1994-1995 ACM International Collegiate Programming Contest
Western European Regional

Problem A - Presentation Error

One of the main burdens of the Jury of the Scholastic Programming Contest is not to decide whether a submitted program is incorrect, but how to classify the error. In the past, we had 'Failed Testcase', 'Wrong Answer', 'Wrong Output Format', and 'Too much/Too Little Output' to worry about.

The interpretation of these messages depended largely on the jury member involved. For instance, while some believe that 'Wrong Answer' indicates that all answers are wrong, and 'Failed Testcase' applies when at least one answer is right, others feel that 'Wrong Answer' should be used if more than one answer is wrong, and 'Failed Testcase' only if exactly one test went wrong.

Fortunately, all these worries are gone, since now we only need to distinguish between 'Wrong Answer', 'Presentation Error' and 'Accepted' (all other messages are the result of compilation errors, run-time errors, and non-terminating programs).

To eliminate any subjectivity in deciding between a 'Presentation Error' and a 'Wrong Answer', the Jury of this year's Programming Contest has defined an exact procedure to determine whether a program produces a 'Wrong Answer', a 'Presentation Error', or should be 'Accepted'.

In the description of the rules, we distinguish between JuryOut and SubmitOut, as the output intended by the Jury, and the output submitted, respectively. The JuryOut contains parts which are considered essential in the output of a correct algorithm. Those essentials are placed between '[' and ']'. Those brackets are not part of the output, thus they should not appear in SubmitOut. The algorithm to decide between 'Accepted', 'Wrong Answer' and 'Presentation Error' is as follows:

1. From each line in both outputs, all trailing white space (blanks and tabs) should be removed. After that, all trailing empty lines should be removed.
2. If after step 1, JuryOut and SubmitOut are identical, the algorithm returns 'Accepted'.
3. All letters in both outputs are changed to uppercase (including those between square brackets).
4. We name the essentials E1 through En.
5. If each of the strings E1 through En occurs as a string in SubmitOut, and Ei comes after (without overlapping) E(i-1), for all 2<=i<=n, then the algorithm returns 'Presentation Error'.
6. The algorithm returns 'Wrong Answer'.

As we (the Jury) need to have such a program (and we need it NOW), your job is to write it for us.

Input Specification

The input contains on the first line the number of test cases (N). Each test case has on its first line the number (J) of lines in JuryOut, and the number (S) of SubmitOut lines, separated by a single space. Then follow the J lines of JuryOut and the S lines of SubmitOut. Both JuryOut and SubmitOut are no longer than 10 lines. A line is at most 80 characters long. Essentials are non-empty strings that do not cross line boundaries. The first and last characters of an essential are not white space. Essentials will not be nested.

Output Specification

The output has to be ’Accepted’, ’Wrong Answer’, or ’Presentation Error’ on a single line for each test case.

Sample Input

```
4
1 2
Just one line?
Just one line?

2 2
The first characters of the alphabet are:
[abcde]
Here they come:
a b c d e
1 1
That’s it: [abcde]
That’s it: AbCdE
1 1
[2] and [3] make [5]
I guess 2 and 3 are less than 50.
```

Output for the Sample Input

```
Accepted
Wrong Answer
Presentation Error
Presentation Error
```

1994-1995 ACM International Collegiate Programming Contest
Western European Regional

Problem B - Bits

It is not always easy to transfer data from one computer system to the other. You need proper standards for data encoding, and may also need to compress data to save bandwidth and thus reduce costs.

To assist the designer in making implementation choices related to the available bandwidth, a tool is required that computes the size of each message in bits. The tool has to read and interpret the format of each message to do so.

