Contents


TESTUDO
HSPC 2011

1 Base K

We are now used to the decimal positional notation for representing numbers, the modern version of which only dates back to the 9^{th} century. In this notation we use 10 digits, from 0 to 9, to represent a number. However there is nothing special about 10, and we can just as easily use any other base to represent numbers. For example, in base-2 (binary) notation, we only use 2 digits, 0 and 1, whereas in base-16 (hexadecimal) notation, we use 15 digits: 0-9, A, B, C, D, E, F (the last 6 representing the digits ten, eleven, twelve, thirteen, fourteen, and fifteen); both these notations are commonly used by computers. In fact, several other bases are commonly seen in history: base-60 (used by ancient Babylonians), base-13 (used by Mayans), base-20 (Aztecs and Mayans, and possibly old Europe), base-27 (Telefol language and the Oksapmin language), and so on.

More generally, a base-k representation uses k digits: 0 to $k-1$. Given a number n represented in base-k as $a_3a_2a_1a_0$, where a_3, a_2, a_1, a_0 are digits, we have that:

$$n = (a_3a_2a_1a_0)_k = a_3 \times k^3 + a_2 \times k^2 + a_1 \times k^1 + a_0 \times k^0$$

Here we use the subscript k to denote that the number is being written in base-k.

For example,
$465_7 = 4 \times 7^2 + 6 \times 7^1 + 5 \times 7^0 = 243_{10}$
$46E_{16} = 4 \times 16^2 + 6 \times 16^1 + 14 \times 16^0 = 1134_{10}$ (recall E represents the number fourteen in base-16)

Our use of decimal notation is likely based on it being easier to count up to 10 using fingers. However, programs have no such restriction. When Sam arrives on the Grid world, he discovers that the programs use arbitrary bases to communicate numbers to each other. Sam, not really being a program, has serious trouble with this, and practices arithmetic in arbitrary bases as much as he can. For example, when he sees any two numbers written together, call them n and k, he times how long it takes him to decide whether base-k representation of n contains all digits from 0 to $k-1$.

For example, base-3 representation of fifteen (15_{10}) is 120_3:

$$120_3 = 1 \times 3^2 + 2 \times 3^1 + 0 \times 3^0 = 9 + 6 + 0 = 15_{10}$$

which contains all 3 digits. In comparison, base-4 representation of eighteen (18_{10}) is 102_4:

$$102_4 = 1 \times 4^2 + 0 \times 4^1 + 2 \times 4^0 = 16 + 0 + 2 = 18_{10}$$

does not contain all 4 digits (it only contains 0, 1, and 2).

You are to write a program to help Sam verify his answers, i.e., given two numbers n and k, you are to decide if base-k representation of n contains all digits from 0 to $k-1$. Note that the digit 0 must appear somewhere in the middle of the number; so the number 123456789 in base-10 representation does NOT contain all digits (it only contains 1-9, but no 0, even though we can also write the number as 0123456789).

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (< 100). After that, each line contains one test case, i.e., two numbers n and k, in that order. Assume that $2 \leq k \leq 30$, and $0 \leq n < 2^{30}$.

Output: For each test case, you are to output the base-k representation of n and whether it contains all digits or not. The exact form of the output is shown below.

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static void solveBaseK(int n, int k)
```

The body of the procedure should set the two variables `basek_representation` and `containsAllDigits` appropriately.

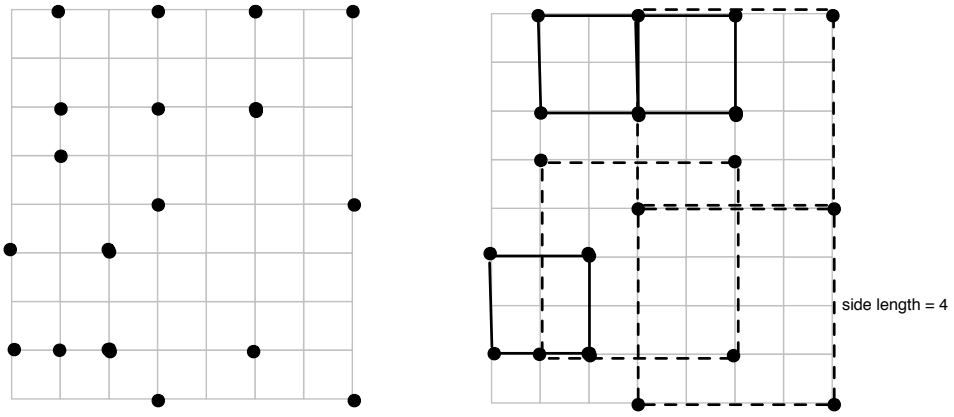
Examples:

The output is shown with an extra blank line so that each test case input is aligned with the output; that blank line should not be present in the actual output.

Input:	Output:
6	
100 10	Base-10 representation of 100 is 100; does not contain all digits.
343 7	Base-7 representation of 343 is 1000; does not contain all digits.
240 5	Base-5 representation of 240 is 1430; does not contain all digits.
1234567890 10	Base-10 representation of 1234567890 is 1234567890; contains all digits.
1 2	Base-2 representation of 1 is 1; does not contain all digits.
2 2	Base-2 representation of 2 is 10; contains all digits.

2 Most Frequent Square

As Sam is flying over the Grid world to the outlands, he notices some interesting formations on the ground, where there is well-defined small square grid with only a subset of its grid points lit (see figure). To pass time, Sam starts counting the number of different axis-parallel squares formed by the grid points. You are to write a program to help Sam with this task.



Specifically, given a set of grid coordinates, your goal is to find the most common size of an axis-parallel square formed by subsets of the given points. An axis-parallel square is such that the square is aligned with the grid axes (i.e., the sides of the square are either vertical or horizontal, but are not allowed to at an angle). For instance, in the figure above, there are 3 such squares with side length 2, and three squares with side length 4.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (≤ 50). After that, each line contains one test case. The test case begins with the number of points, k (≤ 30), and then we have k coordinate pairs (for a total of $2k$ numbers). Assume all numbers are ≥ 0, and that the coordinates are all ≤ 30.

Output: For each test case, if there is at least one axis-parallel square present among the provided points, you should output the side length for which there are maximum squares present, and also the number of squares for that side length. If there are more than two side lengths with maximum number of squares, then output the largest of the side lengths. If the provided set of points does not form a single square, then you should output: `No squares among the points.`

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static void solveSquares(int [] x_coordinates, int [] y_coordinates)
```

