ACM International Collegiate Programming Contest 1993

Northwestern European Regionals

Problem A

Chess

Almost everyone knows the problem of putting eight queens on an 8×8 chessboard such that no Queen can take another Queen. Jan Timman (a famous Dutch chessplayer) wants to know the maximum number of chesspieces of one kind which can be put on an $m \times n$ board with a certain size such that no piece can take another. Because it's rather difficult to find a solution by hand, he asks your help to solve the problem.

He doesn't need to know the answer for every piece. Pawns seems rather uninteresting and he doesn't like Bishops anyway. He only wants to know how many Rooks, Knights, Queens or Kings can be placed on one board, such that one piece can't take any other.

Input

The first line of input contains the number of problems. A problem is stated on one line and consists of one character from the following set `r`, `k`, `Q`, `K`, meaning respectively the chesspieces Rook, Knight, Queen or King. The character is followed by the integers *m* ($4 \leq m \leq 10$) and *n* ($4 \leq n \leq 10$), meaning the number of rows and the number of columns or the board.

Output

For each problem specification in the input your program should output the maximum number of chesspieces which can be put on a board with the given formats so they are not in position to take any other piece.

Note: The bottom left square is 1, 1.

Sample Input

```
2
r 6 7
k 8 8
```

Sample Output

```
6
32
```

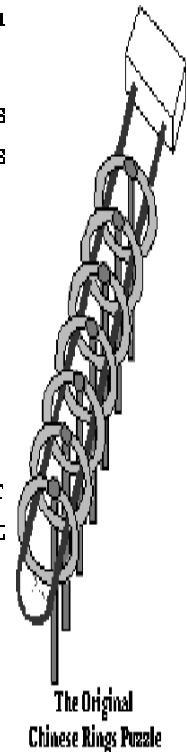
ACM International Collegiate Programming Contest 1993

Northwestern European Regionals

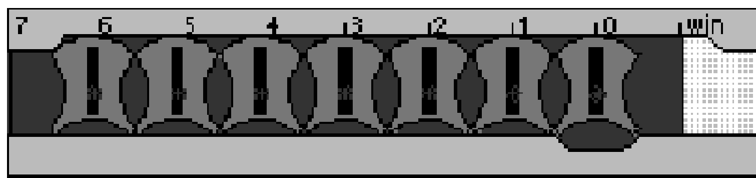
Problem B

Spin

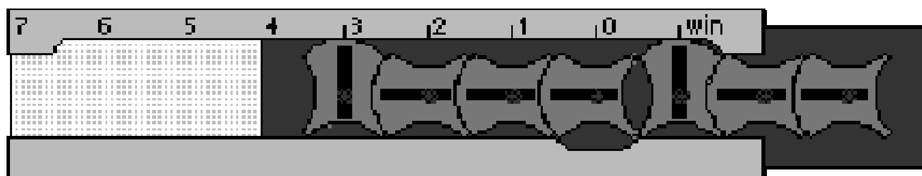
The classic Chinese Rings puzzle comes in a variety of forms. The original version has seven rings linked together by a sliding loop threaded through them. The aim is to remove the loop by manipulating the rings (see right).



A modern implementation uses seven disks with specially shaped cut-outs mounted on a slide. The slide can move left and right. The slide can always move left until it reaches its left-most position, shown here:

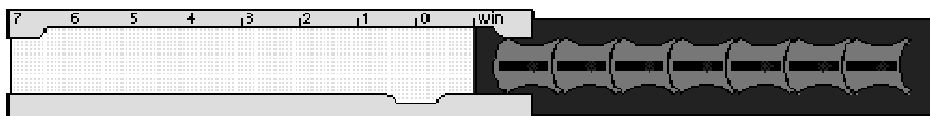


Each disk can be rotated 90°, so the long end of the black bar points either straight up (vertical) or to the left (horizontal). The slide can only move right until a vertical disk hits the end stop under the 'Win' marking:



A disk can be rotated between horizontal and vertical only if it is positioned over the indentation marked '0' *and* the disk on its right is vertical . The right-most disk can always rotate if it is in position '0' since it has no disk on its right.

The aim is to free the slide by moving it so its left edge aligns with the 'Win' mark:



Your task is to write a program which will take several part-solved puzzles and compute the number of steps needed to move the slide to position 'Win' for each puzzle.

Input

There will be several puzzles in the input file. The first line of the file will contain an integer *n* specifying the number of puzzles. There will then be *n* lines, each of the form:

length orientations position

where *length* is an integer indicating the number of disks on the slide, *orientations* is a string of *length* characters from the set {`h`,`v`} giving the orientation of each disk from left to right, and *position* is an integer from 0 to *length* specifying the numbered mark which aligns with the left edge of the slide.