A message can best be described by giving the underlying grammar, which uses the following terminals:

```
id        a sequence of letters of length L (1$\leq$L$\leq$256)
integer   a number between -10 000 and 10 000 (inclusive)
"word"    the literal string of characters word
```

A message is defined by the following grammar:

```
message  ::= data
data     ::= id ":" type
type     ::= record | array | string | enum | range
record   ::= "{" data+ "}"
array    ::= "array" range "of" type
string   ::= "string" "(" integer ")"
enum     ::= "(" id-list ")"
id-list  ::= id | (id "," id-list)
range    ::= "[" integer ".." integer "]"
```

Note that the message grammar is specified according to the following notational conventions:

```
x y      sequence: x followed by y
x | y    choice: x or y
x+       repetition: one or more occurrences of x
( )      used for grouping
```

Any two tokens may be separated by an arbitrary amount of white space (blanks, tabs an newlines). White space does not occur within tokens. The (minimal) amount of bits needed to transmit a message can be computed using the following rules:

```
record   : sum of the sizes of the fields
array    : size of the component type, multiplied by the number of
           elements in the range
string   : the length, multiplied by 7 bits
enum     : smallest number of bits in which all id's can be
           distinguished
range    : smallest number of bits in which all range values can be
           distinguished
```

Input Specification

The input contains on the first line the number of test cases (N). Each test case will contain message according to the grammar above. Messages may be separated by an arbitrary amount of white space. You may assume that the input is syntactically correct. For each range 'L..H', it holds that L<=H. A string consists of a positive number of characters.

Output Specification

For each message, output the sentence: 'A "id" message requires S bits.', where id is the identifier of the message and S its size in bits.

Sample Input

```
3
year : [1970..2030]
team : {
        name : string(14)
        members : array [1..3] of {
                sex : ( male, female )
                name : string(20)
                age : [16..30]
        }
        position : [1..40]
}
jurynames : array [1..3] of string(20)
```

Output for the Sample Input

```
A "year" message requires 6 bits.
A "team" message requires 539 bits.
A "jurynames" message requires 420 bits.
```

1994-1995 ACM International Collegiate Programming Contest
Western European Regional

Problem C - Divisors

Mathematicians love all sorts of odd properties of numbers. For instance, they consider 945 to be an interesting number, since it is the first odd number for which the sum of its divisors is larger than the number itself.

To help them search for interesting numbers, you are to write a program that scans a range of numbers and determines the number that has the largest number of divisors in the range. Unfortunately, the size of the numbers, and the size of the range is such that a too simple-minded approach may take too much time to run. So make sure that your algorithm is clever enough to cope with the largest possible range in just a few seconds.

Input Specification

The first line of input specifies the number N of ranges, and each of the N following lines contains a range, consisting of a lower bound L and an upper bound U, where L and U are included in the range. L and U are chosen such that (1<=L<=U<=1.000.000.000) and (0<=U-L<=10.000).

Output Specification

For each range, find the number P which has the largest number of divisors (if several numbers tie for first place, select the lowest), and the number of positive divisors D of P (where P is included as a divisor). Print the text 'Between L and H, P has a maximum of D divisors.', where L, H, P, and D are the numbers as defined above.

Sample Input

```
3
1 10
1000 1000
999999900 1000000000
```

Output for the Sample Input

```
Between 1 and 10, 6 has a maximum of 4 divisors.
Between 1000 and 1000, 1000 has a maximum of 16 divisors.
Between 999999900 and 1000000000, 999999924 has a maximum of 192 divisors.
```

1994-1995 ACM International Collegiate Programming Contest
Western European Regional

Problem D - Fatman

Some of us may be so fortunate to be thin enough to squeeze through the tiniest hole, others are not. Getting from A to B in a crowded supermarket (even without a cart) can be tough and may require sophisticated navigation: there may seem to be enough room on the one side, but then you may run into trouble with that lady further down...

Let's consider this in an abstract fashion: given an aisle of a certain width, with infinitely small obstacles scattered around, just how fat can a person be and still be able to get from the left side to the right side. Assume that seen from above a (fat) person looks like a circle and the person is incompressible (a person with diameter d cannot go between two obstacles having distance less than d).

Input Specification

The first line of input specifies the number of test cases your program has to process. The input for each test case consists of the following lines:

- One line with the integer length L (0<=L<=100) and integer width W (0<=W<=100) of the aisle, separated by a single space.
- One line with the number of obstacles N (0<=N<=100) in the aisle.
- N lines, one for each obstacle, with its integer coordinates X and Y (0<=X<=L, 0<=Y<=W) separated by a single space.

