working boi

1995-1996 ACM International Collegiate Programming Contest
Western European Regional

Problem A
Modern Art

The famous painter Mel Borp is working on a brilliant series of paintings that introduce a new experimental style of Modern Art. At first glance, these paintings look deceptively simple, since they consist only of triangles of different sizes that seem to be stacked on top of each other. Painting these works, however, takes an astonishing amount of consideration, calculation, and precision since all triangles are painted without taking the brush off the canvas. How exactly Mel paints his works is a well-kept secret.

Recently, he started on the first painting of his new series. It was a single triangle, titled $T_{0,0}$. After that, he created $T_{1,1}$, the basis for his other works (see Figure 1).

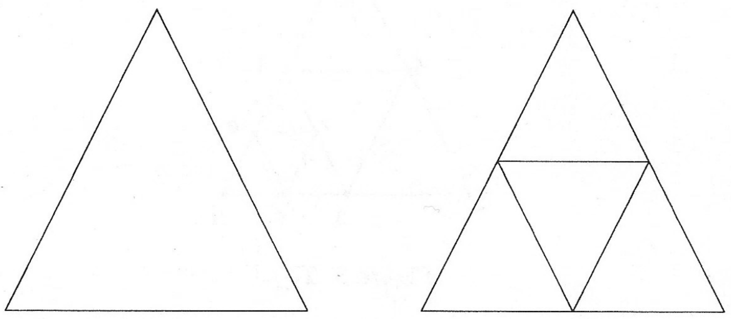


Figure 1: Early work: $T_{0,0}$ (left) and $T_{1,1}$ (right).

Then he decided to take his experimenting one step further, and he painted $T_{1,2}$ and $T_{3,2}$. Compare Figure 1 and Figure 2 to fully appreciate the remarkable progression in his work.

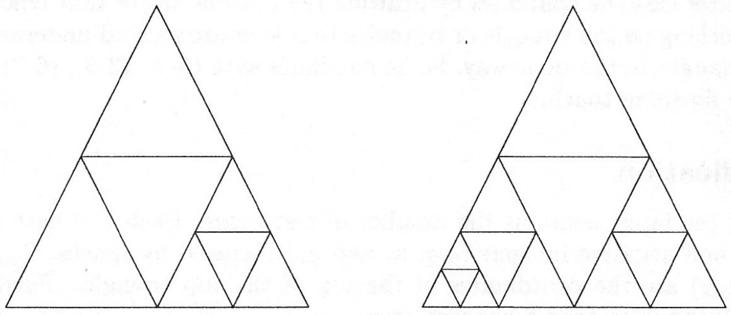


Figure 2: Advanced work: $T_{1,2}$ (left) and $T_{3,2}$ (right).

Note that the shape of the painting can be deduced from its title, $T_{p,q}$, as follows:

- $T_{0,0}$ is a single triangle (see Figure 1);
- $T_{1,1}$ consists of a smaller, inverted triangle placed inside $T_{0,0}$, so that the result consists of four smaller triangles (see Figure 1);
- if $p > 0$ and $q > 1$, then the painting looks like $T_{p,q-1}$, except that an inverted triangle has been placed inside the bottom-right triangle of $T_{p,q-1}$, splitting it into four smaller triangles (compare for example $T_{1,1}$ and $T_{1,2}$);

- if $p > 1$ and $q > 0$, the painting looks like $T_{p-1,q}$, except that an inverted triangle has been placed inside the bottom-left triangle of $T_{p-1,q}$, splitting it into four smaller triangles;
- other values of p and q: it is not a valid title of a painting.

The triangles of a painting look all the same (each triangle is an isosceles triangle with two sides of the same length), but their height and width depend on the size of the canvas Mel used.

Mel wanted to end the series with $T_{10,10}$, the most complex painting he thought he would be able to paint. But no matter how many times he tried, he could not get it right. Now he is desperate, and he hopes you can help him by writing a program that prints, in order, the starting and ending coordinates of the lines Mel has to paint. Of course, you will need to know how Mel paints his works, so we will now reveal his secret technique.

As an example, take a look at $T_{1,2}$ (see Figure 3):

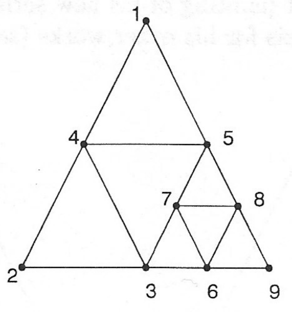


Figure 3: $T_{1,2}$.

