XXXVII Maratón Nacional de Programación Colombia - ACIS / REDIS / CCPL 2023 ICPC

Problems

(This set contains 10 problems; problem pages are numbered from 1 to 17)

General Information. Unless otherwise stated, the conditions stated below hold for all the problems. However, since some problems may have specific requirements, it is important to read the problem statements carefully.

Program name. Each source file (your solution!) must be called

`<codename>.c`, `<codename>.cpp`, `<codename>.java`, or `<codename>.py`

as instructed below each problem title.

Input.

1. The input must be read from the standard input.
2. In most problems, the input can contain several test cases. Each test case is described using a number of lines specific to the problem.
3. In most cases, when a line of input contains several values, they are separated by single blanks. No other spaces appear in the input and there are no empty lines.
4. Every line, including the last one, has the usual end-of-line mark.
5. If no end condition is given, then the end of input is indicated by the end of the input stream. There is no extra data after the test cases in the input.

Output.

1. The output must be written to the standard output.
2. The result of each test case must appear in the output using a number of lines, which depends on the problem.
3. When a line of results contains several values, they must be separated by single spaces. No other spaces should appear in the output. There should be no empty lines.
4. Every line, including the last one, must have the usual end-of-line mark.
5. After the output of all test cases, no extra data must be written to the output.
6. To output real numbers, if no particular instructions are given, round them to the closest rational with the required number of digits after the decimal point. Ties are resolved rounding to the nearest upper value.

A: ASP

Source file name: `asp.c`, `asp.cpp`, `asp.java`, *or* `asp.py`
Author: Rodrigo Cardoso

John is worried about the passwords he uses for his internet activities. He decided to build his passwords according to some rules that he considers safe enough. He calls each one of these passwords an *asp* (for *a safe password*).

Suppose that A is a given asp. A *subasp* of A is any word that corresponds to a sequence of two or more consecutive symbols in A.

John's conditions for an asp A are rather simple:

- A is a word built from an alphabet of N symbols.
- It is not possible to find a subasp of A more than once within it.
- No two consecutive symbols in A are the same.

For instance, if the given alphabet is $\{a, b, c, d\}$, it is possible to build asps like *abac*, *abcbdadb* and *bcbadb*. On the other hand, *baac*, *abcdcabcdb* are not asps.

John wants to know how long an asp could be, given the alphabet's length. For instance, if asps were made with only $\{a, b\}$, a longest asp could have at most three symbols, e.g., *aba*. You are asked to help John with his task.

Input

The input consists of several test cases. A case is defined with a line with a positive integer number N, $1 < N < 1000$, the number of symbols in the alphabet used to build asps. The end of the input is signaled with a line containing a single 0, which should be not processed.

The input must be read from standard input.

Output

For each test case, output a line with one integer value, corresponding to the length of an asp of maximal length that may be built with an alphabet of N symbols.

The output must be written to standard output.

Sample Input	Sample Output

`2`
`3`
`0`
	`3`
	`7`

B: **Be Strong**

Source file name: `be.c`, `be.cpp`, `be.java`, *or* `be.py`
Author: Camilo Rocha

A *prefix* of a string is a substring that occurs at the beginning. If a string has size $N \geq 0$, then it has exactly $N + 1$ prefixes (the empty string is a prefix of any string). Given a collection of strings, its *strong prefix* is the longest prefix common to all strings in the collection. Any difference between lowercase and uppercase characters is considered immaterial.

For example, `te` is the strong prefix for the collection

```
tequila        tEnnessee        Telephone        tetris
```

The empty string is the strong prefix for the collection

```
hello        world
```

In this problem, you are asked to compute the strong prefix of a collection of words.

Input

The input consists of several test cases. Each test case begins with a line containing a number M ($0 \leq M \leq 5\,000$) denoting the number for words in the collection. Then, M lines follow, each containing a string W ($1 \leq |W| \leq 200$) made from English lowercase and uppercase characters. The end of the input is given with $M = 0$, which should not be processed.

The input must be read from standard input.

Output

For each test case, output a single line with the strong prefix of the collection of M words where the case is immaterial. The output can only contain English lowercase characters. If the strong prefix is empty, output a star '*'.

The output must be written to standard output.