Output Specification

For each test case given in the input, print a line saying

```
Maximum size in test case N is M
```

where M is rounded to the nearest fractional part of exactly four digits. M is the maximum diameter of a person that can get through the aisle specified for that test case. N is the current test case number, starting at one.

Example Input

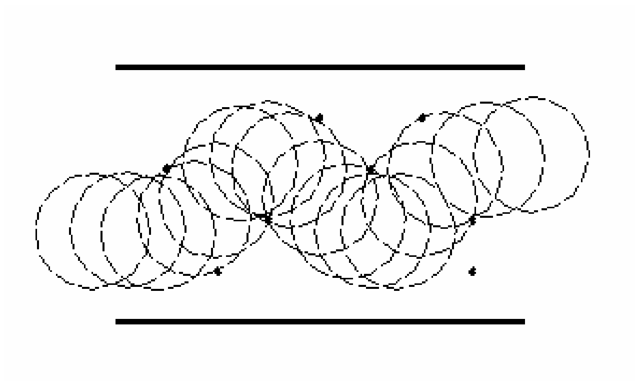
```
1
8 5
8
2 1
1 3
3 2
4 4
5 3
6 4
7 2
7 1
```

Example Output

```
Maximum size in test case 1 is 2.2361.
```

Additional Data

The Example Input looks like:



1994-1995 ACM International Collegiate Programming Contest
Western European Regional

Problem E - Safebreaker

We are observing someone playing the game of Mastermind. The object of this game is to find a secret code by intelligent guess work, assisted by some clues. In this case the secret code is a 4-digit number in the inclusive range from 0000 to 9999, say "3321". The player makes a first random guess, say "1223" and then, as for each of the future guesses, gets a clue telling him how good his guess is. A clue consists of two numbers: the number of correct digits (in this case 1: the "2" at the third position) and the additional number of digits guessed correctly but in the wrong place (in this case 2: the "1" and the "3"). The clue would in this case be: "1/2".

Write a program that given a set of guesses and corresponding clues, tries to find the secret code.

Input Specification

The first line of input specifies the number of test cases (N) your program has to process. Each test case consists of a first line containing the number of guesses G (0<=G<=10), and G subsequent lines consisting of exactly 8 characters: a code of four digits, a blank, a digit indicating the number of correct digits, a '/' and a digit indicating the number of correct but misplaced digits.

Output Specification

For each test case, the output contains a single line saying either:

```
impossible         if there is no code consistent with all guesses.
the secret code    if there is exactly one code consistent with
                   all guesses.
indeterminate      if there is more than one code which is consistent
                   with all guesses.
```

Sample Input

```
4
6
9793 0/1
2384 0/2
6264 0/1
3383 1/0
2795 0/0
0218 1/0
1
1234 4/0
1
1234 2/2
2
6428 3/0
1357 3/0
```

Output for the Sample Input

```
3411
1234
indeterminate
impossible
```

1994-1995 ACM International Collegiate Programming Contest
Western European Regional

