ACM International Collegiate Programming Contest
2013 East Central Regional Contest
Grand Valley State University
University of Cincinnati
University of Windsor
Youngstown State University
November 9, 2013

Sponsored by IBM

Rules:

1. There are **nine** problems to be completed in **five hours**.
2. All questions require you to read the test data from standard input and write results to standard output. You cannot use files for input or output. Additional input and output specifications can be found in the General Information Sheet.
3. When displaying results, follow the format in the Sample Output for each problem. Unless otherwise stated, all whitespace in the Sample Output consists of exactly one blank character.
4. The allowed programming languages are C, C++ and Java.
5. All programs will be re-compiled prior to testing with the judges' data.
6. Non-standard libraries cannot be used in your solutions. The Standard Template Library (STL) and C++ string libraries are allowed. The standard Java API is available, except for those packages that are deemed dangerous by contestant officials (e.g., that might generate a security violation).
7. The input to all problems will consist of multiple test cases.
8. Programming style is not considered in this contest. You are free to code in whatever style you prefer. Documentation is not required.
9. All communication with the judges will be handled by the PC^2 environment.
10. Judges' decisions are to be considered final. No cheating will be tolerated.

Problem A: Battleships

You are all probably familiar with the pen and paper version of Battleships, but humor us while we describe it: Each puzzle consists of a 10-by-10 grid of squares in which 10 different ships have been secretly placed. One of these ships is 4 grid squares long, two of them are 3 grid squares long, three of them are 2 grid squares long, and the remaining four ships are each 1 grid square in length. No two ships can overlap and none can be adjacent to another, even diagonally. The only clues you have to the ship locations are a set of row and column sums printed on the left and bottom of the grid. Each row/column sum specifies how many grid squares in that row/column are occupied by ships. The figures below show an example of a Battleships puzzle and its solution.

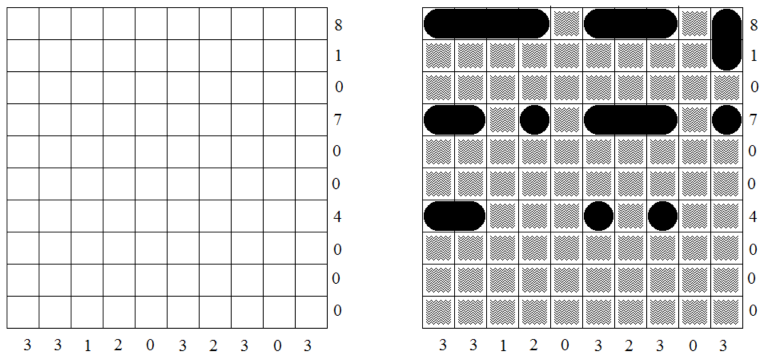


Figure 1

When designing a Battleships puzzle, you must make sure that there is a unique solution for the given row and column sums. Sometimes the sums on the sides are all that are needed to ensure a unique solution (like the example above), but sometimes you must specify one or more squares in the grid in order to rule out all but one solution (as in the example below).

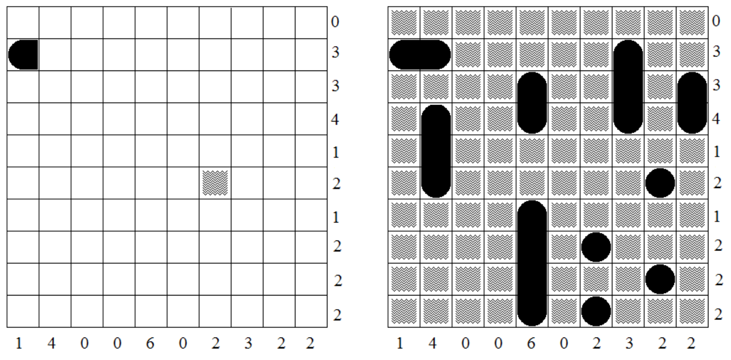


Figure 2

These specified squares can be one of seven types: water, interior of a ship, left end of a horizontal ship, top end of a vertical ship, right end of a horizontal ship, bottom end of a vertical ship, and 1-square ship. These are shown at the top of the next page. The characters beneath each square will be used for your output. Note that the last character is an upper-case letter 'O', not a zero; also note that the 'X' is upper-case but the 'w' and 'v' are lower-case.