Mel always starts at the top of the top triangle, drawing a line straight to the lower-left corner of the lower left triangle (in this example, 1-2), continuing with a line to the lower-right corner of that triangle (in this example, 2-3). Next, he works his way up by drawing a line to the top of that triangle (in this example, 3-4). Now he has either reached the starting point again (finishing yet another masterpiece) or he has reached the lower-left corner of another triangle (in this example, 1-4-5). In the latter case, he continues by drawing the bottom line of that triangle (4-5) and after that he starts working on the triangle or triangles that is or are located underneath the lower-right corner of that triangle, in the same way. So he continues with (5-3), (3-6), (6-7), (7-8), (8-6), (6-9), and (9-1) as the finishing touch.

Input Specification

The first line of the input contains the number of test cases. Each test case consists of one line containing four non-negative integers p, q, x, and y, separated by spaces. $T_{p,q}$ is the title of the painting and (x, y) are the coordinates of the top of the top triangle. Further, $p, q \leq 10$ and $x, y < 32768$. All triangles have a nonzero area.

Output Specification

For every test case, the output contains the pairs of (x, y) integer coordinates of the starting and ending points of all lines Mel has to draw for the painting $T_{p,q}$ in the right order, in the format

$$(startX, startY)\ (endX, endY)$$

followed by a newline. The output for each test case must be followed by an empty line.

Example Input

```
2
0 0 1 1
1 2 512 1024
```

Example Output

```
(1,1)(0,0)
(0,0)(2,0)
(2,0)(1,1)

(512,1024)(0,0)
(0,0)(512,0)
(512,0)(256,512)
(256,512)(768,512)
(768,512)(512,0)
(512,0)(768,0)
(768,0)(640,256)
(640,256)(896,256)
(896,256)(768,0)
(768,0)(1024,0)
(1024,0)(512,1024)
```

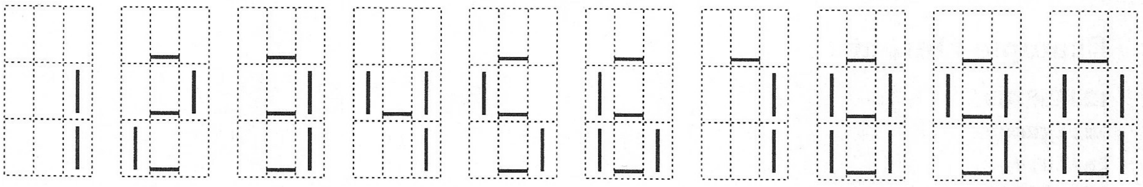
1995-1996 ACM International Collegiate Programming Contest
Western European Regional

Problem B
Bank (Not Quite O.C.R.)

Banks, always trying to increase their profit, asked their computer experts to come up with a system that can read bank cheques; this would make the processing of cheques cheaper. One of their ideas was to use optical character recognition (ocr) to recognize bank accounts printed using 7 line-segments.

Once a cheque has been scanned, some image processing software would convert the horizontal and vertical bars to ASCII bars '|' and underscores '_'.

The ASCII 7-segment versions of the ten digits look like this:



A bank account has a 9-digit account number with a checksum. For a valid account number, the following equation holds: $(d_1 + 2 \times d_2 + 3 \times d_3 + \cdots + 9 \times d_9) \bmod 11 = 0$. Digits are numbered from right to left like this: $d_9d_8d_7d_6d_5d_4d_3d_2d_1$.

Unfortunately, the scanner sometimes makes mistakes: some line-segments may be missing. Your task is to write a program that deduces the original number, assuming that:

- when the input represents a valid account number, it is the original number;
- at most one digit is garbled;
- the scanned image contains no extra segments.

For example, the following input

```
    _  _     _  _  _  _  _ 
  | _| _||_||_ |_   ||_||_|
  | _  _|  | _||_|  ||_| _|
```

used to be "123456789".

Input Specification

The input file starts with a line with one integer specifying the number of account numbers that have to be processed. Each account number occupies 3 lines of 27 characters.

Output Specification

For each test case, the output contains one line with 9 digits if the correct account number can be determined, the string "`failure`" if no solutions were found and "`ambiguous`" if more than one solution was found.

Example Input

```
4

    _  _     _  _  _  _  _
  | _| _||_||_ |_   ||_||_|
  | _  _|  | _||_|  ||_| _|

    _  _  _  _  _  _     _
|_||_|| || ||_   |  |  ||_
  | _||_||_||_|  |  |  | _|

 _  _  _  _  _  _  _  _  _
|_||_||_||_||_||_||_||_||_|
|_||_||_||_||_||_||_||_||_|

 _     _  _  _  _  _  _  _
|_|  ||_||_||_||_||_||_||_|
|_|  ||_||_||_||_||_||_||_|
```

Example Output

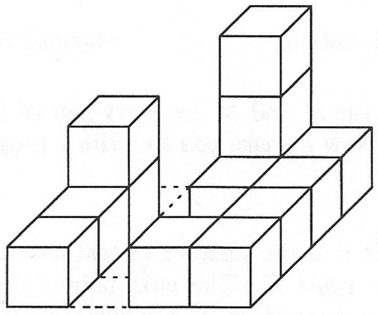
```
123456789
ambiguous
failure
878888888
```

1995-1996 ACM International Collegiate Programming Contest
Western European Regional

Problem C
Matty's Blocks

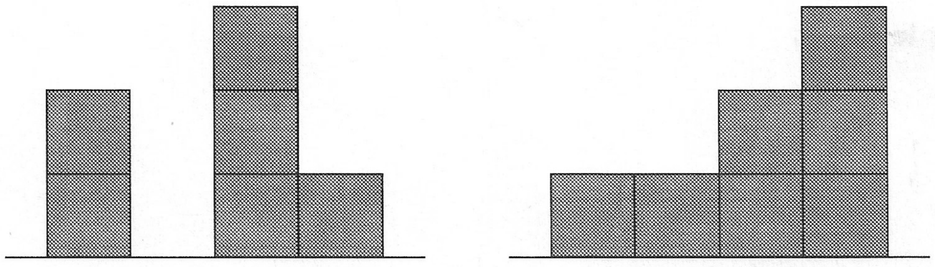
Little Matty liked playing with his blocks very much. He always constructed his 'buildings' in the same way: he made stacks of one or more blocks, and he put those stacks on a square table that was exactly K blocks wide ($K = 8$ for his largest table, so it could contain up to 8×8 block stacks). He didn't put the stacks randomly on the table. No, he always made a nice 'square' pattern. In most buildings, there was no pattern visible in the heights of the stacks. However, since Little Matty himself was only eight blocks tall, a single stack of blocks never consisted of more than eight blocks.

This is an example of one of his buildings. It was built on a table that could contain 4×4 block stacks.



He liked drawing too. To record his block buildings for future generations, he would draw them on paper. Since drawing a three-dimensional block building just was too hard for him, he made two drawings of a building: one straight from the front (you could only see the front of the blocks), and one from the right (you could only see the right side of the blocks). The drawings were in fact two-dimensional projections of the block building, showing only its outline on the front or on the right side.

These are the drawings he made of the building shown above.

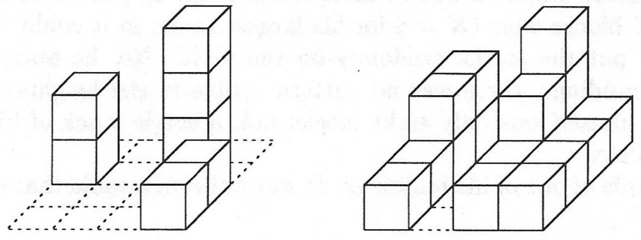


Front *Right side*

He thought that such a pair of drawings would give enough information to be able to re-build the block building later (but he never tried it).

Years later, looking again at the drawings, he realized that this was not the case: from most pairs of drawings, he was able to construct more than one building that had the same outlines (front and right side) as the original one. He found out that from every (front, side)-pair of drawings that he had made, he could construct a 'minimal' building, one that has the same outlines as the original building and contains a minimal number of blocks N (so it was not possible to construct a building with the same outlines with less than N blocks). Furthermore, he could add more blocks to this minimal building, so that the outlines remained the same, up to the point that he had added M ($M \geq 0$) blocks and he had a 'maximal' building, one that still had the same outlines as his minimal one, and adding one extra block would result in a building with a different outline (so there are no buildings with the same outlines that contain more than $N + M$ blocks).