Your function should set the values of two global variables appropriately:

```
int max_occurring_length
int number_of_squares_for_max_occurring_length
```

The second variable (`number_of_squares_for_max_occurring_length`) should be set to 0 (default) if there are no squares.

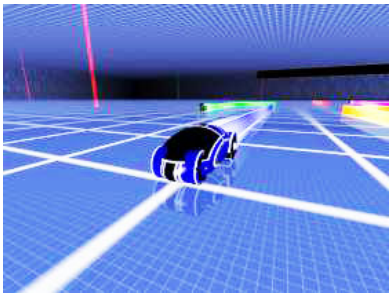
Examples:

The output is shown with an extra blank line so that each test case input is aligned with the output; that blank line should not be present in the actual output.

Input:	Output:
8	
4 0 0 1 1 0 1 1 0	LENGTH = 1, COUNT = 1
4 0 0 2 1 0 1 1 0	No squares among the points.
4 0 0 2 2 0 2 2 0	LENGTH = 2, COUNT = 1
9 0 0 0 1 0 2 1 0 1 1 1 2 2 0 2 1 2 2	LENGTH = 1, COUNT = 4
9 0 0 0 2 0 4 2 0 2 2 2 4 4 0 4 2 4 4	LENGTH = 2, COUNT = 4
10 2 3 0 2 2 1 2 0 1 1 1 2 3 3 3 0 0 1 0 0	LENGTH = 1, COUNT = 1
10 2 0 3 2 0 1 2 1 1 0 3 3 1 3 2 3 2 2 3 0	LENGTH = 1, COUNT = 1
10 3 1 2 2 3 3 0 2 0 0 2 0 0 1 1 1 1 3 1 2	LENGTH = 2, COUNT = 2

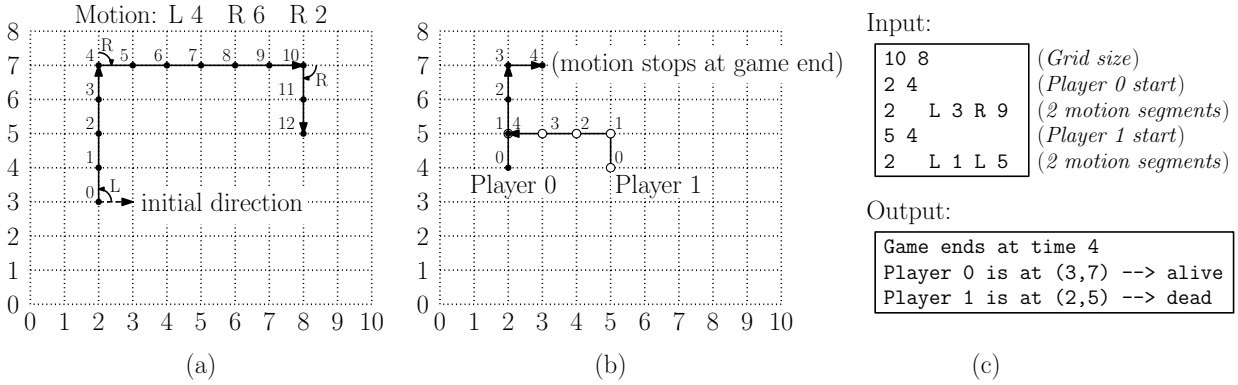
3 Light-Cycle Race

Light-cycle racing involves two motorcycles that drive along the edges of a two-dimensional square grid. As each light cycle traverses its path in the grid, it leaves behind a wall along the path it travels. Two players compete by driving simultaneously from two different starting points. A player may die in three possible ways: (1) driving outside the grid, (2) hitting a wall (either his own or his opponent's), or (3) colliding with the other light cycle. Your task is to input a description of the paths of two players and determine when the game ends, the positions of the players when the game ends, and whether each player is alive or dead at the end of the race.



A player's path description is defined as follows. First, the (x, y) coordinates of the player are given. Initially, the player faces to the east, that is, in the same direction as the positive x-axis. The path is described as a number of *segments*, where each segment is of the form (T, k). T indicates the direction turned, which is either 'S', 'L', or 'R', meaning *Straight*, *Left*, or *Right*, respectively. The integer k is the distance to travel after turning. Thus "R 6" means to turn right, relative to the direction you are currently moving and travel 6 units.

