Problem A: Calculate e !

(blue balloon)

A simple mathematical formula for *e* is

$$e = \sum_{i=0}^{n} \frac{1}{i!}$$

where *n* is allowed to go to infinity. This can actually yield very accurate approximations of *e* using relatively small values of *n*.

Standard Output

Output the approximations of *e* generated by the above formula for the values of *n* from 0 to 9. The beginning of your output should appear similar to that shown below.

Sample Output

```
n e
- -----------
0 1
1 2
2 2.5
3 2.666666667
4 2.708333334
```

Problem B: Networks

(red balloon)

Consider the two networks shown below. Assuming that data moves around these networks only between directly connected nodes on a peer-to-peer basis, a failure of a single node, 3, in the network on the left would prevent some of the still available nodes from communicating with each other. Nodes 1 and 2 could still communicate with each other as could nodes 4 and 5, but communication between any other pairs of nodes would no longer be possible.

Node 3 is therefore a Single Point of Failure (SPF) for this network. Strictly, an SPF will be defined as any node that, if unavailable, would prevent at least one pair of available nodes from being able to communicate on what was previously a fully connected network. Note that the network on the right has no such node; there is no SPF in the network. At least two machines must fail before there are any pairs of available nodes which cannot communicate.
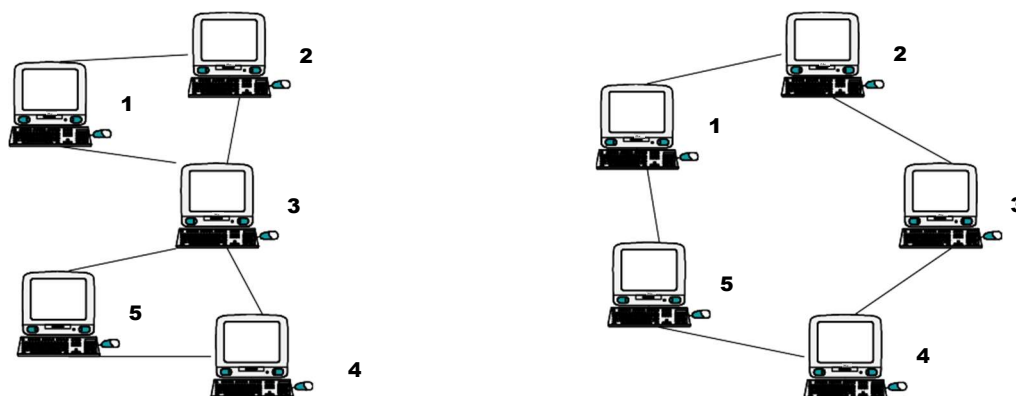
Standard Input

The input will contain the description of several networks. A network description will consist of pairs of integers, one pair per line, that identify connected nodes. Ordering of the pairs is irrelevant; 1 2 and 2 1 specify the same connection. All node numbers will range from 1 to 1000. A line containing a single zero ends the list of connected nodes. An empty network description flags the end of the input. Blank lines in the input should be ignored.

Standard Output

For each network in the input, you will output its number in the standard output, followed by a list of any SPF nodes that exist. The first network in the standard output should be identified as "Network #1", the second as "Network #2", etc. For each SPF node, output a line, formatted as shown in the examples below, that identifies the node and the number of fully connected subnets that remain when that node fails. If the network has no SPF nodes, simply output the text "No SPF nodes" instead of a list of SPF nodes.

Sample Input	Sample Output

<pre>1 2
5 4
3 1
3 2
3 4
3 5
0
 	
1 2
2 3
3 4
4 5
5 1
0
 	
1 2
2 3
3 4
4 6
6 3
2 5
5 1
0
 	
0</pre>
	<pre>Network #1
	  SPF node 3 leaves 2 subnets
	Network #2
	  No SPF nodes
	Network #3
	  SPF node 2 leaves 2 subnets
	  SPF node 3 leaves 2 subnets</pre>

Problem C: Roots

(yellow balloon)

The *digital root* of a positive integer is found by summing the digits of the integer. If the resulting value is a single digit then that digit is the digital root. If the resulting value contains two or more digits, those digits are summed and the process is repeated. This is continued as long as necessary to obtain a single digit.

For example, consider the positive integer 24. Adding the 2 and the 4 yields a value of 6. Since 6 is a single digit, 6 is the digital root of 24. Now consider the positive integer 39. Adding the 3 and the 9 yields 12. Since 12 is not a single digit, the process must be repeated. Adding the 1 and the 2 yields 3, a single digit and also the digital root of 39.