As an example, these are minimal and maximal buildings for the drawings shown above. In this case, $N = 7$ and $M = 10$.



Minimal Building Maximal Building

Matty started determining the N and M for every pair of drawings he had made, but soon he found this task to be tedious. Now he asks *you* to write a program that does the job for him!

Input Specification

The input contains on the first line the number of test cases. Each test case starts with a line containing only the size of the table K. The next pair of lines each contain the description of one drawing. Each description consists of K non-negative integers separated by spaces. Each number indicates the height of the corresponding projection of a stack of blocks in the drawing. The description of the front drawing always precedes the description of the right side drawing. From each pair of drawings at least one block building can be constructed.

Output specification

For each test case output the following line:

`Matty needs at least` N `blocks, and can add at most` M `extra blocks.`

Sample Input

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

Sample Output

```
Matty needs at least 7 blocks, and can add at most 10 extra blocks.
Matty needs at least 1 blocks, and can add at most 0 extra blocks.
```

1995-1996 ACM International Collegiate Programming Contest
Western European Regional

Problem D

не тео

Block Voting

Different types of electoral systems exist. In a *block voting system* the members of a party do not vote individually as they like, but instead they must **collectively** accept or reject a proposal. Although a party with many votes clearly has more power than a party with few votes, the votes of a small party can nevertheless be crucial when they are needed to obtain a majority. Consider for example the following five-party system:

party	votes
A	7
B	4
C	2
D	6
E	6

Coalition $\{A, B\}$ has $7 + 4 = 11$ votes, which is not a majority. When party C joins coalition $\{A, B\}$, however, $\{A, B, C\}$ becomes a winning coalition with $7+4+2 = 13$ votes. So even though C is a small party, it can play an important role.

As a measure of a party's power in a block voting system, John F. Banzhaf III proposed to use the *power index*.[1] The key idea is that a party's power is determined by the number of minority coalitions that it can join and turn into a (winning) majority coalition. Note that the empty coalition is also a minority coalition and that a coalition only forms a majority when it has more than half of the total number of votes. In the example just given, a majority coalition must have at least 13 votes.

In an ideal system, a party's power index is proportional to the number of members of that party.

Your task is to write a program that, given an input as shown above, computes for each party its power index.

Input Specification

The first line contains a single integer which equals the number of test cases that follow. Each of the following lines contains one test case.

The first number on a line contains an integer P in $[1 \ldots 20]$ which equals the number of parties for that test case. This integer is followed by P positive integers, separated by spaces. Each of these integers represents the number of members of a party in the electoral system. The i-th number represents party number i. No electoral system has more than 1000 votes. $\Sigma P \leq 1000$

Output Specification

For each test case, you must generate P lines of output, followed by one empty line. P is the number of parties for the test case in question. The i-th line (i in $[1 \ldots P]$) contains the sentence:

```
party i has power index I
```

where I is the power index of party i.

[1]See John F. Banzhaf. One Man, 3,312 Votes: A Mathematical Analysis of the Electoral College. *Villanova Law Review*, 13:304–346, Winter 1968.

Example Input

```
3
5 7 4 2 6 6
6 12 9 7 3 1 1
3 2 1 1
```

Example Output

```
party 1 has power index 10
party 2 has power index 2
party 3 has power index 2
party 4 has power index 6
party 5 has power index 6

party 1 has power index 18
party 2 has power index 14
party 3 has power index 14
party 4 has power index 2
party 5 has power index 2
party 6 has power index 2

party 1 has power index 3
party 2 has power index 1
party 3 has power index 1
```

[illegible] bewaren tot later

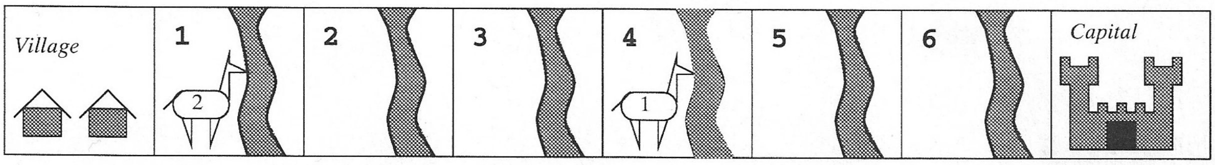
1995-1996 ACM International Collegiate Programming Contest
Western European Regional

Problem E
Donkey

Long ago, a number of farmers wanted to sell their donkeys. These farmers lived in a small village and the marketplace was in the capital. There was only one small road from the village the farmers lived in to the capital, and one day the farmers left for the long journey towards the capital. Every farmer wanted to travel as fast as possible to be the first to sell his donkey, but there was one major problem: the donkeys were very stubborn; everytime a donkey reached a river there was a chance that he would stay there for some time, not willing to cross the river. However, two donkeys never rested at the same river.