Output

For each puzzle, your program should output one integer on a line which counts the minimum number of steps needed to win the puzzle. A step is either a movement of the slide, one unit left or right, or the rotation of a disk.

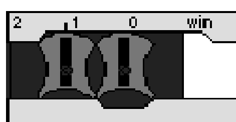
Sample Input

```
2
2 vv 2
7 vhhhvhh 4
```

Output for the Sample Input

```
7
357
```

Diagram of the 1st Puzzle



ACM International Collegiate Programming Contest 1993

Northwestern European Regionals

Problem C

Vertex

Write a program that searches a directed graph for vertices which are inaccessible from a given starting vertex.

A directed graph is represented by *n* vertices where $1 \leq n \leq 100$, numbered consecutively $1 \ldots n$, and a series of edges *p* -> *q* which connect the pair of nodes *p* and *q* in one direction only.

A vertex *r* is reachable from a vertex *p* if there is an edge *p* -> *r*, or if there exists some vertex *q* for which *q* is reachable from *p* and *r* is reachable from *q*.

A vertex *r* is inaccessible from a vertex *p* if *r* is not reachable from *p*.

Input

The input data for this program consists of several directed graphs and starting nodes.

For each graph, there is first one line containing a single integer *n*. This is the number of vertices in the graph.

Following, there will be a group of lines, each containing a set of integers. The group is terminated by a line which contains only the integer 0. Each set represent a collection of edges. The first integer in the set, *i*, is the starting vertex, while the next group of integers, $j \ldots k$, define the series of edges *i* -> $j \ldots i$ -> *k*, and the last integer on the line is always 0. Each possible start vertex *i*, $1 \leq i \leq n$ will appear once or not at all. Following each graph definition, there will be a one line containing list of integers. The first integer on the line will specify how many integers follow. Each of the following integers represents a start vertex to be investigated by your program. The next graph then follows. If there are no more graphs, the next line of the file will contain only the integer 0.

Output

For each start vertex to be investigated, your program should identify all the vertices which are inaccessible from the given start vertex. Each list should appear on one line, beginning with the count of inaccessible vertices and followed by the inaccessible vertex numbers.

Sample Input

```
3
1 2 0
2 2 0
3 1 2 0
0
2 1 2
0
```

Sample Output

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

ACM International Collegiate Programming Contest 1993

Northwestern European Regionals

Problem D

Rubik's Cube

The Rubik's cube has been the source of many fruitless hours of human puzzling. Now it affords a chance for some computer puzzling too! In case you have been on Mars for the past decade, a Rubik's cube is covered with 54 facelets, 9 facelets on each of its six sides. Each facelet may be a certain colour. Most cubes have their facelets painted in the six colours red, yellow, green, blue, white and magenta, but any six colours can be used. People adept at playing with the cubes soon come to recognise patterns which lead to a solution. They are able to do this regardless of the orientation of the puzzle, or of the set of six colours used for its faces.

You are to write a program to assist with this task of pattern recognition. Your program will be given the colours of the facelets of a pair of Rubik's cubes, and it must determine whether the two cubes are 'the same'.

Two cubes are the same if one cube can be made to appear identical with the other by some combination of the following operations:

- Rotating the cube *as a whole* in steps of 90° about one of the six axes running through the centres of its opposite faces
- Repainting all the facelets of a particular colour with some other colour which does not currently appear on any facelet of that cube.

Input

The input begins with an integer, *n*, indicating the number of pairs of cubes your program must process. *n* pairs of cubes then appear. Each pair is represented as two flattened cubes shown side by side, in the 49×9 character form shown in the example input below. Any twelve letters taken from the set {`A...Z`}, {`a...z`} may appear in place of the letters used in the example. A set of exactly six distinct letters will be used for the first cube, and another set of six letters, distinct from each other but not necessarily distinct from the first set, will be used for the second cube. A blank line separates each pair of cubes from the next.