Standard Input

The input will contain a list of positive integers, one per line. The end of the input will be indicated by an integer value of zero.

Standard Output

For each integer in the input, output its digital root on a separate line of the output.

Sample Input	Sample Output
24 39 0	6 3

Problem D: Sort
(Orange balloon)

In this problem you will be given a series of lists containing both words and numbers. The goal is to sort these lists in such a way that all words are in alphabetical order and all numbers are in numerical order. Furthermore, if the nth element in the list is a number it must remain a number, and if it is a word it must remain a word.

Standard Input

The input will contain multiple lists, one per line. Each element of the list will be separated by a comma followed a space, and the list will be terminated by a period. The input will be terminated by a line containing only a single period.

Standard Output

For each list in the input, output the scramble sorted list, separating each element of the list with a comma followed by a space, and ending the list with a period.

Sample Input

```
0.
banana, strawberry, OrAnGe.
Banana, StRaWbErRy, orange.
10, 8, 6, 4, 2, 0.
x, 30, -20, z, 1000, 1, Y.
50, 7, kitten, puppy, 2, orangutan, 52, -100, bird, worm, 7, beetle.
.
```

Sample Output

```
0.
banana, OrAnGe, strawberry.
Banana, orange, StRaWbErRy.
0, 2, 4, 6, 8, 10.
x, -20, 1, Y, 30, 1000, z.
-100, 2, beetle, bird, 7, kitten, 7, 50, orangutan, puppy, 52, worm.
```

Problem E : XML

(purple balloon)

XML, eXtensible Markup Language, is poised to become the *lingua franca* of structured data communication for the foreseeable future, due in part to its strict formatting requirements. XML parsers must report anything that violates the rules of a *well-formed* XML document. An XML document is said to be well-formed if it meets all of the well- formedness constraints as defined by the World Wide Web Consortium (W3C) XML specification.

XML documents are composed of units called elements, that contain either character data and/or other elements. Elements may also contain within their declaration values called attributes. Consider the following XML document:

```
<?xml version="1.0"?>
<customer>
        <name>
                <first>John</first>
                <last>Doe</last>
        </name>
        <address>
                <street>
                        <number>15</number>
                        <direction>West</direction>
                        <name>34th</name>
                </street>
                <city>New York</city>
                <state-code>NY</state-code>
                <zip-code format="PLUS4">10001-0001</zip-code>
                <country-code>USA</country-code>
        </address>
        <orders/>
</customer>
```

The bold identifiers contained within angle brackets are the elements of the document. The italicized identifier "format" within the "zip-code" element is an attribute of that element. All elements, with the exception of "orders", have a start and an end declaration, also called a tags. The "orders" element is an empty element, as indicated by the "/>" sequence that closes the element, and does not require a separate end-tag. The first line is a processing instruction for an XML parser and is *not* considered an element of the document.

The rules for a well-formed document are:

1. **There is exactly one element that is not contained within any other element.** This element is identified as the "root" or "document" element. In the example above, "customer" is the document element.

2. **The structure of an XML document must nest properly.** An element's start-tag must be paired with a closing end-tag if it is a non-empty element.

3. **The name in an element's end-tag must match the element type in the start-tag.** For example, an element opened with <address> must be closed by </address>.

4. **No attribute may appear more than once in the same start-tag or empty-element tag.**

5. **A parsed element must not contain a recursive reference to itself.** For example, it is improper to include another address element within an address element.

6. **A named attribute must have an associated value.**

Standard Input

The standard input will contain a series of XML documents. The start of each document is identified by a line containing only the processing instruction "`<?xml version="1.0"?>`". The end of the input is identified by a line containing only the text "`<?end?>`" (this is not a true XML processing instruction, just a sentinel used to mark the end of the input for this problem). As with all XML documents, white space between elements and attributes should be ignored. You may make the following assumptions with regard to the input.

- The only processing instruction that will be present is the XML version processing instruction, and it will always appear only at the beginning of each document in the input.

- Element and attribute names are case-sensitive. For example, <Address> and <address> are considered to be different.

- Element and attribute names will use only alpha-numeric characters and the dash "-" character.

- XML comments will not appear in the input.

- Values for attributes will always be properly enclosed in double quotes.

Standard Output

For each input XML document, output a line containing the text "well-formed" if the document is well-formed, "non well-formed" otherwise.