The game 'Donkey' is derived from this legend: N players (numbered from $1..N$) start with their donkey in the village. Between the village and the capital are M rivers. The players take turns in throwing a fair, 6-sided die and move their donkey the thrown number of rivers towards the capital. Since only one donkey can rest at a river, the donkey is placed at the next free river if necessary. After player 1 has thrown the die, it is player 2's turn, etc. After N comes 1. The player that passes all rivers first wins the game.

This is an example of a 'Donkey' gameboard, where $N = 2$ and $M = 6$. Player 1's donkey is located at river 4 and player 2's donkey is located at river 1.



Your task is to give the probability that player 1 wins the game, given a certain board position.

Input Specification

The first line contains a single integer which equals the number of test cases that follow. Each of the following lines contains one test case. The first integer on a line gives the number of rivers M, the second integer gives the number of players N ($1 \leq N \leq 4$ and $N \times M \leq 50$). Then follow N integers P_i ($0 \leq P_i \leq M$, $1 \leq i \leq N$), representing the position of the ith player. The river closest to the village has number 1, the river closest to the capital has number M. The village has number 0. Two donkeys may not be positioned at the same river. However, more than one donkey may be standing in the village. The last integer on the line gives the player, whose turn it is.

Output Specification

For every situation you have to print the following line, giving the game number (1 for the first, 2 for the second, ...) and the probability player 1 wins with an accuracy of 3 decimals:

```
Game N:the probability that player 1 wins = D.DDD
```

[illegible]

Example Input

```
3
3 2 1 2 1
6 3 1 2 3 2
6 3 4 1 3 2
```

Example Output

```
Game 1:the probability that player 1 wins = 0.667
Game 2:the probability that player 1 wins = 0.093
Game 3:the probability that player 1 wins = 0.366
```

1995-1996 ACM International Collegiate Programming Contest
Western European Regional

Problem F

Islands

Now that the Eurotunnel between France and Great Britain is in operation, the French government is thinking about a new project to expand the connections with their British neighbors. Inspired by the movie "Bridges of Madison County," their prime architect Clouseau has been working on a plan to connect France to the Channel Islands by building a number of bridges.

To minimize the costs of the bridges, a necessity to get the plan approved, Clouseau has been working hard to find the best places to build the bridges that connect the islands to France. He has figured out that connecting each island to France individually is not the cheapest solution; it is cheaper to construct a bridge between one island and France, and build additional bridges to connect the other islands indirectly. Clouseau, however, is unable to find the best places to build the bridges because of the irregular shapes of the islands.

By approximating the islands as circles, Clouseau was able to report his boss, Mr. Dreyfus, an estimate of the length of the interconnecting bridges. Mr. Dreyfus, however, is not satisfied and demands that Clouseau report by Monday the *exact* length of the bridges needed based on the actual shapes of the Channel Islands. If Clouseau does not report on time he will be fired. Would you be so kind to help out Clouseau and write a program that computes the minimum length of the bridges needed to interconnect the Channel Islands?

Input Specification

The first line of input contains the number of test cases. A test case consists of one line holding the number of islands ($2 \le N \le 100$), followed by N lines that describe the islands. An island is a polygon, which is described as a number ($1 \le P \le 25$) that gives the number of points followed by P pairs of coordinates. Each coordinate is an integer in the range $[-1000 \ldots 1000]$. The points are listed in order such that by connecting consecutive points, and the last point to the first, the perimeter of the island is given. It is guaranteed that islands do not touch or intersect.

Output Specification

For each test case output one line reporting the minimal interconnect as follows:

`The minimal interconnect consists of` N `bridges with a total length of` L

Where N is the number of bridges, and L is the total length, which should be printed as a floating point number with an accuracy of three digits.

Example Input

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

Example Output

```
The minimal interconnect consists of 2 bridges with a total length of 2.000
```

1995-1996 ACM International Collegiate Programming Contest
Western European Regional

Problem G
Maze

Johnny likes solving puzzles. He especially likes drawing and solving mazes. However, solving a maze he has drawn himself is too easy for him.