Output

For each pair of cubes in the input, your program should output one line; if the two cubes are the same by the above definition, your program should output:

`same`

If not, it should say:

```
different
```

Sample Input

```
1
. . . B B B . . . . . . | . . . b b b . . . . . .
. . . B B B . . . . . . | . . . b b b . . . . . .
. . . B B B . . . . . . | . . . b b b . . . . . .
L L L T T T R R R U U U | l l l t t t r r r u u u
L L L T T T R R R U U U | l l l t t t r r r u u u
L L L T T T R R R U U U | l l l t t t r r r u u u
. . . F F F . . . . . . | . . . f f f . . . . . .
. . . F F F . . . . . . | . . . f f f . . . . . .
. . . F F F . . . . . . | . . . f f f . . . . . .
```

Sample Output

```
same
```

ACM International Collegiate Programming Contest 1993

Northwestern European Regionals

Problem E

Rename

In MS-DOS there exists a '`rename`' command that allows you to change the name of a file. There is an equivalent command in Unix called '`mv`'. Both commands take arguments in the same way:

```
rename oldname newname
mv oldname newname
```

However, the two commands treat the wild-card character '`*`' quite differently. In MS-DOS, you can say:

```
rename old* new*
```

and you will find that any filenames you have that previously began with the three characters '`old`' have those characters replaces by '`new`'. Try the equivalent under Unix and you will probably get an error message `:-(` '`mv`' will only take the simple two argument, no wild-card form.

To rectify this discrepancy, your program must convert a '`rename`' command in to a series of '`mv`'s'.

Input

First comes a list of filenames. These appear one per line. The list is terminated by a line containing the word '`end`'. Following the list of filenames is the sequence of '`rename`' commands. Each command appears on one line in the form:

`rename` *wildfrom wildto*

from and *to* will both contain one wild-card character, '`*`'. After the last '`rename`' command will be a line containing only the word '`end`'.

Output

For each rename command in the input, you program should first echo the rename command itself, in the same form as the input:

`rename` *wildfrom wildto*

Following that, your program should output the set of '`mv`' commands needed to perform the equivalent renaming. Each '`mv`' should appear on its own line in the form:

`mv` *from to*

Notes: The real MS-DOS '`*`' has some odd properties which do not concern us here. For example, an MS-DOS '`*`' will match at most eight characters, none of which is a period '`.`'. No such restrictions apply to our idealised '`*`' which will match any number of any printable character. MS-DOS treats upper and lower case letters the same. Unix treats the two cases as distinct, as should your program. MS-DOS limits filenames to 12 characters, including a '`.`' fixed at the 9th position. Some versions of Unix limit filenames to 14 characters. This is the limit your program should assume. Each '`rename`' command should be performed on the original list of filenames, not on the results of the previous command.

Sample Input

```
abFile001.c
abFile001.cxx
abprog001.c
abfile.c
abFile.c
abFileprog.c
end
rename abFile*.c bprog*.cxx
end
```

Sample Output

```
rename abFile*.c bprog*.cxx
mv abFile001.c bprog001.cxx
mv abFile.c bprog.cxx
mv abFileprog.c bprogprog.cxx
```

ACM International Collegiate Programming Contest 1993

Northwestern European Regionals

Problem F

Compress

Nowadays everyone is using compression methods to reduce the space occupied by data. In some cases this is done in such a way you hardly notice it, for example tapestreamers, modems and harddisk doubling programs. In other cases you have to do it yourself by using *pack*, *arj*, *zip*, *zoo* or *arc*.

Most compression-methods are very complicated, but for this problem only the following simplified method is used. An ASCII-character consists of eight bits, which allows the encoding of 256 different characters. It's possible to recode the characters with a new sequence of bits. These sequences may have a different length. When you choose to give the characters which are often used a shorter sequence of bits than the characters which are seldom used the total text size will be reduced.

Recoding characters gives one other problem, which occurs when you try to get the text back. In a standard ASCII text you know the first character begins at the 1st and ends at the 8th bit, the second begins at the 9th and ends at the 16th bit and so on. When using a variable length coding you don't know where a character begins. For example when an '`a`' is coded as '11' and an '`e`' as '1111', given the bit-sequence '111111' you don't know if it means '`ae`', '`ea`' or '`aaa`'.

The last problem is solved when the next principle is used. Suppose you want to give some characters a code length of 3 (and you haven't given any character a code yet). In that case you've got 8 (2 to the power 3) possibilities. Seven codes can be immediately allocated to characters. The last one depends on how many more characters you have left to code. If you only have one left to code, the last code will do, but if you have to code more than one, you must use the last code to indicate that an extension follows. The extension has the same structure. This has to be continued until all different characters in the text are given a code.

For example you've the characters '`a`', '`e`', '`i`', '`o`', '`u`' and '`y`'. You choose to give the first three characters a code with length 2 and the rest an equal length. So '`a`' becomes '00', '`e`' becomes '01' and '`i`' becomes '10'. The other characters have to be an extension of '11'. Then '`o`' becomes '1100', '`u`' becomes '1101' and '`y`' becomes '1110'. The code '1111' is free now. If 7 characters instead of 6 characters should be given a code this last code would be sufficient. If more than 7 characters should be given a code, the last code would be extended and so on.