Sample Input

```
<?xml version="1.0"?>
<acm-contest-problem>
        <title>A Well-Formed Problem</title>
        <text>XML, eXtensible Markup Language, is poised to become the lingua franca
of structured data communication for the foreseeable future. [...]</text>
        <input>probleme.in</input>
        <output>probleme.out</output>
</acm-contest-problem>
<?xml version="1.0"?>
<shopping-list>
        <items>
                <item quantity="1" quantity="1">Gallon of milk</item>
                <item>Frozen pizza
        </items>
</Shopping-list>
 <errand-list>
        <errand>Get some cash at the ATM
                <errand>Pick up dry cleaning</errand>
        </errand>
</errand-list>
<?end?>
```

Sample Output

```
well-formed
non well-formed
```

Problem F : Encoder

(green balloon)

An entropy encoder is a data encoding method that achieves lossless data compression by encoding a message with "wasted" or "extra" information removed. In other words, entropy encoding removes information that was not necessary in the first place to accurately encode the message. A high degree of entropy implies a message with a great deal of wasted information; english text encoded in ASCII is an example of a message type that has very high entropy. Already compressed messages, such as JPEG graphics or ZIP archives, have very little entropy and do not benefit from further attempts at entropy encoding.

English text encoded in ASCII has a high degree of entropy because all characters are encoded using the same number of bits, eight. It is a known fact that the letters E, L, N, R, S and T occur at a considerably higher frequency than do most other letters in english text. If a way could be found to encode just these letters with four bits, then the new encoding would be smaller, would contain all the original information, and would have less entropy. ASCII uses a fixed number of bits for a reason, however: it's easy, since one is always dealing with a fixed number of bits to represent each possible glyph or character. How would an encoding scheme that used four bits for the above letters be able to distinguish between the four-bit codes and eight-bit codes? This seemingly difficult problem is solved using what is known as a "prefix-free variable-length" encoding.

In such an encoding, any number of bits can be used to represent any glyph, and glyphs not present in the message are simply not encoded. However, in order to be able to recover the information, no bit pattern that encodes a glyph is allowed to be the prefix of any other encoding bit pattern. This allows the encoded bitstream to be read bit by bit, and whenever a set of bits is encountered that represents a glyph, that glyph can be decoded. If the prefix-free constraint was not enforced, then such a decoding would be impossible.

Consider the text "AAAAABCD". Using ASCII, encoding this would require 64 bits. If, instead, we encode "A" with the bit pattern "00", "B" with "01", "C" with "10", and "D" with "11" then we can encode this text in only 16 bits; the resulting bit pattern would be "0000000000011011". This is still a fixed-length encoding, however; we're using two bits per glyph instead of eight. Since the glyph "A" occurs with greater frequency, could we do better by encoding it with fewer bits? In fact we can, but in order to maintain a prefix-free encoding, some of the other bit patterns will become longer than two bits. An optimal encoding is to encode "A" with "0", "B" with "10", "C" with "110", and "D" with "111". (This is clearly not the *only* optimal encoding, as it is obvious that the encodings for B, C and D could be interchanged freely for any given encoding without increasing the size of the final encoded message.) Using this encoding, the message encodes in only 13 bits to "0000010110111", a compression ratio of 4.9 to 1 (that is, each bit in the final encoded message represents as much information as did 4.9 bits in the original encoding). Read through this bit pattern from left to right and you'll see that the prefix-free encoding makes it simple to decode this into the original text even though the codes have varying bit lengths.

As a second example, consider the text "THE CAT IN THE HAT". In this text, the letter "T" and the space character both occur with the highest frequency, so they will clearly have the shortest encoding bit patterns in an optimal encoding. The letters "C", "I" and "N" only occur once,

however, so they will have the longest codes.

There are many possible sets of prefix-free variable-length bit patterns that would yield the optimal encoding, that is, that would allow the text to be encoded in the fewest number of bits. One such optimal encoding is to encode spaces with "00", "A" with "100", "C" with "1110", "E" with "1111", "H" with "110", "I" with "1010", "N" with "1011" and "T" with "01". The optimal encoding therefore requires only 51 bits compared to the 144 that would be necessary to encode the message with 8-bit ASCII encoding, a compression ratio of 2.8 to 1.

Standard Input

The standard input will contain a list of text strings, one per line. The text strings will consist only of uppercase alphanumeric characters and underscores (which are used in place of spaces). The end of the input will be signalled by a line containing only the word "END" as the text string. This line should not be processed.

