Problem A: Happy Camper

Source: `camper.{c,cpp,java}`

As Happy Camper Harry pulls into his favorite campground with his family, he notices the sign: 'Campground occupancy is limited to 10 days within any consecutive 20-day period.' Harry is just starting a 28-day vacation. What is the maximum number of days he can occupy a campsite during his vacation?

We state the problem in more general terms. Suppose that $1 < L < P < V$ are integers. Campground occupancy is limited to L days within any consecutive P-day period. Happy Camper Harry is just starting a V-day vacation. What is the maximum number of days he can occupy a campsite during his vacation?

Input

The input will contain data for a number of test cases. For each test case, there will be one line of data, containing values of L, P and V, in that order. All input integers can be represented by signed 32-bit integers. End of data will be signaled by a line containing three zeros, which will not be processed.

Output

There will be one line of output for each test case. It will display the case number and the number of days Happy Camper Harry can occupy a campsite during his vacation. The format is illustrated by the sample output.

Sample Input

```
5 8 20
5 8 17
0 0 0
```

Sample Output

```
Case 1: 14
Case 2: 11
```

Problem B: Chemistry

Source: `chemistry.{c,cpp,java}`

The chemical formula of a molecule **`M`** describes its atomic make-up. Chemical formulas obey the following grammar:

```
M := G | M G
G := S | S C
S := A | '(' M ')'
C := T | N E
E := D | D E
T := '2' | ... | '9'
N := '1' | ... | '9'
D := '0' | .. | '9'
A := U | U L | U L L
U := 'A' | .. | 'Z'
L := 'a' | .. | 'z'
```

The count **`C`** represents a multiplier for the subgroup **`S`** that precedes it. For example, **`H2O`** has two **`H`** (hydrogen) and one **`O`** (oxygen) atoms, and **`(AlC2)3Na4`** contains 3 **`Al`** (aluminum), 6 **`C`** (carbon) and 4 **`Na`** (sodium) atoms.

Input

The input will contain data for one or more test cases. For each test case, there will be one line of input, containing a valid chemical formula. Each line will have no more than 100 characters.

Output

For each line of input there will be one line of output which is the atomic decomposition of the chemical in the form of a sum as shown in the sample output. The atoms are listed in lexicographical order, and a count of 1 is implied and not explicitly written. There are no blank spaces in the output. All of the counts in the correct output will be representable in 32-bit signed integers.

Sample Input

```
H2O
(AlC2)3Na4
```

Sample Output

```
2H+O
3Al+6C+4Na
```

Problem C: Dirichlet's Theorem

Source: `dirichlet.{c,cpp,java}`

Dirichlet's theorem on arithmetic progressions states that for any two positive integers **a** and **b**, if **gcd(a,b) = 1** then the arithmetic progression **t(n) = a*n + b (n ≥ 0)** contains infinitely many prime numbers. Recall that a prime number is a positive integer ≥ 2 that has no divisors other than 1 and itself.

For example, if **a = 4** and **b = 3**, then the arithmetic progression is

```
3, 7, 11, 15, 19, 23, 27, 31, 35, ...,
```

and it can be seen that many prime numbers are contained in the first part of this list.

Given arbitrary integers **a > 0**, **b ≥ 0**, and **U ≥ L ≥ 0**, your job is to count how many values of **t(n) = a*n+b** are prime, where **L ≤ n ≤ U**.

Input

The input consists of a number of cases. The input for each case is specified by the four integers **a**, **b**, **L**, and **U** on a line. You may assume that **a*U+b ≤ 10^{12}** and **U - L ≤ 10^{6}**. A line containing a single 0 indicates the end of input.

Output

For each test case, print:

```
Case xxx: yyy
```

where **xxx** is the case number (starting from 1), and **yyy** is the number of **t(n), L ≤ n ≤ U**, that are prime.

Sample Input

```
4 3 0 8
1 0 2 100
2 7 0 1000
0
```

Sample Output

```
Case 1: 6
Case 2: 25
Case 3: 301
```

Problem D: Fuel Stops

Source: `fuel.{c,cpp,java}`

You are required to take a circular tour of a given set of cities: start at a certain city, visit each city once, and return to the city at which you started.

Each city is identified by a number: 1, 2, 3, etc. The numbering of the cities specifies the path you must take, but the starting point is not specified. From the highest numbered city, you proceed to City 1. For example, if there are three cities (numbered 1, 2, 3) you have three choices for completing the tour:

Start at 1, proceed to 2, then to 3, then return to 1.
Start at 2, proceed to 3, then to 1, then return to 2.
Start at 3, proceed to 1, then to 2, then return to 3.

There is a refueling station in each city, with a given quantity of available fuel. The sum of all the fuel supplies at the refuelling stations is equal to the fuel required to make the entire tour. You start with an empty tank at one of the refuelling stations. You will be running out of fuel just as you pull into the starting station upon successful completion of the tour.

You must determine which city (or cities) will qualify as a starting point for the tour without running out of fuel before returning to the starting station. Assume the fuel tank is sufficiently large to handle all of the refuelling operations.

Input

The input will contain data for several test cases. For each test case, there will be three lines of data. The first line will specify the number of cities as a single integer. The second line will specify the quantity of fuel available at each of the refuelling stations, in the order of city numbers: 1, 2, 3, etc. The third line will specify the quantity of fuel needed to get from each station to the next one, in the order of city numbers: from the station at city 1 to the station at city 2, from the station at city 2 to the station at city 3, etc; the last number specifies the quantity of fuel required to get from the highest numbered city's station back to the station at city 1. All fuel quantities are positive integers given in imperial gallons. The sum of the fuel supplies will not exceed the range of signed 32-bit integers. There will be at least two cities and up to 100000 cities. End of input will be indicated by a line containing zero for the number of cities; this line will not be processed.

Output

For each test case, there will be one line of output. After the case number, the output will list the city numbers that work as starting cities for a successful tour, as described above. In case of several possible starting cities, they must be listed in increasing order separated by a single space. Follow the format of the sample output. The Hungarian mathematician L. Lovász proved that there is always at least one possible starting city.

Sample Input

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

Sample Output

```
Case 1: 2 3
Case 2: 1
Case 3: 4
```

Problem E: Approximate Sorting

Source: `sorting.{c,cpp,java}`

In many programming languages, library functions are provided for sorting elements in an array. In order for these sorting functions to work for different types of elements, the user supplies a comparison function **`less(x,y)`** which returns true if and only if **`x`** comes before **`y`** in the sorted order. Of course, the comparison function has to 'make sense'. For example, for any two different elements **`x`** and **`y`**, exactly one of **`less(x,y)`** and **`less(y,x)`** should be true. For the purpose of this problem, an array is sorted when there are no inversions with respect to the comparison function. An inversion with respect to **`less(x,y)`** in an array **`A`** of size **`n`** (indexed from 0) is a pair of integers **`0 ≤ i < j < n`** such that **`less(A[j], A[i]) = true`** (note that this may not be equivalent to **`less(A[i], A[j]) = false`**).

Unfortunately, some programmers are not very careful in defining the comparison function. In such cases, there may be no way to sort the elements in an array to satisfy the comparison function. The best we can do is to produce a permutation minimizing the number of inversions with respect to the given comparison function.

Input

For each case, an integer **`n`**, $1 \le n \le 18$, indicating the size of the array is given on one line. The elements in the array are labelled **`0`**, **`1`**, **`2`**, ..., **`n-1`**. The next **`n`** lines each contains a binary string of length **`n`**, with the **`j`**th character of the **`i`**th line indicating the result of the comparison function **`less(i, j)`** (0 means false and 1 means true). The end of input is indicated by a case in which **`n = 0`**. The last case should not be processed.

Output

For each case, output the permutation that has the smallest number of inversions with respect to the given comparison function. This is followed by a line containing a single integer indicating the number of inversions in the permutation. If there are multiple permutations with the same number of inversions, output the one that is lexicographically smallest.

Sample Input

```
4
0111
0000
0100
0110
3
011
011
011
6
101010
011010
110110
000000
111010
001010
0
```

Sample Output

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

Problem F: Nine

Source: `nine.{c,cpp,java}`

Randall Munroe from **`xkcd.com`** pointed out that 9 is the most rarely used key on a microwave. Let's all share the load.

Given a desired cooking time, find a sequence of keys with the greatest number of 9's such that the resulting time has less than 10% error compared to the desired cooking time. In other words, if **`T`** is the desired cooking time in seconds, and **`T9`** is the cooking time specified by the found sequence, then **`10|T - T9| < T`**. If there are multiple possibilities, choose the one that has the smallest error (in magnitude). If there are still ties, choose the one that is lexicographically smallest.

For example, for T = 01:15, the times 00:68-00:82 and 1:08-1:22 have less than 10% error. Of these, 00:69, 00:79, 01:09, and 01:19 have the greatest number of 9's, and the ones with the smallest error are 00:79 and 01:19. The lexicographically smaller of these is 00:79.

EVER SINCE I HEARD THE SIMILE "AS NEGLECTED AS THE NINE BUTTON ON THE MICROWAVE" I'VE FOUND MYSELF ADJUSTING COOK TIMES.

Input

The input consists of a number of cases. For each case, the desired cooking time in **`MM:SS`** format is specified on one line. Each **`M`** or **`S`** can be any digit from 0 to 9. The end of input is indicated by 00:00.

Output

For each case, output on a single line the four keys to use as input to the microwave, in **`MM:SS`** format.

Sample Input

```
00:30
01:00
02:00
91:30
46:03
00:00
```

Sample Output

```
00:29
00:59
01:59
90:99
49:99
```

Problem G: Function Overloading

Source: `overload.{c,cpp,java}`

Many programming languages (including C, C++ and Java) allow the programmer to define overloaded functions, i.e., several functions that have the same name but different parameter lists. However, in some languages (such as Ada) it is possible to overload by return type as well. That is, it is possible for two functions to have the same name and parameter list, but different return types. For example:

```
char f(float x, int y)
char f(float x, float y)
float f(float x, float y)
float g(float x, int y)
float g(int x, float y)
```

Given these function declarations, suppose the program contains the following variable declarations and function call:

```
float a = 1.0, b = 2.0;
int c = 3;
float d = g(c, f(a, b));
```

The first two declarations of f would not work here, but the third one does: **`f(<float>, <float>)`** returns **`<float>`**, which reduces the function call to **`g(<int>, <float>)`**, and the second declaration of **`g`** will return the **`<float>`** that can be assigned to variable **`d`**. Since we used the 3rd version of **`f`** and the 2nd version of **`g`**, we say that the given function call is resolved by

```
d = g2(c, f3(a, b)).
```

Using the same declarations, the function call **`c = g(a, f(a, c))`** cannot be resolved. As a final example, consider the function declarations

```
float x(float w)
int x(float w)
char y(float v)
char y(int v)
```

and the variable declaration and function call

```
float a = 1.0;
char c = y(x(a));
```

In this case, we see that the resolution of the given function call is ambiguous.

Input

The input will consist of a list of function declarations (one per line). Each function declaration in the input will have the form

```
name num_params param(1) param(2) ... param(num_params) rettype
```

where **name** is the function name, **param(i)** is the data type of the i-th parameter, and rettype is the data type of the return value (this problem does not deal with 'void' functions); **num_params** is at least 1 and at most 9. Note that the parameters do not have names, it is only their data types that matter. Function names are single lower case letters, while data types are single upper case letters. Different functions with the same name will appear consecutively in the input, and there are at most 500 different functions for each function name. No two function declarations will be exactly the same. Each function declaration carries with it, implicitly, a 'serial number'. The serial numbers start out at 1 and increase until a new function name is encountered; then they start out at 1 again.

The list of function declarations in the input will be concluded by a line containing only a pound sign ('#'). Thereafter will come a list of function calls (one per line).

The structure of each function call can be defined by the grammar:

```
<function_call> := <data_type> = <right_hand_side>
<right_hand_side> := <fname> <num_params> <param_list>
<param_list> := <param> | <param_list> <param>
<param> := <data_type> | <right_hand_side>
<data_type> := <upper_case_letter>
<num_params> := '1' | '2' | ... | '9'
<fname> := <lower_case_letter>
```

Here the symbols **:=** and **|** are used to define the grammar, they will not show up in the actual function call. Furthermore, in any function call the specified num_params will match exactly the number of parameters given in the parameter list. Each function call will contain no more than 500 function names (not necessarily distinct). The end of the list of function calls will be marked by a line containing only a pound sign. In each function declaration, as well as in each function call in the input, adjacent tokens (lower or upper case letters, digits, equals sign) will be separated by exactly one blank space.

Output

For each function call in the input, there will be one line of output. If the function call can be resolved uniquely, the output will be the same as the input function call, but each function name will have a serial number appended to it, to indicate which version of the function was used there. Otherwise, the output will be either 'impossible' or 'ambiguous', as explained above. If it is ambiguous, also output the number of ways the function call can be resolved, or print '> 1000' if there are more than 1000 ways.

Sample Input

```
f 2 A B C
f 2 A A C
f 2 A A A
g 2 A B A
g 2 B A A
x 1 A A
x 1 A B
y 1 A C
y 1 B C
h 2 A B E
h 2 C D F
k 2 E F A
#
A = g 2 B f 2 A A
B = g 2 A f 2 A B
C = y 1 x 1 A
A = k 2 h 2 A B h 2 C D
#
```

Sample Output

```
A = g2 2 B f3 2 A A
impossible
ambiguous 2
A = k1 2 h1 2 A B h2 2 C D
```

Problem H: Trees

Source: `trees.{c,cpp,java}`

A graph consists of a set of vertices and edges between pairs of vertices. Two vertices are connected if there is a path (subset of edges) leading from one vertex to another, and a connected component is a maximal subset of vertices that are all connected to each other. A graph consists of one or more connected components.

A tree is a connected component without cycles, but it can also be characterized in other ways. For example, a tree consisting of n vertices has exactly n-1 edges. Also, there is a unique path connecting any pair of vertices in a tree.

Given a graph, report the number of connected components that are also trees.

Input

The input consists of a number of cases. Each case starts with two non-negative integers **`n`** and **`m`**, satisfying **`n ≤ 500`** and **`m ≤ n(n-1)/2`**. This is followed by m lines, each containing two integers specifying the two distinct vertices connected by an edge. No edge will be specified twice (or given again in a different order). The vertices are labelled **`1`** to **`n`**. The end of input is indicated by a line containing **`n = m = 0`**.

Output

For each case, print one of the following lines depending on how many different connected components are trees (T > 1 below):

```
Case x: A forest of T trees.
Case x: There is one tree.
Case x: No trees.
```

`x` is the case number (starting from 1).

Sample Input

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

Sample Output

```
Case 1: A forest of 3 trees.
Case 2: There is one tree.
Case 3: No trees.
```

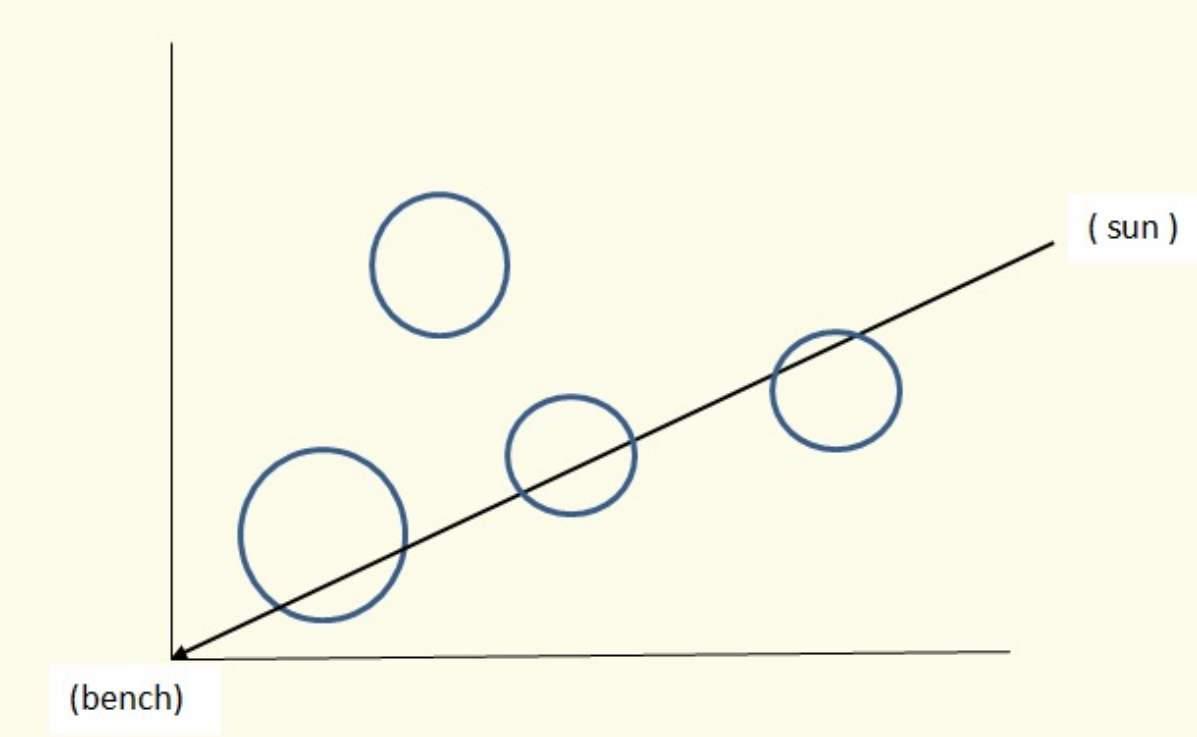
Problem I: Catching Shade in Flatland

Source: `shade.{c,cpp,java}`

It is always sunny in Flatland. This tends to get annoying so Alex is heading to a park trying to catch some shade.

The park is a square with sides of 400 units centered at the origin of the Cartesian plane (vertices at (±200,±200)). There is a point-sized bench in the center of the park (at the origin) and the Sun is another point traveling clockwise along the circle centered at origin and with a radius of 500 units. At midnight the Sun is at the point (0,500) and it takes 24 hours for it to complete the full revolution.

There are several trees in the park that can provide shade for Alex. Trees are modeled as solid circles with integer coordinates and radii. The 'shade value' at any particular time is defined as the sum of the lengths of the chords created by the intersections of a ray of sunlight with trees. (The sum of the lengths of the chords is a measure of how much 'leaves and branches' the ray of light would have to pass through.) This is illustrated in the diagram, in which trees are solid circles but only the boundaries are drawn to show the chords.



We compute the shade value every minute, on the minute over a 24-hour period, starting at midnight: at 00:00, 00:01, ..., 23:59. We would like to determine the maximum of these shade values.

Input

There are number of test cases (at most 100), each starting with a line with integer **N**, **1≤N≤200**, the number of trees. Next follow **N** lines each containing three integers x_i, y_i, r_i - coordinates of the center and the value of the radius of i^{th} tree. The last case is followed by a line containing a single 0.

No tree contains or intersects (but may touch) another one nor does it go outside the edge of the park. No tree contains or touches the bench.

Output

For each test case, print on a single line the maximum of the computed shade values, displayed to 3 decimal places. An output value is acceptable if it is within 0.001 of the correct value,

Sample Input

```
1
50 0 10
2
30 0 10
60 20 20
0
```

Sample Output

```
20.000
53.036
```

Problem J: Emergency Room

Source: `emergency.{c,cpp,java}`

An emergency room is serviced by a number of doctors. When patients arrive in the emergency room, their arrival time is recorded and they are assigned a number of treatments. Each treatment has a priority level and duration. For any particular patient, the priority levels of successive treatments form a strictly decreasing sequence, such as the numbers 8, 5, 3 in the following example:

```
Treatment 1: priority = 8, duration = 10 units of time
Treatment 2: priority = 5, duration = 25 units of time
Treatment 3: priority = 3, duration = 15 units of time
```

Each treatment must be performed by a doctor; different treatments for the same person do not need to be performed by the same doctor. Any particular doctor can treat only one patient at any time.

All doctors of the facility will open their cubicles in the morning and become available at the same time. Some patients may have already arrived at that time, and additional patients may arrive subsequently. Whenever there is an available doctor, the patient with the highest priority (i.e., priority of the next treatment to be performed) will be selected from the waiting room and assigned to the available doctor. In case of a tie, the person with earliest original arrival time will be chosen. When more than one doctor is available, more than one patient may be admitted to the next treatment at the same time.

When, for a particular patient, a treament has been completed, but there are still remaining treatments pending, the patient will return to the waiting room and wait for his/her next turn. When all of a particular patient's scheduled treatments have been completed, the patient will be released from the facility.

Input

The input will contain data for several test cases. Under each test case, the first line of input will contain two positive integers: the number of doctors at the facility, and the clock reading at which the doctors become available in the morning. All times in the input will be expressed in some unspecified unit of time, as a single positive integer up to 1000. The remainder of the input for each test case will provide information about the patients. Under each patient, the first line will state the arrival time. The remaining lines under each patient will specify the pending treatments for that patient, one line per treatment. Each treatment will be represented by two positive integers: priority and duration. Under each patient, the priorities will be in strictly decreasing order. No two patients have the same arrival time. The patients will be listed in increasing order of arrival times. A line containing two 0s will mark the end of each patient's data, a line containing -1 will mark the end of each test case, and another line containing two 0s will mark the end of input. There will be at most 500 doctors and 500 patients in each case. The maximum priority and duration for any treatment is 100.

Output

After printing the case number, the output will display the time when each patient can be released from the emergency room. Patients will be identified by their arrival times; they will be listed in increasing order of release times. If there are several patients released at the same time, display them in the order of original arrival times. See the sample output for the exact format.

Sample Input

```
1 50
10
10 5
5 20
4 5
0 0
30
25 10
8 5
5 5
0 0
110
20 10
0 0
-1
2 50
10
10 5
5 20
4 5
0 0
30
25 10
8 5
5 5
0 0
110
20 10
0 0
-1
0 0
```

Sample Output

```
Case 1:
Patient 30 released at clock = 95
Patient 10 released at clock = 100
Patient 110 released at clock = 120
Case 2:
Patient 30 released at clock = 70
Patient 10 released at clock = 80
Patient 110 released at clock = 120
```