As an example, consider a grid of width 10 and height 8, and a player that starts at $(2, 3)$ and executes the three segments "L 4", "R 6", and "R 2". The result is shown in Figure 1(a). The cycle moves at one grid square per second, and the labels on the points indicate the time that the cycle arrives at this point.



Input:

```
10 8
2 4
2   L 3 R 9
5 4
2   L 1 L 5
```

(*Grid size*)
(*Player 0 start*)
(*2 motion segments*)
(*Player 1 start*)
(*2 motion segments*)

Output:

```
Game ends at time 4
Player 0 is at (3,7) --> alive
Player 1 is at (2,5) --> dead
```

Figure 1: Example of a light cycle path.

The program inputs the dimensions of the driving area, descriptions of the two players' motions. Each description contains the player's initial (x, y) coordinates, the number of motion segments, followed by the list of segments. Your program should simulate the game, starting at time 0, and continues until either:

- the first time that either player dies (both may die at the same time), or
- neither has died, but one player completes its motion.

When the game stops, the program outputs the finish time, the player positions, and the final player states (alive or dead). An example is shown in Figure 1(b). There are two players (0 and 1). The game

ends at time 4 when player 1 hits the wall of player 0. Any motion segments that are given after the end of the game are simply ignored.

Input/Output Format:

We have provided a program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static void playGame()
```

Your procedure gets its input data from the following global variables:

Name	Type	Description
`width`	`int`	Width of playing grid
`height`	`int`	Height of playing grid
`xStart[p]`	`int[]`	Initial x-coordinate of player p ($p = 0, 1$)
`yStart[p]`	`int[]`	Initial y-coordinate of player p ($p = 0, 1$)
`nSeg[p]`	`int[]`	Number of motion segments for player p
`turn[p][i]`	`char[][]`	i-th turn ('L', 'R', or 'S') for player p
`dist[p][i]`	`int[][]`	i-th distance for player p

Your procedure writes the results to the following global variables:

Name	Type	Description
`endTime`	`int`	time at which game ends
`xFinal[p]`	`int[]`	x-coordinate of player p at end
`yFinal[p]`	`int[]`	y-coordinate of player p at end
`isAlive[p]`	`boolean[]`	true if player p is alive at end

Example:

Sample input is shown to be easily readable. The actual input files have fewer white spaces and newlines.

Input:	Output:

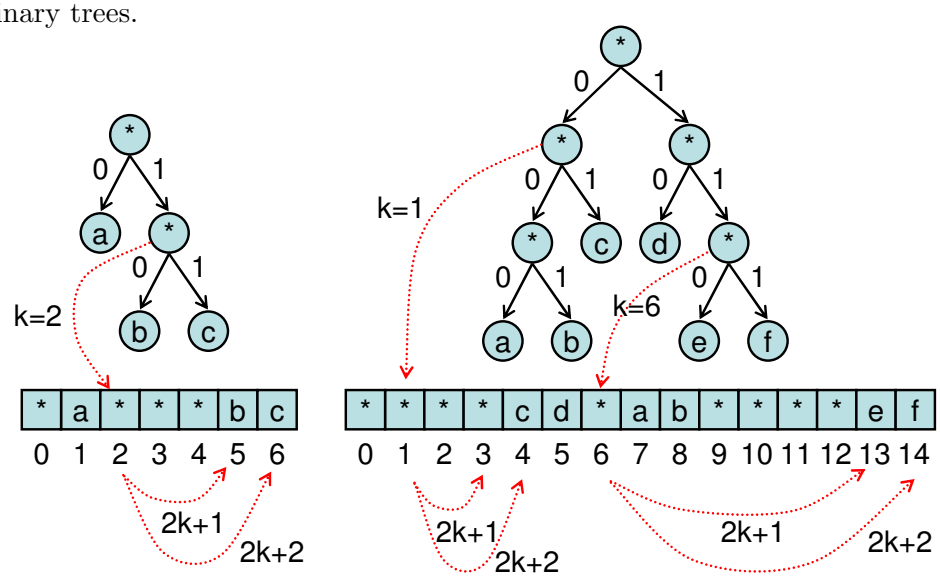
`1`
`10 8`
`2 4`
`2  L 3  R 9`
`5 4`
`2  L 1  L 5`
	`----------------------------------------`
	`Width = 10 Height = 8`
	`Player 0:`
	`  Start: (2, 4)`
	`  Segs:  L 3 R 9`
	`Player 1:`
	`  Start: (5, 4)`
	`  Segs:  L 1 L 5`
	`Game ends at time 4`
	`Player 0 final position: (3, 7) --> alive`
	`Player 1 final position: (2, 5) --> dead`

4 Prefix Codes

Sam has found an ancient communications channel to the outside world. However, the channel is really slow, so Sam and his friends would like to use a compression code to send messages to each other more quickly. After much thought, Sam decides to use a binary *prefix code*.

A prefix code is a variable-length code system where no valid code in the system is a prefix of any other code. For instance, {a=0, b=10, c=11} is a prefix code, but {a=0, b=10, c=01} is not, since 0 is a prefix of 01.

A binary prefix code (codes consisting of 0's and 1's) can be represented as a binary tree, where symbols are leaves and the path from the root to a leaf provides its code. For simplicity we assume the path to the left child contains a 0, and the path to the right child contains a 1. For instance, the binary prefix codes {a=0, b=10, c=11} and {a=000, b=001, c=01, d=10, e=110, f=111} may be represented using the following binary trees.



Furthermore, a binary tree may be represented compactly using a string. To see how this works, view a string of length N as a sequence of characters at positions 0 through N-1. Then assume the root node of the tree is at position 0, and the left and right child of a node at position k are at positions 2k+1 and 2k+2, respectively. For instance, the binary trees used earlier may be represented by the strings *a***bc and ****cd*ab****ef, where * is used to represent an interior (non-leaf) or missing node.

Using binary prefix codes, Sam and his friends can compress their messages into binary form for fast transmission. Compressed messages can be decoded easily using the binary tree, starting at the root node and going to either the left or right child, depending on whether the next value is a 0 or 1. If a leaf node representing a character is reached, the character is output and the remainder of the message can be decoded by restarting at the root node.

Your task is to help Sam decode strings encoded as binary numbers back to their original state, using binary prefix codes provided in string format.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (< 100). After that, each line contains one test case. The test case begins with k, the number of strings to be decoded, the string representation of the prefix code, followed by the k strings to be decoded. You may assume all strings to be decoded are composed of characters found in the prefix code.

Output: For each test case, you are to output a line containing each decoded string, separated by spaces.

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

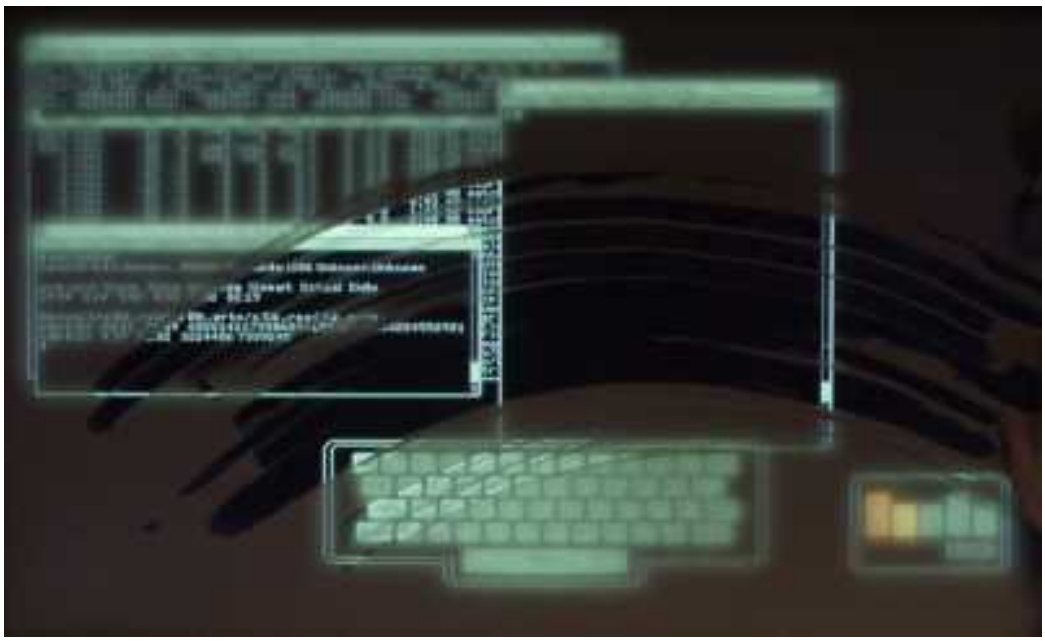
```
private static String decode(String prefixCode, String code)
```

which should return the decoding of the input argument "code" using the code described in input argument "prefixCode".

Examples:

The output is shown with an extra blank line so that each test case input is aligned with the output; that blank line should not be present in the actual output.

Input:	Output:
4	
3 *a***bc 0 10 11	a b c
2 *a***bc 11010 010100	cab abba
6 ****cd*ab****ef 000 001 01 10 110 111	a b c d e f
5 ****cd*ab****ef 11100001110 000 00100010 01000111110 001110110	face a bad cafe bee



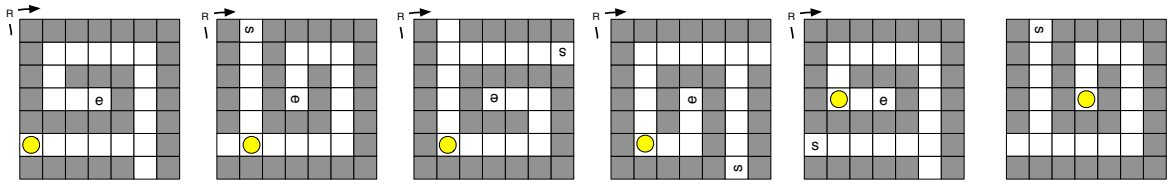
5 Rotation Maze

Kevin Flynn (Sam's father) likes to design innovative and fun video games for his video arcade. He is currently thinking about a maze game with a twist (literally) as discussed below. Since not all mazes are solvable, you are to help him decide whether a particular maze is solvable or not.

In the game, you are provided a maze, specified by a 2D array of cells. Each cell can be either empty, a wall, the starting point, or the ending point. A ball starts at the cell marked as the starting point and your goal is to get the ball to the end point. Gravity is acting in the maze and the ball will fall down the screen unless a wall is supporting it from below.

The operations you can perform are to either (a) Rotate the maze 90° to the left (counter-clockwise), or (b) Rotate the maze 90° to the right (clockwise). A solution is a string of of "`L`" and "`R`", indicating a series of left and right rotations that will cause the ball to come to rest at the designated ending point. If no solution exists, return the string "`NONE`".

For example, consider the input maze at the far left:



The solution here is "`RRRRR`" – i.e., a sequence of 5 clockwise rotations will bring the ball to rest at the ending point.

Here are some more specific details about the game.

1. The ball can pass through the starting and ending point. It needs to come to rest on the ending point for it to be a valid solution. For example, if the ending point was the empty square adjacent to the current ending point in the above example, there is no solution since the ball will always pass through that ending point.
2. The start and end points do not need to be at the edge of the board.
3. The maze does not have to be completely enclosed. It is possible for the ball to fall off the board.
4. If there is more than one sequence of rotations, return the shortest one. If there are several shortest sequences, return the one that would come alphabetically first if viewed as a word.
5. The board starts in the orientation as it is shown in the file (with gravity acting down the screen). The ball should be allowed to fall from its starting place (if it can) before any rotations occur.

Input/Output Format:

Input: The first line of test data file contains the number of test cases (≤ 20). Then the input cases are listed one after the other. Each input case looks like this:

```
nx ny
XXXXXXXXXXX
XsX...X...X
X.X.X.X.X.X
X.X.X.X.X.X
X...X...XeX
XXXXXXXXXXX
```

where `nx` and `ny` are integers giving the number of columns and rows in the maze, respectively. Following these numbers are `ny` lines, each of length `nx`. Each character in these lines represents a cell of the maze. A "." character represents an empty part of the maze (over which the ball can roll). The "`s`" and "`e`" characters represent the starting and goal (end) positions of the ball, respectively. Leading and trailing whitespace is removed. Any other character represents an obstacle (a wall) that the ball cannot pass through.

Output: Return a string that lists out the sequence of turns to achieve the goal. If there is more than one sequence of rotations, return the shortest one. If there are several shortest sequences, return the one that would come alphabetically first if viewed as a word. If there is no way to solve the maze, then return "NONE".

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
public static String solveRotationMaze(BlockType [][] Maze,
                                int nx, int ny, int startx, int starty)
```

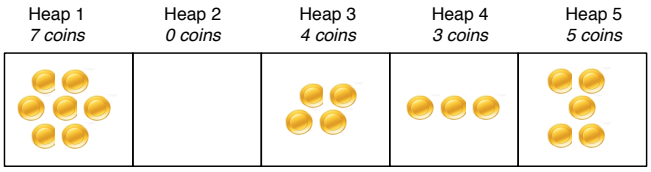
Example:

Input:	Output:

`3`
`4 4`
`xxxx`
`xs.x`
`x.ex`
`xxxx`
`13 6`
`XXXXXXXXXXXXX`
`X...X...X...X`
`X.X.X.X.X.X.X`
`X.X.X.X.X.X.X`
`XsX...X...XeX`
`XXXXXXXXXXXXX`
`13 6`
`XXXXXXXXXXXXX`
`X...X...X....`
`X.X.X.X.X.X.X`
`X.X.X.X.X.X.X`
`XsX...X...XeX`
`XXXXXXXXXXXXX`
	`Case 1: R`
	`Case 2: LLLLRRLLRRLL`
	`Case 3: NONE`

6 Grid Nim

While hiding out in the Outlands, Sam and Quorra are getting bored and start playing a game called *grid nim* which is a complex variation on the classic game of *nim*. The game board consists of n heaps arranged in a sequence, each containing a number of identical coins (see figure). The two players move alternately. When a player moves, he or she chooses a heap from either the left end or the right end of the sequence, and removes it from the sequence. The game is made more complex by the restriction that a player cannot select three consecutive heaps in three consecutive turns (this restriction does not apply when there is only one heap left at the end of the game). The game is over when the board is exhausted. The first player to play wins if the total number of coins he or she has selected is at least as much as the total number of coins collected by the second player. Otherwise the second player wins.



Here is one possible way the game may proceed in the example shown in the figure, assuming Sam moves first.

- Sam takes heap 1 from left end (7 coins)
- Quorra takes heap 5 from right end (5 coins)
- Sam takes heap 4 from right end (3 coins)
- Quorra takes heap 3 from right end (4 coins)
- Sam takes heap 2 (0 coins)

At the end, Sam has $7 + 3 = 10$ coins which is more than Quorra, and hence Sam wins this game. In fact, Sam can win no matter what Quorra does – if she chooses the heap 2 with 0 coins, then Sam can take the heap with 5 coins, and end up with a higher total score of 15 or 16 coins.