You only have to code the characters which occur in the text. You don't have to give the code table itself.

You must write a program that reduces a given text by recoding each character and give the minimum total filelength in bits.

Input

The first line of the input file contains the number of problems. A problem starts with the number of lines *n* to follow, followed by *n* lines. A line consists of characters and the characters 'A'..'Z', 'a'..'z', space, '.', ',' , '-' and '$' are allowed. The character '$' will always be at the end of a line and cannot occur anywhere else. This character has to coded instead of the real end of line mark.

Output

For each problem in the input your program should output the minimal number of bits to code the given text.

Sample Input

```
2
3
Hello Contestant,$
Please write a program which gives$
the text Hello world.$
1
To be or not to be, that is the question.$
```

Sample Output

```
335
167
```

ACM International Collegiate Programming Contest 1993

Northwestern European Regionals

Problem G

Logic

Consider a 10×10 grid. Cells in this grid can contain one of five logic operations (AND, OR, NOT, Input, Output). These can be joined together to form a logic circuit. Given a description of a circuit and a set of boolean values, build the logic circuit and execute the input stream against it.

Input

The first line of the input contains a single integer *n*, which specifies the number of circuits to be processed. There will then be *n* groups of circuit descriptions and test values.

A circuit is made up of a number of operations. Each line describing an operation begins with three characters: the co-ordinates for a cell, 0-9 on the *X*-axis then 0-9 on the *Y*-axis, followed by a single character to represent the operation of that cell (‘`&`’ for AND, ‘`|`’ for OR, ‘`!`’ for NOT, ‘`i`’ for Input and ‘`o`’ for Output). Optionally following each triple is a set of co-ordinate pairs which represent the *x* and *y* co-ordinates of cells that take the output of this cells operation as an input for theirs. This (possibly empty) output list is terminated by ‘`..`’. The list of operations is terminated by a line containing the word ‘`end`’.

Next, for each circuit, comes the set of test values. The first line contains an integer *t* which gives the number of test cases your program must run. Next, there are *t* lines, each line containing a sequence of ‘`0`’ and ‘`1`’ characters symbolising the input values for one test case. The number of inputs will always correspond to the number of inputs defined by the circuit description. The input values are to be applied to the inputs in the order in which the input operations were defined in the circuit description.

The next circuit description, if any, will then follow.

Output

For each circuit, your program should output one line for each test case given in the input. The line should contain one ‘`0`’ or ‘`1`’ character for each output defined by the circuit description in the order in which the outputs were defined.

Your program should output a blank line after each set of test cases.

Sample Input

```
1
00i 11 13 ..
```

```
02i 11 13 ..
11& 21 ..
21o ..
13| 23 ..
23o ..
end
4
00
01
10
11
```

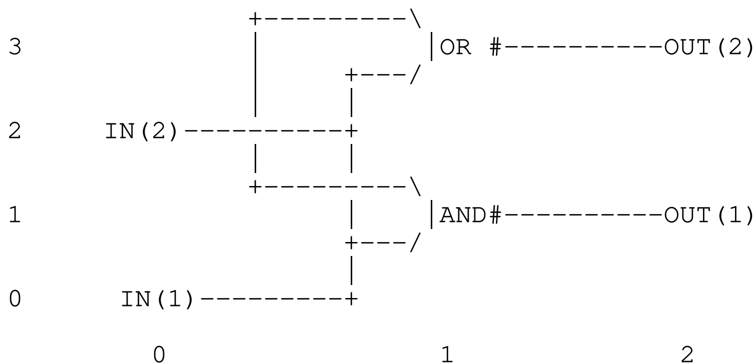
Sample Output

```
00
01
01
11
```

Notes:

- i, o and ! operations will always have exactly one input.
- & and | operations will always have exactly two inputs.
- Even if an operation can feed others, it does not have to.
- No recursive circuits.

Hint: Sample input specifies a circuit consiting of an ‘AND’ and an ‘OR’ operation in parallel both fed from the same two inputs:



In grid terms this is two inputs at 0,0 and 1,0. The first input feeds the AND operation at 1,1 and the OR operation at 1,3. The second input operation feeds the second input for the same AND and OR operations. The AND operation then feeds an output operation at 2,1. The OR operation also feeds an output operation, this one at 2,4.