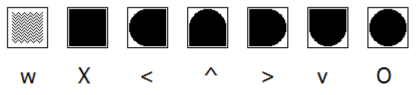


Figure 3

Your job is the following: given a set of row and column sums, you are to find the minimum number of specified squares needed to ensure that there is a unique solution. If this number is greater than 2, you are to reject the puzzle as being too ambiguous.

Input

The input file will start with an integer n indicating the number of test cases. Each test case consists of two lines, the first containing the ten row sums and the second containing the ten column sums.

Output

For each test case, output the case number followed by the total number of solutions possible for the given row and column sums (this number will never exceed 12,000) followed by the minimum number of specified squares needed to force a unique solution. If this number is greater than 2, display the phrase

```
too ambiguous
```

Otherwise, output the location of the "best" square(s) that can be used to force a unique solution followed by the type of square to use, using the characters shown in Figure 3. In the case of a single square, the "best" square is the square with the lowest row number, breaking any ties by choosing the one with the lowest column number. After this, if there is a choice between the type of square to use, use the lexicographically first one, where the lexicographic ordering is shown in Figure 3, with "water" being the lexicographically first square type, followed by "interior of ship", etc. In the case of two squares, output the pair whose first square is the best among all first squares, breaking ties by choosing the one with the best second square. All row and column numbers start at 1.

Sample Input

```
4
8 1 0 7 0 0 4 0 0 0
3 3 1 2 0 3 2 3 0 3
0 3 3 4 1 2 1 2 2 2
1 4 0 0 6 0 2 3 2 2
1 2 1 2 5 0 1 2 1 5
2 0 0 1 1 2 3 3 2 6
0 4 1 0 4 0 4 4 0 3
3 1 0 2 3 3 3 0 2 3
```

Sample Output

```
Case 1: 1 0
Case 2: 993 2 (2,1)=< (6,7)=w
Case 3: 8 1 (2,5)=O
Case 4: 30 too ambiguous
```

Problem B: Cuckoo for Hashing

An integer hash table is a data structure that supports insert, delete and lookup of integer values in constant time. Traditional hash structures consist of an array (the *hash table*) of some size n, and a *hash function* $f(x)$ which is typically $f(x) = x \mod n$. To insert a value x into the table, you compute its *hash value* $f(x)$ which serves as an index into the hash table for the location to store x. For example, if $x = 1234$ and the hash table has size 101, then 1234 would be stored in location $22 = 1234 \mod 101$. Of course, it's possible that some other value is already stored in location 22 ($x = 22$ for example), which leads to a *collision*. Collisions can be handled in a variety of ways which you can discuss with your faculty advisor on the way home from the contest.

Cuckoo hashing is a form of hashing that employs two hash tables T_1 and T_2, each with its own hash function $f_1(x)$ and $f_2(x)$. Insertion of a value x proceeds as follows: you first try to store x in T_1 using $f_1(x)$. If that location is empty, then simply store x there and you're done. Otherwise there is a collision which must be handled. Let y be the value currently in that location. You replace y with x in T_1, and then try to store y in T_2 using $f_2(y)$. Again, if this location is empty, you store y there and you're done. Otherwise, replace the value there (call it z) with y, and now try to store z back in T_1 using $f_1(z)$, and so on. This continues, bouncing back and forth between the two tables until either you find an empty location, or until a certain number of swaps have occurred, at which point you *rehash* both tables (again, something to discuss with your faculty advisor). For the purposes of this problem, this latter occurrence will never happen, i.e., the process should always continue until an empty location is found, which will be guaranteed to happen for each inserted value.

Given the size of the two tables and a series of insertions, your job is to determine what is stored in each of the tables.

(For those interested, cuckoo hashing gets its name from the behavior of the cuckoo bird, which is known to fly to other bird's nests and lay its own eggs in it alongside the eggs already there. When the larger cuckoo chick hatches, it pushes the other chicks out of the nest, thus getting all the food for itself. Gruesome but efficient.)

Input

Input for each test case starts with 3 positive integers n_1 n_2 m, where n_1 and n_2 are the sizes of the tables T_1 and T_2 (with $n_1, n_2 \leq 1000$ and $n_1 \neq n_2$) and m is the number of inserts. Following this will be m integer values which are the values to be inserted into the tables. All of these values will be non-negative. Each table is initially empty, and table T_i uses the hash function $f_i(x) = x \mod n_i$. A line containing 3 zeros will terminate input.

Output

For each test case, output the non-empty locations in T_1 followed by the non-empty locations in T_2. Use one line for each such location and the form `i:v`, where `i` is the index location of the table, and `v` is the value stored there. Output values in each table from lowest index to highest. If either table is empty, output nothing for that table.

Sample Input

```
5 7 4
8 18 29 4
6 7 4
8 18 29 4
1000 999 2
1000
2000
0 0 0
```

Sample Output

```
Case 1:
Table 1
3:8
4:4
Table 2
1:29
4:18
Case 2:
Table 1
0:18
2:8
4:4
5:29
Case 3:
Table 1
0:2000
Table 2
1:1000
```

Problem C: Playing Fair with Cryptography

Encryption is the process of taking *plaintext* and producing the corresponding *ciphertext*. One method to do this is the Playfair cipher. This cipher was invented by (who else) Charles Wheatstone in the 1850's (but got its name from one of its most ardent promoters, the Scottish scientist Lyon Playfair). Unlike many substitution ciphers, the Playfair cipher does not encrypt single letters at a time, but groups of two letters called *digraphs*. The encryption process uses a 5×5 grid generated by a secret key known only to the two parties using the cipher (hopefully). The grid is generated as follows: You write the letters of the key, one letter per square row-wise in the grid starting at the top. Any repeated letters in the key are skipped. After this is done, the remaining unused letters in the alphabet are placed in order in the remaining squares, with I and J sharing the same square. For example, if the key was "ECNA PROGRAMMING CONTEST", the generated square would look like the following:

E	C	N	A	P
R	O	G	M	I/J
T	S	B	D	F
H	K	L	Q	U
V	W	X	Y	Z

You encrypt a digraph using the following rules:

1. If both letters are in the same row, replace each with the letter to its immediate right (wrapping around at the end). For example, using the above grid the plaintext digraph DS is encrypted as FB, and AP is encrypted as PE.

2. If both letters are in the same column, replace each with the letter immediately below it (again wrapping around at the bottom). For example, PF is encrypted as IU (or JU), and WO is encrypted as CS.

3. Otherwise, "slide" the first character across its row until it is in the same column as the second character, and do the same for the second character. The two characters you end up on are the encrypted characters of the digraph. If you view the original digraph as corners of a rectangle in the grid, you are replacing them with the characters in the other two corners. For example, TU is encrypted as FH, UT is encrypted as HF, and ZC is encrypted as WP.

What happens when the digraph contains the same letter twice? In the original Playfair cipher you insert the letter 'X' between the two letters and continue encrypting. You also use an extra 'X' at the end of the plaintext if you have an odd number of letters. Thus the plaintext "OOPS" would first be changed to the digraphs "OX", "OP" and "SX" and then would be encrypted as "GWICBW" (using the grid above). Note that the plaintext "POOS" would not need any added letters since the two 'O's do not appear in the same digraph.

We'll modify the Playfair cipher in one simple way: Instead of always using 'X' as the inserted letter, use 'A' the first time an insertion is needed, then 'B' the next time, and so on (though you should never use 'J' as an inserted letter; just go from 'I' to 'K'). Once you hit 'Z', you go back to using 'A'. If the inserted letter would result in a digraph with the same two letters, you just skip to the next inserted letter. For example, "OOPS" would now become the digraph stream "OA", "OP", "SB" before being encrypted, and the plaintext "AABCC" would become the digraph stream "AB", "AB", "CD", "CE".

Given a key and plaintext, you are to generate the corresponding ciphertext.

Input

The input file will start with an integer n indicating the number of problem instances. Each instance consists of two lines, the first containing the key and the second the plaintext to encrypt. Both of these may contain upper- and lower-case letters, as well as spaces and non-alphabetic characters (both of which should be ignored). For simplicity, the letters 'j' and 'J' will never appear in any key or plaintext.

Output

For each problem instance, output the case number followed by the corresponding ciphertext using upper-case letters with no space. Always use the letter 'I' instead of 'J' in the ciphertext.

Sample Input

```
1
ECNA Programming Contest 2013
This is the easy problem!
```

Sample Output

```
Case 1: HVOFOFHVCPCPDWEIGSHNGD
```

Problem D: Rent-A-Pixel

Harriet T. Emmel is selling “real estate” on her web site in the form of 10-by-10 square blocks of pixels. She has divided her web page into a rectangular grid, with grid lines 10 pixels apart. Anyone may rent 10-pixel-by-10-pixel blocks in this grid for posting artwork, advertising, or anything else.

Harriet expected that most customers would want to purchase rectangular regions of blocks, but a few want to be more creative. For instance, an optician wanted to purchase blocks in the shape of a pair of eyeglasses and a bow-and-arrow company wanted to rent space in the shape of a bullseye (in the following, each square is a 10-pixel-by-10-pixel block):

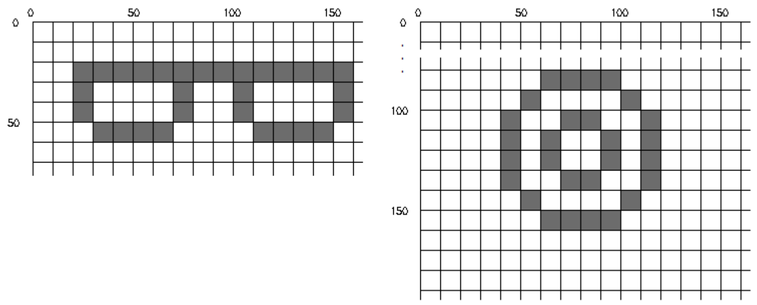


Figure 1

Harriet has decided that, in the interests of simplicity, each purchase must be a single “orthogonally convex” region of blocks. This simply means that any row or column of pixels, when intersected with the region, must consist of either zero or one connected segments. For the two examples above, the smallest orthogonally convex regions containing the desired blocks are:

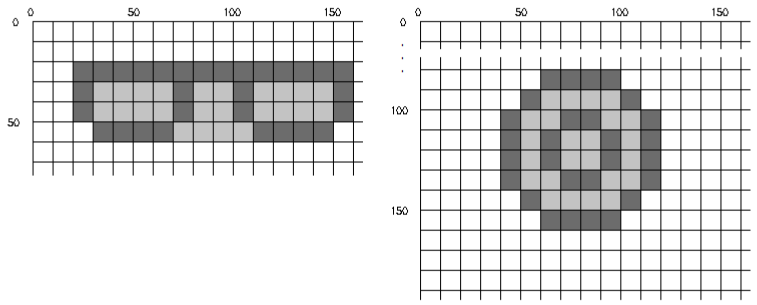


Figure 2

As a service to her customers, H. T. Emmel lets them choose any set of blocks, then she calculates the smallest orthogonally convex region containing them. Although, in general, this calculated region might be disconnected, we assume, for simplicity, that it is connected, i.e., that there is an orthogonal path of pixels connecting any pair of pixels in the region. Write the program that does this smallest region calculation.

Input

Each test case will consist of a line containing a positive integer n, $n \leq 10000$, indicating the number of 10-pixel-by-10-pixel blocks requested by the user, followed by one or more lines containing a total of $2n$ integers $r_0\ c_0\ r_1\ c_1 \ldots r_{n-1}\ c_{n-1}$, $0 \leq r_i, c_i \leq 10^9$. Each $r_i\ c_i$ pair gives the row and column number of the upper left pixel of one of these blocks. All of these coordinates will be multiples of 10 and no coordinate pair will be repeated withing a test case. Each test case is guaranteed to be covered by a single, connected, minimal, orthogonally convex polygon. A line containing a single 0 will terminate input.

Output

For each test case, print the case number followed by the (row, column) pixel coordinates of the vertices of the smallest orthogonally convex polygon containing the blocks described by the input. The first coordinate should be for the block with the lowest row number and, among those, the lowest column number. The coordinates should describe a clockwise traversal of the polygon.

Sample Input

```
30
20 20 20 30 20 40 20 50 20 60 20 70 20 80 20 90 20 100
20 110 20 120 20 130 20 140 20 150
30 20 30 70 30 100 30 150
40 20 40 70 40 100 40 150
50 30 50 40 50 50 50 60 50 110 50 120 50 130 50 140
28
80 60 80 70 80 80 80 90
90 50 90 100
100 40 100 70 100 80 100 110
110 40 110 60 110 90 110 110
120 40 120 60 120 90 120 110
130 40 130 70 130 80 130 110
140 50 140 100
150 60 150 70 150 80 150 90
0
```

Sample Output

```
Case 1: 20 20 20 159 49 159 49 149 59 149 59 30 49 30 49 20
Case 2: 80 60 80 99 90 99 90 109 100 109 100 119 139 119 139 109 149 109 149 99 159 99
        159 60 149 60 149 50 139 50 139 40 100 40 100 50 90 50 90 60
```

(Note: Because of space limitation, the output for Case 2 is shown over multiple lines. In actuality, it would all be on a single line.)

Problem E: Stampede!

You have an $n \times n$ game board. Some squares contain obstacles, except the left- and right-most columns which are obstacle-free. The left-most column is filled with your n pieces, 1 per row. Your goal is to move all your pieces to the right-most column as quickly as possible. In a given turn, you can move each piece N, S, E, or W one space, or leave that piece in place. A piece cannot move onto a square containing an obstacle, nor may two pieces move to the same square on the same turn. All pieces move simultaneously, so one may move to a location currently occupied by another piece so long as that piece itself moves elsewhere at the same time.

Given n and the obstacles, determine the fewest number of turns needed to get all your pieces to the right-hand side of the board.

Input

Each test case starts with a positive integer n indicating the size of the game board, with $n \leq 25$. Following this will be n lines containing n characters each. If the j^{th} character in the i^{th} line is an `'X'`, then there is an obstacle in board location i, j; otherwise this character will be a `'.'` indicating no obstacle. There will never be an obstacle in the 0^{th} or $(n-1)^{st}$ column and there will always be at least one obstacle-free path between these two columns. A line containing a single 0 will terminate input.

Output

For each test case output the minimum number of turns to move all the pieces from the left side of the board to the right side.

Sample Input

```
5
.....
.X...
...X.
..X..
.....
5
.X...
.X...
.X...
.XXX.
.....
0
```

Sample Output

```
Case 1: 6
Case 2: 8
```

Problem F: Super Phyllis

Phyllis works for a large, multi-national corporation. She moves from department to department where her job is to uncover and remove any redundancies inherent in the day-to-day activities of the workers. And she is quite good at her job.

During her most recent assignment, she was given the following chart displaying the chain of command within the department:

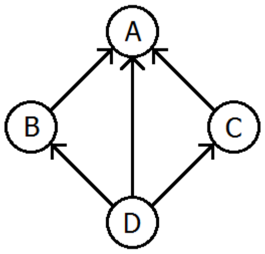


Figure 1

Whenever anyone needs to send a report to their bosses, they use the above chart, sending one report along each arrow. Phyllis realized almost instantly that there were redundancies here. Specifically, since D sends a report to B and B sends a report to A, there's really no need for D to send a report to A directly, since it will be summarized in the report B sends to A. Thus the connection from D to A can be removed. If there had also been a connection from C to B, then the connections from D to B and C to A could have been removed as well.

Phyllis would like your help with this. Given a description of a chart like the one above, she would like a program that identifies all connections that can be removed from the chart.

Input

The first line of each test case will contain a positive integer m indicating the number of connections in the chart. Following that will be m lines each containing two strings s_1 s_2 indicating that there is a connection from employee s_1 to his/her boss s_2. In any test case there will be no more than 200 employees listed and no connection will appear more than once. A line containing a single 0 will terminate the input.

Output

For each test case, output the number of connections that should be removed followed by a list of the deleted connections in lexicographic order. Connection s_1 s_2 should be represented by the string `s1,s2`.

Sample Input

```
5
D B
D C
D A
B A
C A
6
D B
D C
D A
C B
B A
C A
1
Danny Tessa
0
```

Sample Output

```
Case 1: 1 D,A
Case 2: 3 C,A D,A D,B
Case 3: 0
```

Problem G: Tree Lighting

Arbor Day is a big day for the Pine Family of Chestnut Grove. Each year the family, led by their dad Hickory, decorates their front yard and the front of their house with hundreds of Arbor Day decorations. At night, Hickory likes to shine a yard light onto the front of the house so that the passing onlookers can get a better look at all the displays. Unfortunately, several of the decorations block the light, making it difficult to shine a light on the entire house. This is mitigated a bit by the fact that some of the decorations act like mirrors and can reflect the light onto the house. The figure below shows an example: the light emanates from a point at the bottom of the figure, and is blocked by the horizontal decoration in the middle of the figure, but gets reflected by the other decoration on the right. As a result, only about 75% of the front of the house (at the top of the figure) gets illuminated.