Problem F - Quadtrees

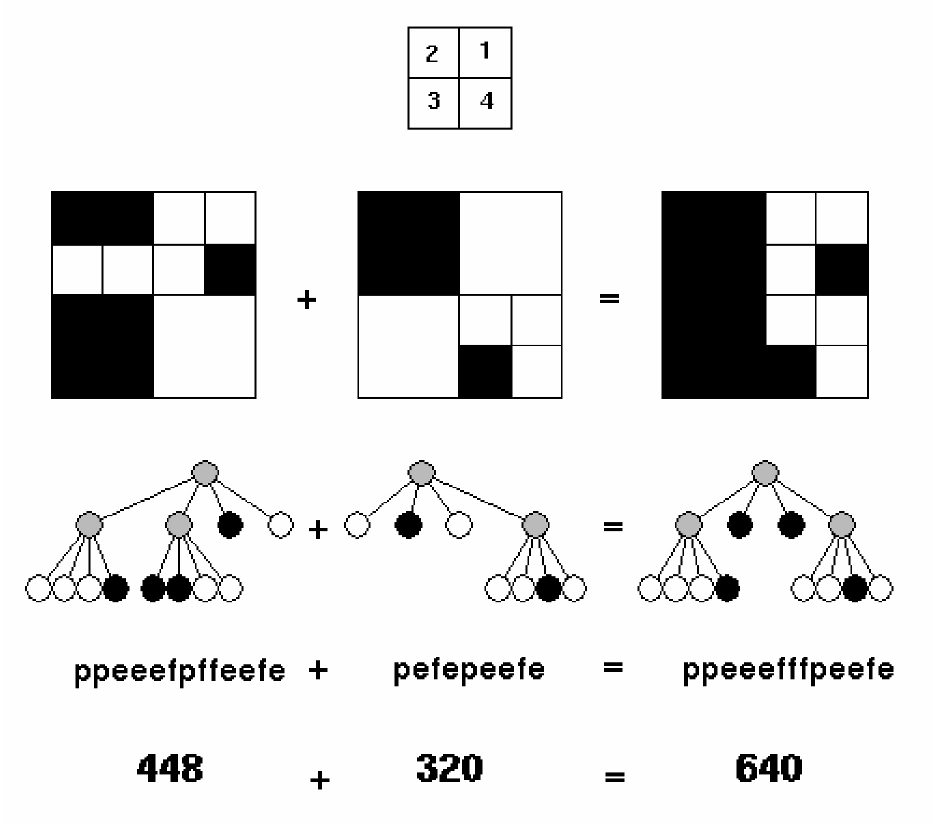
A quadtree is a representation format used to encode images. The fundamental idea behind the quadtree is that any image can be split into four quadrants. Each quadrant may again be split in four sub quadrants, etc. In the quadtree, the image is represented by a parent node, while the four quadrants are represented by four child nodes, in a predetermined order.

Of course, if the whole image is a single color, it can be represented by a quadtree consisting of a single node. In general, a quadrant needs only to be subdivided if it consists of pixels of different colors. As a result, the quadtree need not be of uniform depth.

A modern computer artist works with black-and-white images of 32*32 units, for a total of 1024 pixels per image. One of the operations he performs is adding two images together, to form a new image. In the resulting image a pixel is black if it was black in at least one of the component images, otherwise it is white.

This particular artist believes in what he calls the **preferred fullness**: for an image to be interesting (i.e. to sell for big bucks) the most important property is the number of filled (black) pixels in the image. So, before adding two images together, he would like to know how many pixels will be black in the resulting image. Your job is to write a program that, given the quadtree representation of two images, calculates the number of pixels that are black in the image, which is the result of adding the two images together.

In the figure, the first example is shown (from top to bottom) as image, quadtree, pre-order string (defined below) and number of pixels. The quadrant numbering is shown at the top of the figure.



Input Specification

The first line of input specifies the number of test cases (N) your program has to process. The input for each test case is two strings, each string on its own line. The string is the pre-order representation of a quadtree, in which the letter 'p' indicates a parent node, the letter 'f' (full) a black quadrant and the letter 'e' (empty) a white quadrant. It is guaranteed that each string represents a valid quadtree, while the depth of the tree is not more than 5 (because each pixel has only one color).

Output Specification

For each test case, print on one line the text 'There are X black pixels.', where X is the number of black pixels in the resulting image.

Sample Input

```
3
ppeeefpffeefe
pefepeefe
peeef
peefe
peeef
peepefefe
```

Output for the Sample Input

```
There are 640 black pixels.
There are 512 black pixels.
There are 384 black pixels.
```

1994-1995 ACM International Collegiate Programming Contest
Western European Regional

Problem G - Race Tracks

Many boring math classes have been spent playing Race Tracks, where two players have to maneuver their cars on a race track drawn on a piece of paper, while their cars can only accelerate by a limited (positive or negative) amount per move.

A variant of Race Tracks involves Hoppers. Hoppers are people on a jump stick who can jump from one square to the other, without touching the squares in between (a bit like a knight in chess). Just like the aforementioned cars, they can pick up speed and make bigger hops, but their acceleration per move is limited, and they also have a maximum speed.