Sample Input	**Sample Output**

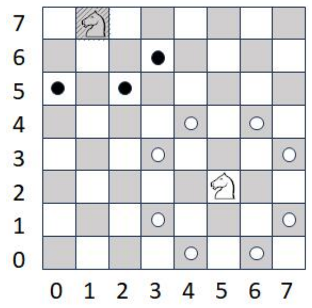
```
4
tequila
tEnnessee
Telephone
tetris
2
hello
world
2
De
dE
0
```
	```
	te
	*
	de
	```

C: **Knights**

Source file name: `knights.c`, `knights.cpp`, `knights.java`, *or* `knights.py`
Author: Rodrigo Cardoso

KA and *KB* are knights in a generalized chessboard with dimensions $n \times n$, with $n \geq 1$. Its rows and columns are numerated $0, 1, \ldots, n-1$: the square (i, j), with $0 \leq i, j < n$, is located at the i-th row and j-th column of the chessboard.

The two knights *KA* and *KB* are positioned in squares of the board, and their goal is to wander from their initial locations to meet each other. They follow the rules of knights in chess: they can move two squares vertically and one square horizontally, or two squares horizontally and one square vertically (i.e., forming the shape of a capital L), always within the chessboard boundaries. The following figure illustrates two examples of knight moves on a typical 8×8 chessboard:



They move alternately, one move each time, and *KA* moves first. If in *KB*'s k-movement (with $k \geq 0$), *KB* may reach the square *KA* is located at, it is said that they *meet* in k movements. Note that it could be that, depending on the initial positions, *KA* and *KB* cannot meet. For instance, if on an 8×8 chessboard the initial positions of *KA* and *KB* are $(0, 0)$ and $(6, 2)$, respectively, then they can meet in 2 movements with the following sequence of movements:

$$(2, 1) \quad (5, 4) \quad (4, 2) \quad (4, 2).$$

Your task is to write a program that, given the size of the chessboard and the initial positions of *KA* and *KB*, determines the minimum number of movements needed to meet or identifies if this is impossible.

Input

The input consists of several test cases. A case is defined with a line with 5 blank-separated integer values n, a, b, c, d, $1 < n < 300$ and $0 \leq a, b, c, d < n$, where n is the number of rows and columns of the chessboard, and (a, b) and (c, d) are the initial locations for *KA* and *KB*, respectively. The end of the input is given with $n = a = b = c = d = 0$, which should not be processed.

The input must be read from standard input.

Output

For each test case, output a single line: the minimum number of moves required for *KA* and *KB* to meet whenever this is possible, and '*' otherwise.

The output must be written to standard output.

Sample Input	**Sample Output**

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

D: **Robot Arm**

Source file name: `arm.c`, `arm.cpp`, `arm.java`, *or* `arm.py`
Author: Rafael García

Two-Dimensional Reacher (`RD2`) is a robotic arm (distant ancestor of the famous `R2D2`) that has been used as a welder. Its movements are restricted to two dimensions.

`RD2` can be described as a chain of N straight and rigid sections made of very hard metal, $N > 0$, and a joint between each pair of consecutive sections. The free end of the first section is pinned to a point that acts as a shoulder and the free end of the last section acts as a fingertip. The joints between sections allow the arm to be folded at any angle between 0 and 2π (even on itself). The same goes for the shoulder as it also allows the first section to be moved in any direction.

Assuming that the shoulder of `RD2` is pinned at the point $(0, 0)$ and that the N sections have lengths $l_1, \ldots, l_N$, the goal is to decide if each point $(x_1, y_1), \cdots, (x_m, y_m)$ in the plane can be touched by `RD2`'s fingertip. For example, if $N = 3$ and `RD2` has sections with lengths 2, 5, and 2, respectively, then `RD2` can touch the points $(4, 5)$ and $(2, 3)$; however, it cannot touch the points $(0, 0)$ and $(9, 1)$.

Input

The input consists of several test cases. A case begins with a line with two positive integers N and m where N is the number of sections and m the number of points to be touched ($1 \leq N \leq 10^3$ and $1 \leq m \leq 10^3$). A line with N integer positive numbers $l_1, \ldots, l_N$ follows, describing the lengths of the sections: l_1 is the length of the first section, l_2 the length of the second section, and so on ($1 \leq l_i \leq 10^4$). Each of the following m lines contains two blank-separated integer numbers x_i and y_i, defining the coordinates of the ith point to be touched ($-10^7 - 1 \leq x_i, y_i \leq 10^7 + 1$).

The end of the input is denoted with a line that contains two blank-separated zeroes, which should not be processed.

The input must be read from standard input.

Output

For each case, output m lines, one per point: the i-th line contains '`Y`' if `RD2` can touch the point (x_i, y_i) and '`N`' otherwise.

The output must be written to standard output.

Sample Input	**Sample Output**

`3 4`
`2 5 2`
`4 5`
`2 3`
`0 0`
`9 -1`
`1 2`
`10000`
`1 800`
`10000 0`
`0 0`
	`Y`
	`Y`
	`N`
	`N`
	`N`
	`Y`

E: **LISP Extravaganza**

Source file name: `extravaganza.c`, `extravaganza.cpp`, `extravaganza.java`, *or* `extravaganza.py`
Author: Camilo Rocha

LISP (an acronym for LISt Processing) is a family of programming languages with a long history and a distinctive, fully parenthesized notation. It is a favored programming language for artificial intelligence (AI) research, and today some of its general-purpose dialects include Common Lisp, Scheme, and Clojure.

In LISP, all program code is written as parenthesized lists. Although its syntax is simple and consistent, during the years it has been given nicknames such as *Lots of Irritating Superfluous Parentheses* and the likes. In this problem, you will have to deal with strings that are made of parentheses (i.e., ‘`(`’ and ‘`)`’), such as in LISP.

A string s of parentheses is *balanced* if s satisfies one of the following conditions:

- it is the empty string;
- it is the string (t) and t is a balanced string; or
- it is the string tu for some balanced strings t and u.

For instance, the strings `()` and `((())())()(())` are balanced, while `())()` and `((())()` are not.

Deciding if a given string is balanced is quite simple for experienced programmers such as yourself. Hence, the problem to solve here is a little different: given a string s, can you compute the length of a longest balanced subsequence of s? In the case of `()`, `((())())()(())`, `())()`, and `((())()` the answers are 2, 14, 4, and 6, respectively. In the first two cases the sequences are balanced; in the third case `()()` and the fourth case `(())()` are the longest balanced subsequences.

Input

The input consists of several test cases. The first line of the input contains a number $m \geq 0$ indicating the number of test cases; then m lines follow. Each test case is given in a single line comprising a natural number n and a string s ($1 \leq n \leq 50\,000$), separated by a blank, where n is the length of s and s contains parentheses only.

The input must be read from standard input.

Output

For each test case, in a single line, output the length of a longest balanced subsequence of s.

The output must be written to standard output.

Sample Input	**Sample Output**

```
4
2 ()
14 ((())())()(())
5 ())()
7 ((())()
```
	```
	2
	14
	4
	6
	```

F: **Finding Common Passwords**

Source file name: `find.c`, `find.cpp`, `find.java`, *or* `find.py`
Author: Andrés Mejía

As a Security Engineer, one of your main responsibilities is to ensure that employees in the company use passwords that are uncommon and hard to guess. Unfortunately, you suspect that your colleagues are not adhering to this idea: several email accounts, with confidental company information, have been recently compromised by a hacking group known as the International Common Password Connoisseurs (ICPC).

To protect against future attacks, you started an audit of an anonymized sample of passwords, and have discovered that people frequently include common words like `admin` or `secret` in their passwords. It turns out, for instance, that `qwerty` appeared in at least 60% of the passwords! You have decided to systematically look for long strings appearing often in the list of passwords. Specifically, given a list of N passwords and a positive integer K, you are looking for the longest string in the N passwords appearing at least in K of them.

For instance, consider the following list of $N = 5$ passwords:

```
monkey   monk   money   motorcycle   recycle
```

In the case $K = 2$, the longest such (sub)string is `cycle`. In the case $K = 3$, the longest substring is `mon`. However, in the case $K = 5$, the longest substring is the empty one.

Recall that a substring is defined as a contiguous sequence of characters within a string. For example, `monk`, `onke`, and `key` are substrings of `monkey` but `money` is not. By definition, a string is always a substring of itself. Note that, in some cases, there may be multiple answers. In this problem, in such cases, the interest is in the substring lexicographically smallest (i.e., the one appearing first in a dictionary). For example, `aac` is lexicographically smaller than `aacb`, `ab`, and `aad`, but `aac` is not lexicographically smaller than `aab`.

Given a list of N passwords and an integer K, what is the longest string in the N passwords appearing at least in K of them?

Input

The input consists of several test cases. Each test case starts with a line with two integers N and K ($1 \leq N \leq 100\,000$ and $1 \leq K \leq N$). The next N lines contain the password list, one per line. The passwords are non-empty and only contain lowercase English letters (`a-z`). The sum of the lengths of all passwords in the list does not exceed 100 000. The last line of the input contains two blank-separated zeros and should not be processed.

The input must be read from standard input.

Output

For each test case output a single line with the longest string that is a substring of at least K passwords. If there are multiple possible options, output the one that is lexicographically smallest. If such a string is the empty one, output a single asterisk (*).

The output must be written to standard output.

Sample Input	Sample Output
5 1 monkey monk money motorcycle recycle 5 2 monkey monk money motorcycle recycle 5 3 monkey monk money motorcycle recycle 5 4 monkey monk money motorcycle recycle 5 5 monkey monk money motorcycle recycle 3 1 security security insecure 3 2 security security insecure 3 3 security security insecure 0 0	motorcycle cycle mon mo * insecure security secur

G: **Grain Silos**

Source file name: `grain.c`, `grain.cpp`, `grain.java`, *or* `grain.py`
Author: Julián Badillo

Winter is coming! King Arthur can feel it in his bones and Camelot needs to prepare for it. This is a gigantic amount of work, including harvesting grass for the cattle, salting and smoking meat, collecting firewood, and the sort.

One crucial task is to stockpile grain sacks to survive the winter. Camelot, being a prosperous kingdom, harvests and collects different types of grain: barley, wheat, rye, and many more. They are bagged in sacks, which are --in turn-- stacked in silos. All silos follow the same simple design: all have the same capacity and grain sacks are piled on top of each other.

An industrious, but not very grain-virtuoso, Sir Lancelot was bestowed with the noble task of gathering all the grain in the Kingdom. When King Arthur --his boss-- came to inspect his work, he found that Sir Lancelot mixed sacks from different types of grain in the same silo. This is a major disaster because it makes accounting, planning, and distribution rather difficult. There is also the risk of cross-contamination.

Your task is to help Sir Lancelot in sorting the sacks of grain in the silos, so that each silo ends up with one type of grain and all sacks of the same type are stored in a single silo. You need to find an efficient way to do it (i.e., one with the minimum amount of moves). Remember, it is late already and winter is coming, and King Arthur is not highly pleased with the current situation.

Keep in mind that:

- Silos only have a hatch at the top. You can only take the top sack of grain from one silo and place it at the top of another silo.
- You can move a sack of grain from one silo to another only if the target silo is empty or contains the same type of grain at the top. The capacity of a silo cannot be exceed at any point in time.
- You cannot leave any sack of grain in the open at any point in time or you'll be feeding the crows, which is considered a bad omen.

Consider the following example, where Sir Lancelot has five silos to account for, each with capacity for 3 sacks. Each grain sack is represented by a letter corresponding to the type of grain it contains: W for wheat, B for barley, and R for rye.

```
B R
B R W
W B W R
---------
1 2 3 4 5
```

Some valid moves are:

- Moving the top sack of grain (R) from silo 2 to silo 4.
- Moving the top sack of grain (W) from silo 3 to silo 5.

Some invalid moves would be:

- Moving the top sack of grain (R) from silo 2 to silo 3, since silo 3 contains a different type of grain at the top.

- Moving any sack to silo 1, since it is at full capacity.

The following sequence of moves sorts the grain as King Arthur requires:

```
B R                   B                     B     R
B R W                 B R W R               B   W R
W B W R               W B W R               W B W R
---------             ---------             ---------
1 2 3 4 5             1 2 3 4 5             1 2 3 4 5


      R                 B   R                 B W R
B B W R                 B W R                 B W R
W B W R               W B W R                 B W R
---------             ---------             ---------
1 2 3 4 5             1 2 3 4 5             1 2 3 4 5
```

Although there may be different ways to sort this mess, 5 moves is the best available option.

Input

The input consists of several test cases. Each case begins with a line containing two blank-separated integers N and C, $1 \leq N, C \leq 15$, indicating the number of silos and the capacity of each silo, respectively. Then, C lines follow, each with exactly N characters. They define the initial configuration of the N silos, where each silo configuration is given by the corresponding column (from top to bottom, similar to the descriptions used in the problem statement). A letter represents a sack of grain and a period '`.`' an empty slot in a silo. There are not empty slots beneath a sack grain. You can assume that there are at most N different types of grain (i.e., each type of grain may have its own silo) and at most C number of sacks of the same type of grain (i.e., all sacks of the same grain will fit in a single silo).

The last line contains two blank-separated zeros, which should not be processed.

The input must be read from standard input.

Output

For each test case, output a single line with the minimum number of moves required to sort the grain according to the given rules and King Arthur's standards. If it is not possible, output '`Camelot Will Starve!`'.

The output must be written to standard output.

Sample Input	**Sample Output**

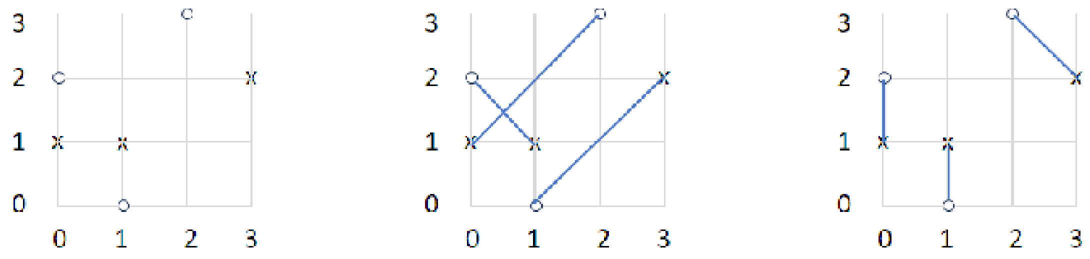
```
5 3
BR...
BRW..
WBWR.
3 2
BB.
RR.
3 4
...
BBB
RRR
WWW
0 0
```
	```
	5
	3
	Camelot Will Starve!
	```

H: **Match Points**

Source file name: `matchp.c`, `matchp.cpp`, `matchp.java`, *or* `matchp.py`
Author: Rodrigo Cardoso

International Connecting Points Co. (ICPC) is a technological enterprise that produces printed electronic circuit cards. A card consists of a square grid where *connecting points* of two kinds, O and X, are located. A card is said to be *well done* if every O-point is connected, by means of a straight segment, with exactly one X-point, and vice versa. Of course, since we are talking about printed circuits, there cannot be any crossing among connections in a well done card. A *match* is a list describing how points are connected on a card. For m a natural number, an m-*card* may be considered as a grid where $2m$ points are located. As you may suppose, there are m O-points and m X-points defined on the card.

The next figure depicts a match problem for a 3-card: the one on the left establishes the set of O- and X-points to connect; the one at the center defines a match, but not a well done one; and the one on the right does indeed define a well done match.



Another interesting feature of a well done match is its *length*, defined as the sum of the lengths of the segments defined in the match. It should be clear that ICPC prefers well done matches that use as few connection material as possible. An *optimal match* is a well done match of minimal length.

ICPC researchers have demonstrated a curious fact: if no three of the given $2m$ points on a card are collinear, there is always at least one match that corresponds to a well done card. The problem is that, for the moment, nobody knows how to mechanically find such a match. Anyway, knowing of the existence of these non-crossing matches, ICPC is now interested in calculating the length of an optimal match. In the above example, it is easy to see that the given solution (the one on the right) corresponds to an optimal match.

Your task is to help ICPC to find the length of an optimal match, given the O- and X-points of an m-card (no three of which are collinear).

Input

The input consists of several test cases, each defining an m-card. A case begins with a line containing an integer m, the number of O- and X-points, $0 < m \leq 200$. Then, two lines follow, each one with $2m$ integer values

$$x_1 \; y_1 \; x_2 \; y_2 \; \cdots \; x_m \; y_m$$
$$u_1 \; v_1 \; u_2 \; v_2 \; \cdots \; u_m \; v_m$$

where $\langle x_i, y_i \rangle$ and $\langle u_i, v_i \rangle$ are the coordinates of the i-th O-point and the i-th X-point, respectively. You may assume that no three of these $2m$ points are collinear, $0 \leq x_i, y_i, u_i, v_i \leq 1\,000$, and $1 \leq i \leq m$. The end of the input is indicated by a line containing a single zero, which should be not processed.

The input must be read from standard input.

Output

For each test case, output a line consisting of a real number rounded to 3 decimal places: the length of an optimal match for the given *m*-card.

The output must be written to standard output.

Sample Input

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

Sample Output

```
2.000
3.414
```

I: Stack Solitaire

Source file name: `stack.c`, `stack.cpp`, `stack.java`, *or* `stack.py`
Author: Rafael García

The International Casino for Programming Contestants (ICPC) has introduced a new game: the *Stack Solitaire*. At the start of the game, the player is given a stack of N coins, each having a, possibly different, positive integer value. As the game goes on, the number of stacks will increase.

At each turn, the player selects one stack from those having at least two coins and divides it into two smaller stacks, keeping the original order of the coins. The payoff obtained in each play is equal to the sum of the values of the coins in the split stack. The player's final payoff is obtained by adding up the payoffs from each play. The game ends when the player cannot split any stack, i.e., there are N stacks of one coin each.

For example, for the initial stack $\begin{bmatrix}5\\10\\2\\7\end{bmatrix}$, there exist five possible sequences of plays. Two of them are

$$\begin{bmatrix}5\\10\\2\\7\end{bmatrix} \longrightarrow \begin{bmatrix}5\end{bmatrix}\begin{bmatrix}10\\2\\7\end{bmatrix} \longrightarrow \begin{bmatrix}5\end{bmatrix}\begin{bmatrix}10\\2\end{bmatrix}\begin{bmatrix}7\end{bmatrix} \longrightarrow \begin{bmatrix}5\end{bmatrix}\begin{bmatrix}10\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}7\end{bmatrix}$$

with a total payoff of $(5 + 10 + 2 + 7) + (10 + 2 + 7) + (10 + 2) = 55$; and,

$$\begin{bmatrix}5\\10\\2\\7\end{bmatrix} \longrightarrow \begin{bmatrix}5\\10\end{bmatrix}\begin{bmatrix}2\\7\end{bmatrix} \longrightarrow \begin{bmatrix}5\end{bmatrix}\begin{bmatrix}10\end{bmatrix}\begin{bmatrix}2\\7\end{bmatrix} \longrightarrow \begin{bmatrix}5\end{bmatrix}\begin{bmatrix}10\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}7\end{bmatrix}$$

with a total payoff of $(5 + 10 + 2 + 7) + (5 + 10) + (2 + 7) = 48$.

Given an initial stack, what is the minimum possible final score?

Input

The input consists of several test cases, each defined by two lines. The first line contains an integer N ($2 \le N \le 1\,000$), the number of coins in the initial stack. The second line contains N positive integers representing the values of the N coins in the stack, from bottom to top. Each coin value v is such that $1 \le v \le 100$.

The end of the input is indicated by a line with a single zero.

The input must be read from standard input.

Output

For each test case, output a line with the minimum possible final payoff for the given initial stack.

The output must be written to standard output.

Sample Input	**Sample Output**

```
4
7 2 10 5
3
2 10 7
0
```
	```
	48
	31
	```

J: **TNumbers**

Source file name: `tnumbers.c`, `tnumbers.cpp`, `tnumbers.java`, *or* `tnumbers.py`
Author: Rodrigo Cardoso

Note that

$$1 + 2 = 3,$$

but

$$\begin{aligned} 1 &\neq 2 + 3 + 4 \\ 1 + 2 &\neq 3 + 4 \\ 1 + 2 + 3 &\neq 4. \end{aligned}$$

A *TNumber* n is a positive integer for which there exist a number k, $1 \leq k < n$, such that the sum of the first k numbers equals the sum of the numbers from $k + 1$ to n. It is clear that 3 is a TNumber, but 4 is not.

Given two non-negative integers a and b, with $a \leq b$, determine how many numbers n satisfying $a \leq n \leq b$ are TNumbers.

Input

The problem input consists of several cases. A case is described with a line with two integer numbers a and b, $1 \leq a \leq b \leq 10^8$. The end of the input is signaled by a line with two zero values '`0 0`', which should not be processed.

The input must be read from standard input.

Output

For each case, output a line with exactly one integer value indicating how many TNumbers are there.

The output must be written to standard output.

Sample Input	**Sample Output**

`1 5`
`3 3`
`4 8`
`0 0`
	`1`
	`1`
	`0`