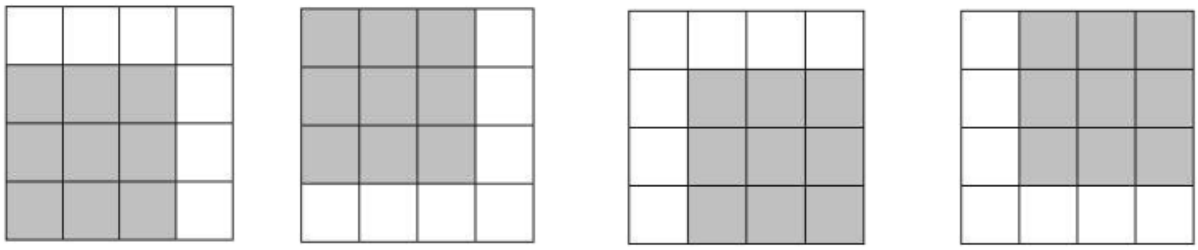
1. *Cutting Square Board*. Given a square board of size ***m*** by ***m***, which is marked with grid lines dividing the board into ***m*** x ***m*** one-square units (see below for an example), you are to find how many ways that you will be able to cut a square (of side length ***n***) from the board along the grid lines. You may assume that ***m*** and ***n*** are positive integers. In the following example, we have **4** different ways of cutting a 3x3 square from a 4x4 board.



Input: the first line gives an integer ***k*** which is the number of test cases and is followed by ***k*** lines of inputs, each of which consists of two integers for a test case, ***m*** and ***n*** as defined above (separated with a whitespace).

- Example:

  3
  4 3
  15 13
  34 35

Output: ***k*** lines of output, each of which is a single integer (for each test case) representing the number of ways in cutting the square as described above.

- Example (the output for the above inputs):

  4
  9
  0

2. *Clock*. Given the time on a 24 hour clock, convert it to the time on a 12 hour clock (AM/PM) format.

Input: the first line gives an integer ***k*** which is the number of test cases and is followed by ***k*** lines of inputs, each of which contains the time on a 24 hour clock, in the format HH:MM, where 00<=HH<=23 and 00<=MM<=59.

- Example:

  3
  05:05
  12:51
  16:13

Output: ***k*** lines of output, each of which is a single line containing the time in a 12 hour clock, in the format HH:MM XM, where 0<=HH<=12, 00<=MM<=59, and X is either 'A" or 'P'.

- Example (the output for the above inputs—note: there is a single white space before "AM" or "PM"):

  5:05 AM
  12:51 PM
  4:13 PM

3. *CheckDigit*. Credit card numbers are usually encoded with a "Check Digit" and there are many algorithms for calculating check digits. One of them is called the Luhn Formula for validating credit card numbers as shown below:

1. Double the value of alternate digits of the card number beginning with the second digit from the right (the rightmost digit is the check digit).
2. Add the individual digits comprising the products obtained in Step 1 to each of the unaffected digits in the original number.
3. The total obtained in Step 2 must be a number ending zero (20, 30, 40, etc.) for the account number to be validated.

For example, to validate the credit card number 49927398716:

Step 1:
```
4  9  9  2  7  3  9  8  7  1  6
  x2    x2    x2    x2    x2
---------------------------------
  18     4     6    16     2
```
Step 2: 4 +(1+8)+ 9 + (4) + 7 + (6) + 9 +(1+6) + 7 + (2) + 6

Step 3: Sum = 70 : Card number is valid because the 70/10 yields no remainder.

You are to write a Java program to determine if the card numbers are valid or invalid.

Input: the first line gives an integer ***k*** which is the number of test cases and is followed by ***k*** lines of inputs, each of which is a single positive integer for the card number (you may assume that the number will have less than 20 digits).

- Example:

```
4
1234
49927398716
513467882134
432876126
```

Output: ***k*** lines of output, each of which is a single output line, consisting of a word "VALID" if the card number is valid or a word "INVALID" followed by a single white space and the correct check digit, which is the right-most digit, that would make the card number valid.