The restriction mentioned above does not come up in this example. However, consider an example where there are a large number of heaps, and consider an initial sequence of moves:

```
Sam - Left, Quorra - Right, Sam - Left, Quorra - Right
```

In the next turn, Sam cannot choose from the left end again, and must take the heap from the right end. Now the sequence of moves is:

```
Sam - Left, Quorra - Right, Sam - Left, Quorra - Right, Sam - Right
```

Now, although Quorra chose from the Right end last two times, she can choose either from the Right or the Left end. Since Sam just chose from the Right end, Quorra will not be taking a 3rd consecutive heap and hence she can choose from the Right end.

Quorra is very good at playing this game and always makes the best possible move. To account for this, she allows Sam to play first. Your goal is to help Sam win. Specifically, given an input board, your goal is to find out whether Sam (playing first) has a winning strategy assuming Quorra always makes the best possible move at every turn.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (≤ 50). After that, each line contains one test case. The test case begins with the number of elements in the sequence, k, and then we have k numbers which specify the numbers of coins in the heaps in the sequence. Assume all numbers are ≥ 0, and $< 2^{30}$.

Output: For each test case, you are to output "YES" (if Sam, playing first, has a winning strategy), or "NO" (if he does not).

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static boolean solveGridNim(int[] seq)
```

The procedure should return *true* if the first player has a winning strategy, and *false* otherwise.

Example:

The output is shown with an extra blank line so that each test case input is aligned with the output; that blank line should not be present in the actual output.

Input:	Output:
4	
3 1 10 1	NO
4 5 7 2 1	YES
7 0 0 8 4 2 5 2	NO
5 7 0 4 3 5	YES

7 Buddy, Can You Spare a Tronk?

The principal unit of currency in the world of Tron is the *Tronk*. Making change for the Tronk in this virtual world is challenging. In the USA, we have half-dollars, quarters, dimes, nickels, and pennies, worth respectively $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{10}$, $\frac{1}{20}$, and $\frac{1}{100}$ of a US dollar. In the world of Tron, they have *infinitely* many coins, one for the reciprocal of *every* integer:

$$1,\ \frac{1}{2},\ \frac{1}{3},\ \frac{1}{4},\ \frac{1}{5},\ldots$$

In particular, there is no smallest coin. Sam meets his Clu, his father's alter ego, who offers him the following puzzle:

> *If you could use any combination of n coins, how many different ways are there of making change for one Tronk?*

For example, if you could use $n = 3$ coins, there are exactly three ways to make change for one Tronk:

$$\frac{1}{3}+\frac{1}{3}+\frac{1}{3} \qquad \frac{1}{6}+\frac{1}{3}+\frac{1}{2} \qquad \text{and} \qquad \frac{1}{2}+\frac{1}{4}+\frac{1}{4}.$$

Clu adds two more constraints:

- You are given an integer r. No coin may be used more then r times. (For example, if $r = 2$, then the first solution above would not be valid, but the other two would be.) If $r = 0$, then each coin may be used an arbitrary number of times.
- You are given a list of integers $F = \{f_1, f_2, \ldots, f_k\}$. None of the reciprocals on this list may be used. For example, if $F = \{4, 6\}$, then $\frac{1}{4}$ and $\frac{1}{6}$ cannot be used, and so only the first solution above is valid.

Sam wonders, "But, how do I know what the smallest coin is that I will need to use?" Clu answers, "Sorry, you gotta figure that out on your own." To help Sam, design a program to solve this problem. The program's input consists of n, r, and the list F (the input format is given below). The program should output a list of all the valid solutions, followed by the total number of solutions.

Clu tells Sam that he also wants the output to be nicely sorted. Each line should hold the reciprocals of the coins that sum to 1, listed in *increasing order*. For example, the sequence $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ must be output as `2 3 6`. Also, each of the sequences of coins must be sorted in *lexicographically increasing order*. Thus, the above solution would be output as

```
2 3 6
2 4 4
3 3 3
3 solutions found
```

Clu also warns Sam that each solution he produces must be a correct solution. He explains to Sam that because of floating point errors and extremely small coins involved (e.g., coins like 1/10000000), a set of coins might sum up to arbitrarily close to 1, but if the values do not sum exactly to 1, the answer is incorrect.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (≤ 20). After that, each line contains one test case. The first number is the number of coins that you must use n (≤ 10). The second number is the number of repetitions allowed for each coin (r). If $r = 0$, that means any number of repetitions are allowed. The next number is the number of forbidden coins k (≤ 20), and the following k numbers denote the forbidden coins.

Output: For each test case, you are to find all different solutions, sorted as described above. We may give full credit if you miss a few solutions, but if your program returns a wrong solution (i.e., a solution where the set of coins does not add up to exactly 1), then you will not get any credit. You can also assume that you will not need to use any coin smaller than the reciprocal of 10,000,000.

Note: We have provided a skeleton program that handles the input and output. Specifically, there is a function that should be called to print out a single solution.

```
private static void printOneOutput(int[] coins)
```

You are to write the following procedure which should call the above function to do the actual output for every solution found.

```
private static void makeChange(
  int n,      // number of coins to use
  int r,      // maximum number of repetitions per coin
  int[] F)    // forbidden coins
```

Note that your code should make sure to generate and print the outputs (by calling the above function) in the correct sorted order.

Examples:

Input:	Output:
3 3 0 0 5 1 2 3 4 4 1 6 2 3 4 5 6 7	Case 1 : Number of coins = 3; Repetitions = 0; Forbidden = [] 2 3 6 2 4 4 3 3 3 3 solutions found Case 2 : Number of coins = 5; Repetitions = 1; Forbidden = [3 4] 2 5 6 8 120 2 5 6 9 45 2 5 6 10 30 2 5 6 12 20 4 solutions found Case 3 : Number of coins = 4; Repetitions = 1; Forbidden = [2 3 4 5 6 7] No solutions found

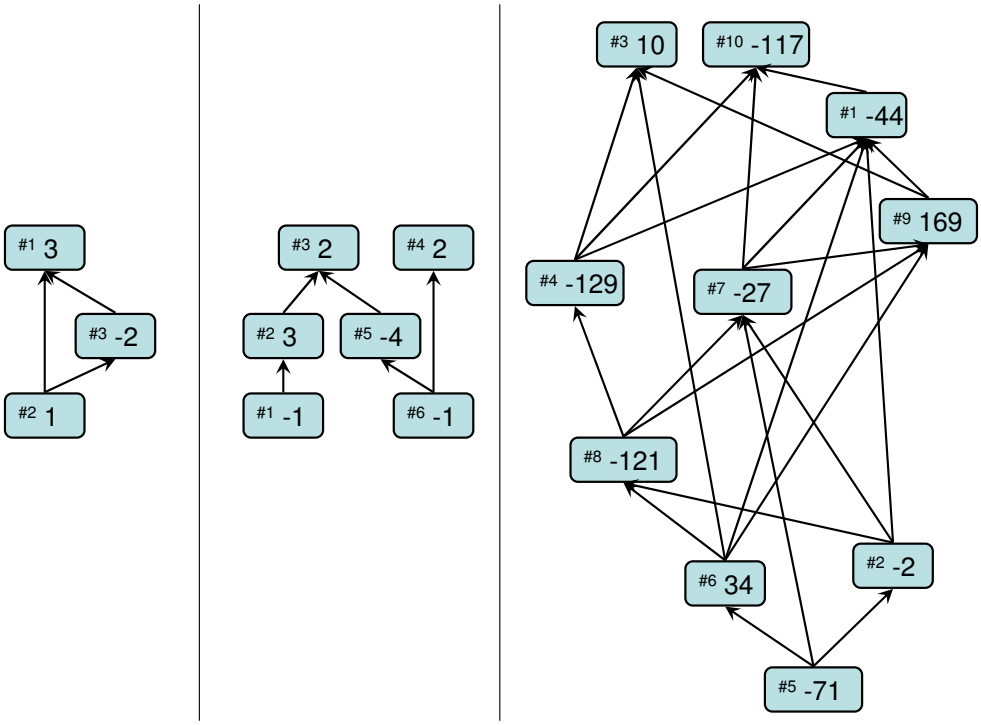
8 Find Fame in Disc Arena

In Grid World, the Disc Arena is a location where programs play gladiator disc games against other programs. There are k games in the disc arena. By winning each game, you gain fame and become popular in the Grid World. Of course, if you lose a game, you are *deleted*. The amount of fame you gain by winning the i^{th} game is f_i (which may not necessary be positive). Furthermore, to be able to play in the i^{th} game, you are required to have first played and won a specified subset S_i of the games (known in advance) – the games in set S_i are also called *prerequisites*. There are no cycles in the prerequisites structure, e.g., it can not happen that a game a is required for game b, b is required for game c, and c is required for playing game a. Note that, the only reason you would play a game with negative fame is because it is a prerequisite to a game with higher positive fame.

A set C of games is called *feasible* if for any game $i \in C$, all the prerequisites S_i are also included in C, i.e., if $i \in C$, then $S_i \subset C$. The fame associated with C is simply the sum of the fames of all games in C.

The goal is of course to become as famous as possible !! Given the fames and the prerequisites for a set of games, your goal is to find a feasible set of games with maximum fame.

For instance, consider the following sets of disc arena games. The prerequisites for each game are indicated using arrows between games. For the first set of games, game 1 has games 2 and 3 as prerequisites, and game 3 has game 2 as a prerequisite. The maximum possible fame is 2, achieved by selecting the set of games {1,2,3}. Game 3 must be selected even though it has a negative fame value, because it is a prerequisite for game 1. For the second set of games, the maximum fame is 3, achieved by selecting the set of games {1,2,4,6}. For the third set of games, the maximum fame is 0, achieved by selecting no games.



Input/Output Format:

Input: The first line of the input is the number of test cases (≤ 100).

The first line in each test case lists k, the number of games ($k \leq 1000$). The games are labeled 1 to k. The next k lines contain the information about the games. The $(i+1)^{th}$ line is in the form "$f_i d_i u_1 u_2 \cdots u_{d_i}$". The first number is the fame you gain by playing game i, the second number is S_i – the number of prerequisites, and the remaining numbers are members of S_i.

Assume that in each scenario, the total number of prerequisites of all games, i.e., $\Sigma_i |S_i|$, is ≤ 6000.

Output: For each scenario, print the maximum fame you can gain by playing a feasible set of games as shown below in the example.

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static int solveDiscArena(int[] fames, int[][] prereqs)
```

which should return the maximum attainable fame that you can gain by playing a feasible set of games. The first argument, `fames`, is an array whose i'th entry contains the fame attainable by playing game i. The second argument captures the prerequisites: `prereqs[i]` contains the prerequisites for game i.