Let’s be a bit more formal: our variant of Race Tracks is played on a rectangular grid, where each square on the grid is either empty or occupied. While hoppers can fly over any square, they can only land on empty squares. At any point in time, a hopper has a velocity (x,y), where x and y are the speed (in squares) parallel to the grid. Thus, a speed of (2,1) corresponds to a knight jump, (as does (-2,1) and 6 other speeds).

To determine the hops a hopper can make, we need to know how much speed he can pick up or lose: either -1, 0, or 1 square in both directions. Thus, while having speed (2,1), the hopper can change to speeds (1,0), (1,1), (1,2), (2,0), (2,1), (2,2), (3,0), (3,1) and (3,2). It is impossible for the hopper to obtain a velocity of 4 in either direction, so the x and y component will stay between -3 and 3 inclusive.

The goal of Hopping Race Tracks is to get from start to finish as quickly as possible (i.e. in the least number of hops), without landing on occupied squares. You are to write a program which, given a rectangular grid, a start point S, and a finish point F, determines the least number of hops in which you can get from S to F. The hopper starts with initial speed (0,0) and he does not care about his speed when he arrives at F.

Input specification

The first line contains the number of test cases (N) your program has to process. Each test case consists of a first line containing the width X (1<=X<=30) and height Y (1<=Y<=30) of the grid. Next is a line containing four integers separated by blanks, of which the first two indicate the start point (x1,y1) and the last two indicate the end point (x2,y2) (0<=x1, x2<X, 0<=y1, y2<Y). The third line of each test case contains an integer P indicating the number of obstacles in the grid. Finally, the test case consists of P lines, each specifying an obstacle. Each obstacle consists of four integers: x1, x2, y1 and y2, (0<=x1<=x2<X, 0<=y1<=y2<Y), meaning that all squares (x,y) with x1<=x<=x2 and y1<=y<=y2 are occupied. The start point will never be occupied.

Output specification

The string ’No solution.’ if there is no way the hopper can reach the finish point from the start point without hopping on an occupied square. Otherwise, the text ’Optimal solution takes N hops.’, where N is the number of hops needed to get from start to finish point.

Sample Input

```
2
5 5
4 0 4 4
1
1 4 2 3
3 3
0 0 2 2
2
1 1 0 2
0 2 1 1
```

Output for the Sample Input

```
Optimal solution takes 7 hops.
No solution.
```

1994-1995 ACM International Collegiate Programming Contest
Western European Regional

Problem H - Train Swapping

At an old railway station, you may still encounter one of the last remaining "train swappers". A train swapper is an employee of the railroad, whose sole job it is to rearrange the carriages of trains.

Once the carriages are arranged in the optimal order, all the train driver has to do, is drop the carriages off, one by one, at the stations for which the load is meant.

The title "train swapper" stems from the first person who performed this task, at a station close to a railway bridge. Instead of opening up vertically, the bridge rotated around a pillar in the center of the river. After rotating the bridge 90 degrees, boats could pass left or right. The first train swapper had discovered that the bridge could be operated with at most two carriages on it. By rotating the bridge 180 degrees, the carriages switched place, allowing him to rearrange the carriages (as a side effect, the carriages then faced the opposite direction, but train carriages can move either way, so who cares).

Now that almost all train swappers have died out, the railway company would like to automate their operation. Part of the program to be developed, is a routine which decides for a given train the least number of swaps of two adjacent carriages necessary to order the train. Your assignment is to create that routine.

Input Specification

The input contains on the first line the number of test cases (N). Each test case consists of two input lines. The first line of a test case contains an integer L, determining the length of the train (0<=L<=50). The second line of a test case contains a permutation of the numbers 1 through L, indicating the current order of the carriages. The carriages should be ordered such that carriage 1 comes first, then 2, etc. with carriage L coming last.

Output Specification

For each test case output the sentence:

```
Optimal train swapping takes S swaps
```

where S is an integer.

Example Input

```
3
3
1 3 2
4
4 3 2 1
2
2 1
```

Example Output

```
Optimal train swapping takes 1 swaps.
Optimal train swapping takes 6 swaps.
Optimal train swapping takes 1 swaps.
```