- Example (the output for the above inputs):

```
INVALID 0
VALID
INVALID 2
VALID
```

4. *BinaryWeight*. The binary weight of a number is the count of 1s in the number's binary representation. For example, 43 in binary is 101011, so the binary weight is 4. Given a decimal number ***m***, you are to find the *next greater decimal number*, ***n***, that has the same binary weight. In this example, 45 (101101) is such a number.

Hints: a binary representation of a number is the sum of powers of 2. Number 43 is $1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$**,** which evaluates to 1*32 + 0*16 + 1*8 + 0*4 + 1*2 + 1*2 = 32 + 8 + 2 + 1 = 43 (i.e., for its binary format, 101011). The following gives a series of (***m***→***n***) pairs: 0001→0010 (1→2), 0110→1001 (6→9), 1011→1101 (11→13), …, etc.

Input: the first line gives an integer ***k*** which is the number of test cases and is followed by ***k*** lines of inputs, each of which is a positive integer, ***m***, as defined above. You may assume that 1 <= ***m*** <= 1 000 000 000 (thus you will need to use long rather than int datatype. Class Scanner has a method called *nextLong*()).

- Example:

```
5
4
10
255
32
1234567
```

Output: ***k*** lines of output, each of which is an integer ***n*** as defined above.

- Example (the output for the above inputs):

```
8
12
383
64
1234571
```

5. *Cogwheels*. A cogwheel is a toothed wheel and cogwheels are connected together when assembling machinery (see below). A rotating cogwheel spins all of the directly connected cogwheels in an opposite direction. However, too many cogwheels can lock up. For example, three cogwheels that are all connected will not be able to move. Your task is to write a Java program to describe the directions of rotation of the 2nd and 3rd cogwheels for a cogwheel system that has a least ***n*** unique cogwheels. You may assume that the system has at least three unique cogwheels, that is, the 1st, 2nd, and 3rd wheels and that the 1st cog is pushed in a *clockwise* direction and this is the only source of power.

Input: the first line gives an integer ***k*** which is the number of test cases and is followed by ***k*** data sets for the test cases. Each data set is defined in the following format:

1. The first line contains two integers, ***n*** as defined above (the number of cogwheels) and ***m*** indicating the number of connections in the cogwheel system; ***m*** and ***n*** are separated by a single white space; 3 <= ***n*** <= 100; 3 <= ***m*** <= 100.
2. The next ***m*** lines have a pair of integers separated by a single space—that describe a connection between the cogwheels.

- Example:

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

Output: ***k*** lines of output, each of which is a string of two characters, describing the directions of rotation of the 2nd and 3rd cogs (characters ‘A’ for clockwise, ‘B’ for counter-clockwise, and ‘L’ if a cog is locked.

- Example (the output for the above inputs):

```
LL
BA
```

6. *MaxValues*. You are passing through a 8-by-8 grid consisting of periods (i.e., ".") indicating empty spaces), hashes (i.e., "#") indicating walls, and digits (i.e., 1, 2, …, 8, 9) indicating the values. You enter the grid field from its bottom leftmost corner and make your way to the top rightmost corner. The only valid moves are up or to the right; there is no backtracking. You are to find a path such that the sum of the values on the path will be the maximum.

Input: the first line gives an integer ***k,*** which is the number of test cases (i.e., the number of 8-by-8 grids) and is followed by ***k*** data sets for the test-case-inputs. Each data set is defined in the following format:

- An 8-by-8 grid defined as above (i.e., 8 lines of strings, each of which consisting of 8 characters).
- A separating line of dashes

Example:

```
2
..2.....
..2.....
..2.....
.....9..
...##...
...5#...
........
.3......
--------
........
......6.
#.....6.
.....#..
......9.
.......7
...98...
...42...
--------
```

Output: ***k*** lines of output, each of which is a single integer (for each test case) representing the maximum value for the test case defined above.

- Example (the output for the above inputs):

```
12
42
```