Since his computer is his best friend, he figures that he needs a program drawing the mazes for him. So he starts thinking about an algorithm performing this difficult task for him and he comes up with 'Johnny's Simple Algorithm.'

Johnny's Simple Algorithm

You start with a $M \times N$ grid, where M is the number of rows and N is the number of columns of the grid. Initially, no two cells of the grid are connected to each other, so every cell is surrounded by walls on all four sides. The walls consist of an underscore ('_') for a horizontal wall, and a vertical bar ('|') for a vertical one. For example, if $M = 3$ and $N = 4$, the grid looks like this:

```
 _ _ _ _
|_|_|_|_|
|_|_|_|_|
|_|_|_|_|
```

Every cell of the grid has unique coordinates (p, q). The lower left corner in the example is $(1, 1)$, the upper right corner is $(3, 4)$.

After choosing the dimensions of the maze, you choose a starting cell. From now on you keep track of a list of *pending* cells, which initially contains only one cell (the starting cell), and you repeat the following steps:

1. If the list is empty, you stop. The maze is ready.
2. Else, you consider the most recently added cell in the list (call this cell AC). If this cell (at the end of the list) has no *unvisited* neighbor cells then you remove this cell from the list. Every cell has at most 4 neighbor cells: on the right, left, above and below. A cell is unvisited if it has never been added to the list.
3. If AC has at least one unvisited neighbor cell, you choose one of the unvisited neighbor cells (call this cell NC), remove the wall between AC and NC and add NC to the end of the list.

Johnny makes a nice little program using this algorithm and it works fine, but Johnny is not completely satisfied with the results. He is a demanding little boy and in his opinion the so-called *branching factor* of the maze is too low, i.e. the generated mazes contain very long paths and too few crossings. Therefore, the mazes are still too easy to solve for him.

A little trick can be applied to Johnny's Simple Algorithm, giving much better results. Johnny does not know it, but you will, since it will be explained below!

The idea behind the trick is to sometimes change the order of the cells in the list. This avoids long paths and results in more branches. Changing the order of the cells in the list is done by 'flipping' part of the list. A *flip* can be specified by giving the position of a cell in the list (where the first cell has position 1) and consists of reversing the sub-list starting at the specified cell and ending with the last cell in the list.

For example, if the list consists of the following cells:

```
(1,1) (1,2) (2,2) (3,2) (3,3)
```

then a flip with starting cell number 2 results in:

```
(1,1) (3,3) (3,2) (2,2) (1,2)
```

Now, we will reveal 'Johnny's Advanced Algorithm.'

Johnny's Advanced Algorithm

The algorithm is pretty much the same as 'Johnny's Simple Algorithm,' only sometimes part of the list is flipped. The steps you repeat after choosing the dimensions of the maze, choosing the starting cell and adding this cell to the list are:

1. If the list of cells is empty, you stop. The maze is ready.
2. Else you consider the last cell in the list. If this cell has no unvisited neighbor cells, then you remove this cell from the list.
3. Otherwise, you read a command. If this command is:

   'F *n*' you flip the list, starting at position *n*.

   'U' you go up: you remove the wall between the last cell in the list and the cell above it. The cell above the last cell in the list is added to the list.

   'D' you go down.

   'L' you go left.

   'R' you go right.

Since you are taking part in a programming contest, we ask you to write a program generating nice mazes for Johnny, using 'Johnny's Advanced Algorithm,' to make him happy again. The maximum size of a maze is 39×39.

Input Specification

The first line of the input contains the number of test cases. The input for every test case is divided into three parts:

- The first line contains two integer values M and N, specifying the dimensions of the maze: the number of rows M followed by the number of columns N.
- The second line contains the coordinates of the starting point (again, row followed by column).
- The next lines each contain a command. A command is one of the upper case characters 'F', 'U', 'D', 'L,' and 'R', appearing at the start of a line. An 'F' character is followed by a space and an integer (the starting position of the flip.)

The input for each test case contains exactly the number of commands needed for that maze.

Output Specification

The resulting mazes should be printed using spaces (' '), underscores ('_'), vertical bars ('|') and end-of-line characters. No unnecessary whitespace should be printed. The mazes should be followed by one blank line.