Examples:

Input:	Output:
3 3 3 2 2 3 1 0 -2 1 2 6 -1 0 3 1 1 2 2 5 2 2 1 6 -4 1 6 -1 0 10 -44 5 7 2 4 9 6 -2 1 5 10 3 4 9 6 -129 1 8 -71 0 34 1 5 -27 3 8 5 2 -121 2 6 2 169 3 6 7 8 -117 3 7 1 4	Case 1: Maximum attainable fame = 2 Case 2: Maximum attainable fame = 3 Case 3: Maximum attainable fame = 0

Practice 1

When new programs arrive in the grid world, they start by playing the simplest of games in the Disc Arena against other novice programs. One of those games is played in front of a large board as follows: a sequence of numbers appears on the board, and the players have to decide whether the sum of even numbers in the sequence is larger than the sum of odd numbers in the sequence, or if they are equal. The first player to provide the correct answer wins the game, and the losers are *deleted*.

For example, given the sequence:

5 7 2 1 10 13 6 12

the sum of even numbers is 2 + 10 + 6 + 12 = 30, whereas the sum of odd numbers is 5 + 7 + 1 + 13 = 26. Hence the correct answer here is "EVEN".

You are to help Sam (our protagonist) win this game so he can survive, and look for his father.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (≤ 50). After that, each line contains one test case. The test case begins with the number of elements in the sequence, k (< 100), and then we have k numbers which specify the sequence. Assume all numbers are ≥ 0, and $< 2^{20}$.

Output: For each test case, you are to output "EVEN" (if the sum of even numbers in the sequence is larger than the sum of odd numbers), "ODD" (if the reverse is true), or "TIE" (if the two sums are identical).

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static int checkEvenSumMoreThanOddSum(int[] sequence)
```

which should return 1 if even sum > odd sum, −1 if even sum < odd sum, and 0 if they are equal.

Example:

The output is shown with an extra blank line so that each test case input is aligned with the output; that blank line should not be present in the actual output.

Input:	Output:
4	
8 5 7 2 1 10 13 6 12	EVEN
4 5 2 3 4	ODD
8 1 2 1 2 1 1 1 1	ODD
6 1 1 2 3 5 8	TIE

Practice 2

Sam has discovered a possible communication pathway to the external world that was likely left open by Clu, but he needs to find the correct password to check if it still works. He also knows that Clu has trouble remembering the passwords to all the millions of gates in the Grid world, and usually writes down the passwords somewhere close by. However, Clu does not write down the password directly, but encodes the password in multiple strings written on a wall next to the gate. The first step in decoding the password is to find the *longest decreasing suffix (LDS)* of every string written on the wall; Sam knows how to go about constructing the password afterwards, but needs your help with this first step.

A decreasing suffix of a string is a suffix such that each character in the suffix is larger than the next character. The LDS of a string is the longest such suffix. For example, for string "abcdbca", the LDS is "ca", whereas for the string "abcdljksdflkjzfcba", it is "zfcba".

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (≤ 50). After that, each line contains an input string of length < 100 characters. You can assume that the input strings will only contain lowercase characters from a to z.

Output: For each test case, find the longest decreasing suffix of the input string and print it out as shown below in the example.

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static String findLongestDecreasigSuffix(String s)
```

which should return the LDS of input string s.

Examples:

The output is shown with an extra blank line so that each test case input is aligned with the output; that blank line should not be present in the actual output.

Input:	Output:
3	
abcdbca	The longest decreasing suffix of abcdbca is ca
abcdljksdflkjzfcba	The longest decreasing suffix of abcdljksdflkjzfcba is zfcba
qlpxosmpzygb	The longest decreasing suffix of qlpxosmpzygb is zygb

Practice 3

Sam and Quorra are trying to build a map of a secret base in the Grid world that contains a number of (virtual) *sites* interconnected by (virtual) pathways. Although they know the number of sites (n), their knowledge of interconnections between the sites is sparse. Several different programs that have been inside the base are able to give them some information about the *connections* between the sites, and Sam and Quorra need to put it together. In particular, they would like to make sure that they have enough information to know how to go from every site to every other site.

Your goal is to help them decide if a specified set of connections is enough to ensure that there is a path from every site to every other site. Assume that the connections are bi-directional, i.e., if we know that there is a connection between site 1 and site 2, then we can go from site 1 to site 2, and vice versa.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (≤ 50). After that, each line contains a test case. The first number in each test case is the number of sites, n (≤ 100). The sites are numbered from 0 to $n-1$. The second number is the number of connections between the sites, k (≤ 200). After that there are k pairs of numbers, one for each connection. The connections might repeat and further, there may be self-connections (i.e., a connection from a site to itself).

Output: For each test case, you are to check whether there is path from every site to every other site, and output "Connected." or "Not connected." accordingly.

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static boolean checkIfConnected(int numberOfSites, int[][] connections)
```

which should return *true* if there is a path from every site to every other site, and *false* otherwise.

Examples:

The output is shown with an extra blank line so that each test case input is aligned with the output; that blank line should not be present in the actual output.

Input:	Output:
3	
5 4 0 1 1 2 2 3 3 4	Connected.
6 6 0 1 1 2 2 0 3 4 4 5 5 3	Not connected.
5 9 0 1 1 2 2 3 3 4 1 0 2 1 3 2 2 3 4 3	Connected.