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 E_1 through E_n.
5. If each of the strings E_1 through E_n occurs as a string in SubmitOut, and E_i comes after (without overlapping) E_{i-1}, for all $2 \leq \mathrm{i} \leq \mathrm{n}$, 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.

Example 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.
```

Example Output

```
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
$x \mid y$	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 *ids* 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 $L \leq H$. 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.

Example 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)
```

Example Output

```
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 $1 \leq L \leq U \leq 1\,000\,000\,000$ and $0 \leq U - L \leq 10\,000$.

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.

Example input

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

Example output

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

Output Specification

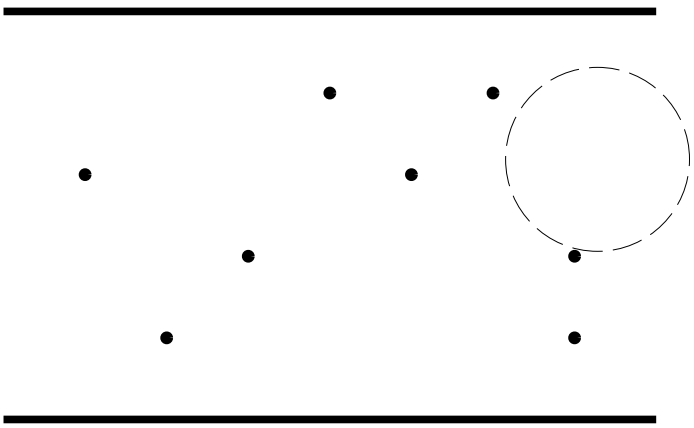
For each test case given in the input, print a line saying '`Maximum size in test case` N `is` M.', where M is rounded to the nearest fractional part of exactly four digits. M is the maximum diameter of a person that can get through the aisle specified for that test case. N is the current test case number, 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: