **W**, **X**, **Y** < **Z**   
1 ≤ **Z** ≤ 1,000,000   
1 ≤ **Ai**, **Bi** ≤ **N**   
0 ≤ **Ui**, **Di** ≤ 1,000,000   
**Ai**, ≠ **Bi**

**Explanation of Sample**

In the first case, **H** = [0, 10]. You can increase the first platform's height by 3.5 and decrease the second's by 3.5 in 3.5 seconds, yielding **P** = [3.5, 6.5]. The single parkourist will then be able to successfully complete their route from platform 1 to platform 2 by jumping upwards by a height of at most 3.

In the second case, **H** = [50, 59, 55, 51, 47]. One optimal possibility is **P** = [54.0, 54.5, 53.5, 52.5, 51.5].

In the third case, **H** = [46, 38, 38, 22, 8].

In the fourth case, **H** = [53, 25, 24, 81, 77, 40, 29, 21].

Sample input

5

2 1

0 10 0 0 0 11

1 2 3 8

5 1

50 59 0 1 96 100

1 5 2 1

5 2

46 38 2 4 44 50

4 1 3 2

3 5 4 1

8 5

53 25 15 23 54 100

1 8 14 9

3 1 5 7

6 8 2 1

5 4 1 8

8 1 11 10

100000 5

72464 815932 291056 735002 4758 972844

68327 29055 2880 3051

98105 26231 3531 3141

4018 31397 2797 3619

92594 65725 3824 3003

81932 8087 3372 3158

Sample output

Case #1: 3.500000

Case #2: 4.500000

Case #3: 15.000000

Case #4: 24.000000

Case #5: 483009.500000

D. **Evening of the Living Dead**

A quiet evening has set over a residential area. As families sit down for supper in the safety of their homes, a calm atmosphere permeates the outside air. The neighborhood feels truly at peace, separated from the frenzy of the rest of the world. Also, a bunch of zombies have just risen out of the ground and want to eat everybody.

The neighborhood has **N** yards in a row, numbered from 1 to **N**. There are also **N**-1 fences, one between each pair of adjacent yards. The fence between yards *i* and *i*+1 has an unknown integral height drawn uniformly at random from the inclusive interval [**Ai**, **Bi**]. In other words, the *i*th fence has **Bi** - **Ai** + 1 possible heights, each of which is equally likely.

**M** hungry zombies are also present, with the *i*th of them initially in yard **Yi**. Fortunately for the zombies, they might not be stopped by the surrounding fences so easily. The *i*th zombie has the ability to climb over any fence with a height of at most **Hi**. It may repeatedly move from its current yard to an adjacent one, as long as the fence between the yards is no taller than **Hi**. Multiple zombies may start in the same yard, and multiple zombies may occupy the same yard at any point.

A yard is considered "safe" if it's impossible for any zombies to ever reach it. Determine the probability that at least one of the **N** yards is safe. Let this probability be represented as a quotient of integers *p*/*q* in lowest terms. Output the value of this quotient modulo 1,000,000,007 — in other words, output the unique integer *x* such that 0 ≤ *x* < 1,000,000,007 and *p* = *x*\**q* (modulo 1,000,000,007).

**Input**

Input begins with an integer **T**, the number of neighborhoods. For each neighborhood, there is first a line containing the space-separated integers **N** and **M**. Then, **N-1** lines follow. The *i*th of these lines contains the space-separated integers **Ai** and **Bi**. Then, **M** lines follow. The *i*th of these lines contains the space-separated integers **Yi** and **Hi**.

**Output**

For the *i*th neighborhood, print a line containing "Case #*i*: " followed by 1 integer, the probability that at least one of the yards is safe, expressed as a quotient of integers modulo 1,000,000,007.

**Constraints**

1 ≤ **T** ≤ 6 (T = 6 is only in the sample test, in other tests T = 1)  
1 ≤ **N** ≤ 3,000   
1 ≤ **M** ≤ 3,000   
1 ≤ **Ai** ≤ **Bi** ≤ 1,000,000   
1 ≤ **Yi** ≤ **N**   
1 ≤ **Hi** ≤ 1,000,000

**Explanation of Sample**

In the first case, if the height of the single fence is 100, then the zombie in yard 1 will be able to climb over it to reach yard 2, meaning that no yards will be safe. Otherwise, if the fence's height is 101, then yard 2 will be safe. Therefore, the probability that at least one of the yards is safe is 1/2 = 500000004 (modulo 1,000,000,007).

In the second case, in order for yard 2 to be safe from both surrounding zombies, the first fence's height must be either 3 or 4, and the second fence's height must be 4. The probability of this occurring is 2/4 \* 1/4 = 1/8 = 125000001 (modulo 1,000,000,007).

In the third case, the probability of at least one yard being safe is 2/3 = 666666672 (modulo 1,000,000,007).

Sample input

6

2 1

100 101

1 100

3 2

1 4

1 4

1 2

3 3

4 2

1 4

1 2

3 5

4 3

3 1

6 3

1 20

2 19

3 18

4 17

5 16

6 9

1 13

6 7

6 3

1 20

2 19

3 18

4 17

5 16

4 9

3 13

4 7

16 10

10 40

5 55

20 25

25 50

40 65

35 50

15 20

10 35

15 55

10 70

25 60

5 50

20 50

10 30

15 55

9 32

14 48

5 34

13 41

7 35

2 19

13 20

5 17

3 33

11 24

Sample output

Case #1: 500000004

Case #2: 125000001

Case #3: 666666672

Case #4: 417224706

Case #5: 441220242

Case #6: 292643605