Example Input

```
2
3 3
1 1
U
U
R
D
D
R
U
U
3 4
2 1
R
U
L
F 2
R
U
R
D
D
F 4
D
L
L
```

Example Output

```
 _ _ _
|   | |
| | | |
|_|_ _|

 _ _ _ _
|_  |   |
|_ _   | |
|_ _ _|_|
```

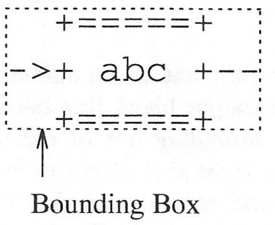
1995-1996 ACM International Collegiate Programming Contest
Western European Regional

Problem H

Syntrax

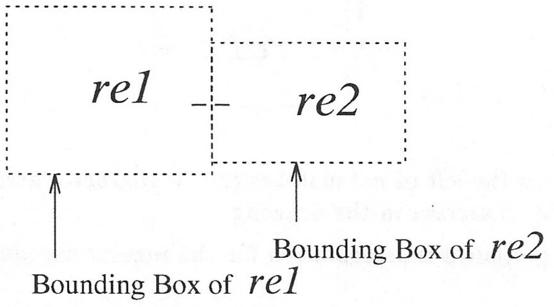
Given a simple syntax for describing regular expressions, one can find a graphical representation for a given regular expression using ASCII characters like '-', ' ', '+', and '/'. The syntax we use can be drawn by using four different patterns:

1. "abc" is the terminal string abc, represented as



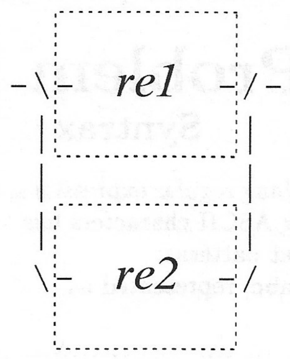
Note that the graphical representation of every expression has a *bounding box*. This is the smallest rectangle that surrounds the graphic.

2. (*re1 re2*) is a sequence of first expression *re1*, then expression *re2*, represented as



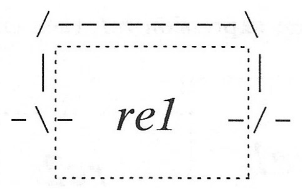
The two expressions *re1* and *re2* have to be concatenated such that the bounding boxes of the two expressions touch and such that the '-' on the right of *re1* matches the '-' on the left of *re2*.

3. {*re1 re2*} represents alternatives, either *re1* or *re2*, represented as



The number of '|' characters that has to be added depends on the shapes of *re1* and *re2*. There has to be exactly one straight blank line between the bounding box of the graphical representation of *re1* and the bounding box of *re2*. If necessary, also a number of '-' characters has to be added on the right side of *re1* or *re2*, to make the drawing possible. Note that the '-' on the left of *re1* and *re2* matches the '\' character and that the '-' on the right of *re1* and *re2* matches the '/' character in the drawing.

4. [*re1*] is a 1-or-more repetition of *re1*, represented as



Note that the '-' on the left of *re1* matches the '\' character and that the '-' on the right of *re1* matches the '/' character in the drawing.

For example the graphical representation for the regular expression `{("f" "bar") ["c"]}` looks like:

```
      +===+    +=====+
---\->+ f +--->+ bar +--/-->
   |  +===+    +=====+  |
   |                    |
   | /---------\        |
   | |  +===+  |        |
   \-\->+ c +--/--------/
        +===+
```

Write a program that reads syntax rules and prints the size of the graphical representation. For esthetic reasons, the entire graphic has a '--' on the left and a '->' on the right.

Input Specification

The input consists of a line holding the number of test cases, followed by the input expressions (one per line). The expressions are formatted according to the following grammar:

expression :: sequence | alternatives | repetition | terminal

sequence :: (ws expression expression) ws

alternatives :: { ws expression expression } ws

repetition :: [ws expression] ws

terminal :: " character* " ws

ws :: (< *space* > | < *tab* >)*

character :: <*any character except " and control-characters (ASCII 0..31)*>

Note that the 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: zero or more occurrences of x
$<\ >$	used for describing a character

Output Specification

For each expression, output a line of the form XxY with X and Y the width and height of the graphical representation of that expression.

Example Input

```
1
{("f" "bar") ["c"]}
```

Example Output

```
28x8
```