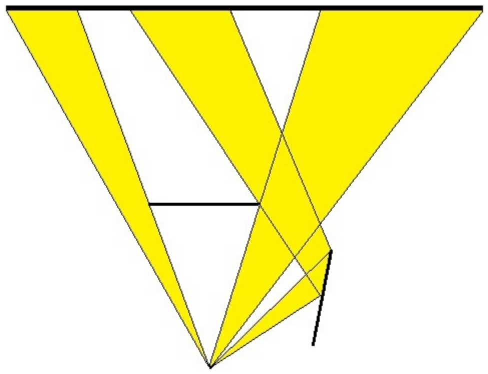


Figure 1

Since their Arbor Day decorations change from year to year, Hickory would like a general method to determine what percent of the front of his house will be lit given a layout of the decorations and whether they reflect or not.

Input

Each test case will start with a line containing three values: an integer n, a double ang and a double len. n specifies the number of decorations ($0 \leq n \leq 10$), and ang represents the spread of the light in degrees ($0 < ang \leq 150$). The light is always located at the origin, and the beam of light is symmetric about the positive y-axis, making an angle of $ang/2$ on either side. len specifies the maximum distance any light ray can travel (after this distance, the beam is diminished enough so that it does not contribute to the lighting of the house). The next n lines will each contain 5 integers x_1 y_1 x_2 y_2 r, where the first four values specify the endpoints of a decoration, and r will be either 0 for a non-reflective decoration or 1 for a reflective decoration. Assume all decorations have 0 thickness and that a reflective decoration is reflective on both sides. Following these n lines will be a single line containing 4 integers x_1 y x_2 y specifying the endpoints of the house front, with $y > 0$. None of the decorations will intersect with each other, the house front or the origin, and none will have y values greater than the y value for the house front. All coordinates will be between -10000 and 10000. For each test case, the placement of the decorations and the value of len will ensure that the total number of beam reflections is no more than 100. A line containing 0 0.0 0.0 will terminate input.

Output

For each test case, output the percentage of the house illuminated, rounded to the nearest hundredth.

Sample Input

```
0 90 100
-20 10 20 10
1 90 100
-2 3 4 8 0
-20 10 20 10
2 150 1000
-60 165 50 165 0
110 25 130 120 1
-205 360 275 360
0 0.0 0.0
```

Sample Output

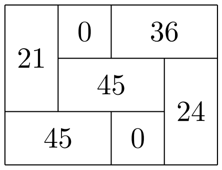
```
Case 1: 50.00
Case 2: 20.83
Case 3: 74.43
```

Problem H: The Urge to Merge

The Acme Consulting Group has sent you into a new technology park to enhance dynamism, synergy and sustainability. You're not sure what any of these terms mean, but you're pretty good at making money, which is what you plan on doing. The park consists of a $3 \times n$ grid of facilities. Each facility houses a start-up with an inherent value. By facilitating mergers between neighboring start-ups, you intend to increase their value, thereby allowing you to fulfill your life-long dream of opening your own chain of latte-and-burrito shops.

Due to anti-trust laws, any individual merger may only involve two start-ups and no start-up may be involved in more than one merger. Furthermore, two start-ups may only merge if they are housed in adjacent facilities (diagonal doesn't count). The added value generated by a merger is equal to the product of the values of the two start-ups involved. You may opt to not involve a given start-up in any merger, in which case no added value is generated. Your goal is to find a set of mergers with the largest total added value generated. For example, the startup values shown in the figure on the left, could be optimally merged as shown in the figure on the right for a total added value of 171.

7	2	4	9
3	5	9	3
9	5	1	8



Input

The first line of each test case will contain a single positive integer $n \leq 1000$ indicating the width of the facilities grid. This is followed by three lines, each containing n positive integers (all ≤ 100) representing the values of each start-up. A line containing a single 0 will terminate input.

Output

For each test case, output the maximum added value attainable via mergers for that set of start-ups.

Sample Input

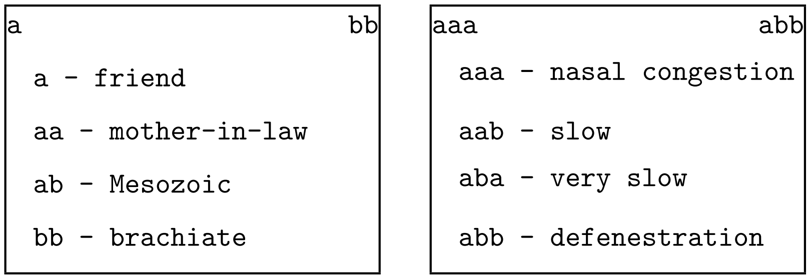
```
4
7 2 4 9
3 5 9 3
9 5 1 8
0
```

Sample Output

```
Case 1: 171
```

Problem I: Xenospeak

It's 2014, and Bob Roberts is an expert linguist and world renowned expert on the language of the alien M'ca. Their language is rather unusual in that all the words are made up of the letter combinations "a", "ab" and "bb" (we're using 'a's and 'b's here, since the actual M'ca characters are unprintable). Thus, some words in their language are aaabbbbb and aababb, but babb is not (you'd be a laughingstock in any M'ca establishment if you tried to used babb in a sentence). Not surprisingly, with such a small alphabet, every possible combination of "a", "ab" and "bb" form a legal M'ca word (up to a certain length, which is of no importance to this problem). Bob is creating a set of M'ca-to-English dictionaries of all legal M'ca words and is following the traditional word ordering of the M'ca: all 1 letter words are listed first (in alphabetical order), then all 2-letter words (in alphabetical order), and so on. The first two pages of one possible dictionary are shown below:



```
a                       bb    aaa                     abb
  a - friend                    aaa - nasal congestion
  aa - mother-in-law            aab - slow
  ab - Mesozoic                 aba - very slow
  bb - brachiate                abb - defenestration
```

Bob needs a little help. He intends for each page of any dictionary to contain the same number of M'ca words, but this number will vary in different editions depending on the page size, font size, etc. As with any dictionary, the first and last word of each page is printed at the top of the page to allow easier searching by users. Here's where you come in: given the number of words per page, and the page number, he would like a program to determine the two words printed at the top of that page.

Input

Input for each test case will consist of a single line containing two positive integers n m, where n is the number of words per page (≤ 30) and m is the page number ($m \leq 10^{18}$). A line containing 0 0 will terminate input.

Output

For each test case output the two words which would appear at the top of page m given that n words are printed on each page (including page m).

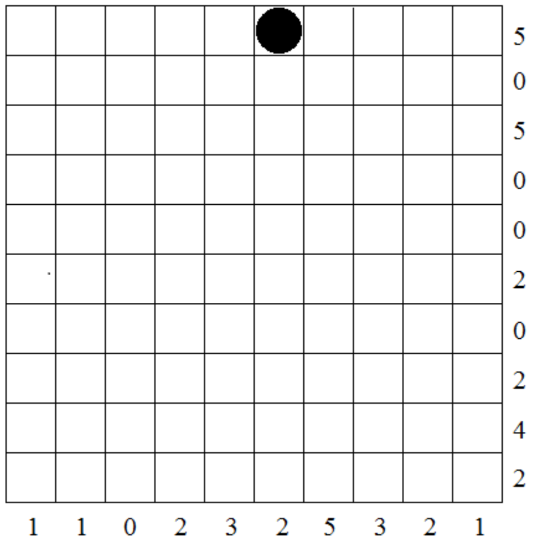
Sample Input

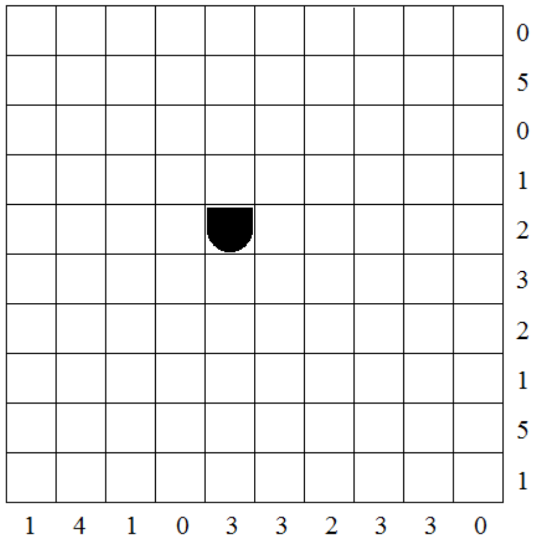
```
4 2
9 10
0 0
```

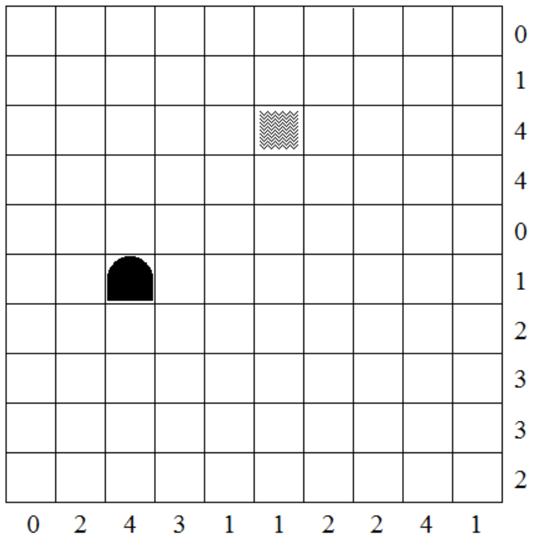
Sample Output

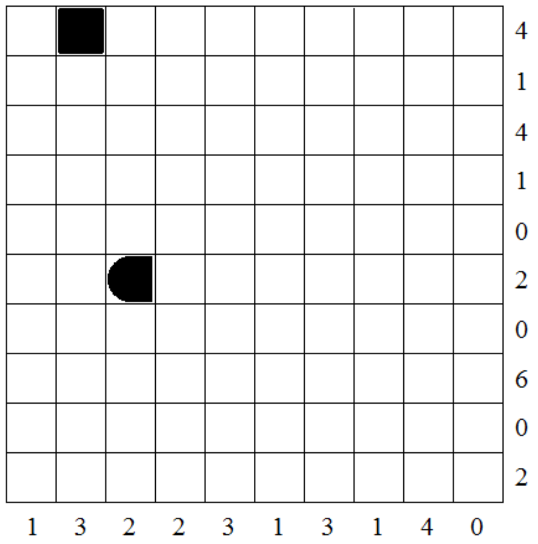
```
Case 1: aaa abb
Case 2: bbbbaa aaaabab
```

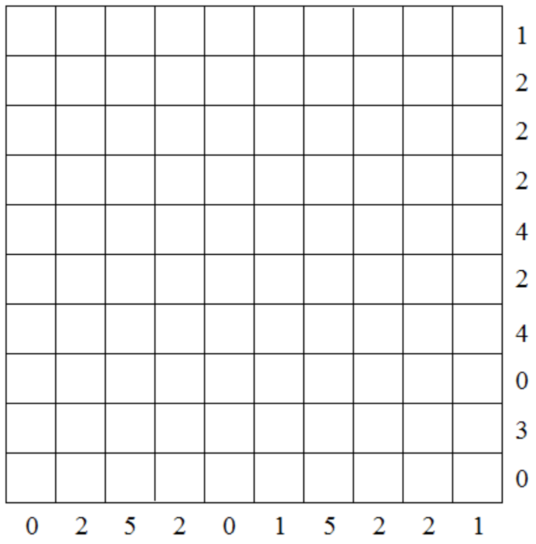
Appendix: More Battleships Puzzles For Your Free Time

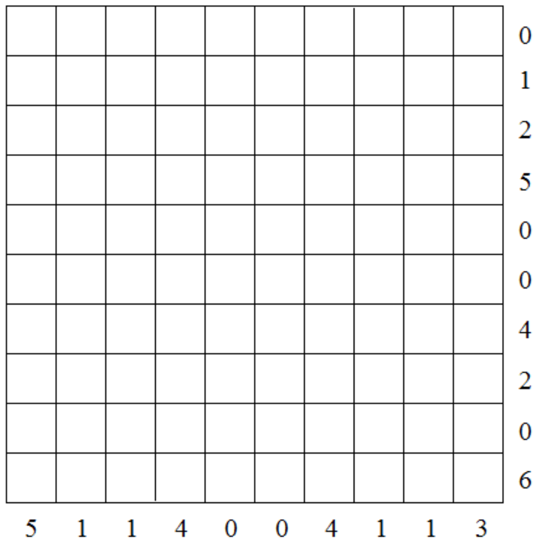

5 0 5 0 0 2 0 2 4 2
1 1 0 2 3 2 5 3 2 1


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


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


4 1 4 1 0 2 0 6 0 2
1 3 2 2 3 1 3 1 4 0


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


0 1 2 5 0 0 4 2 0 6
5 1 1 4 0 0 4 1 1 3