Standard Output

For each text string in the input, output the length in bits of the 8-bit ASCII encoding, the length in bits of an optimal prefix-free variable-length encoding, and the compression ratio accurate to one decimal point.

Sample Input	Sample Output
AAAAABCD THE_CAT_IN_THE_HAT END	64 13 4.9 144 51 2.8

Problem G : Mazes

(white balloon)

An *n-tersection* is defined as a location in *n*-dimensional space, *n* being a positive integer, having all non-negative integer coordinates. For example, the location (1,2,3) represents an n-tersection in three dimensional space. Two n-tersections are said to be *adjacent* if they have the same number of dimensions and their coordinates differ by exactly 1 in a single dimension only. For example, (1,2,3) is adjacent to (0,2,3) and (2,2,3) and (1,2,4), but not to (2,3,3) or (3,2,3) or (1,2). An *n-teresting space* is defined as a collection of paths between adjacent n-tersections.

Finally, an *n-credible maze* is defined as an n-teresting space combined with two specific n-tersections in that space, one of which is identified as the starting n-tersection and the other as the ending n-tersection.

Standard Input

The standard input will consist of the descriptions of one or more n-credible mazes. The first line of the description will specify *n*, the dimension of the n-teresting space. (For this problem, *n* will not exceed 10, and all coordinate values will be less than 10). The next line will contain 2*n* non-negative integers, the first *n* of which describe the starting n-tersection, least dimension first, and the next *n* of which describe the ending n-tersection. Next will be a non-negative number of lines containing 2*n* non-negative integers each, identifying paths between adjacent n-tersections in the n-teresting space. The list is terminated by a line containing only the value –1. Several such maze descriptions may be present in the standard input. The end of the input is signaled by space dimension of zero. No further data will follow this terminating zero.

Standard Output

For each maze output it's position in the input; e.g. the first maze is "Maze #1", the second is "Maze #2", etc. If it is possible to travel through the n-credible maze's n-teresting space from the starting n-tersection to the ending n-tersection, also output "can be travelled" on the same line. If such travel is not possible, output "cannot be travelled" instead.

Sample Input	Sample Output
2 0022 0001 0102 0212 1222 -1 3 111123 112113 113123 111110 110100 100000 -1 0	Maze #1 can be travelled Maze #2 cannot be travelled

Problem H : Parliament

(black balloon)

You are a member of the "Conservative Party" in the parliament. There is one more party called the "Reformist Party". Other members of the parliament are independents and do not belong to either of these two parties. When a bill is put for vote, each member of the two parties has to vote as being decided by his/her party. But an independent member makes his/her mind on how to vote based on different lobbying that always exists. Note that parliament uses the majority rule (more than half of all members) to make decision.

Suppose a bill is intended to put for vote on the floor in the parliament and you know that the Reformist party is against the bill. You and your party-mates want to do lobbying on the independent members so that enough of them make or change their minds so that the bill is passed. Your task is to find the minimum number of independent members that have to vote as you wish so that the bill is passed.

Standard Input

There are multiple test cases in the input. Each test case appears in one line containing three space-separated integers n, m and k which respectively are the total number of members, the number of members in the Conservative Party and the number of members in the Reformist Party ($1 \leqslant n, m, k \leqslant 1000, m + k \leqslant n$). The input terminates with a line containing 0 0 0 which should not be processed.

Standard Output

For each test case, output a line containing the minimum number of independent members that have to vote as you wish so that the bill is passed. Output -1, if there is no way to pass the bill.

Sample Input	Sample Output
8 3 3	2
12 4 6	-1
7 4 3	0
0 0 0	

Problem I : Image processing

(maroon balloon)

In a crowded city, a network of traffic cameras are installed on all crosses and capture the image of the cars that pass the red lights. The law enforcement department requires a software system to process all the captured numbers, and issue a ticket for all the cars that violate the law. An image processing module has already preprocessed the video recordings and generated a file containing the recorded car numbers. You are requested to write the ticketing module. The problem is that the image processing module is not perfect, and some car numbers are wrong.

A true car number should meet all of the following requirements:

- It is 8 characters long.

- The two leftmost characters are identical digits between 1 to 9, which indicate in which city the car number is issued.

- The following two characters are two digits between 1 to 9.

- The following character is a capital English letter.

- The three rightmost characters are also digits between 1 to 9.

Standard Input

The first line of input contains the number of captured car numbers (n) that are reported by the image processing module. In each of the next n lines, there are exactly 8 numerical or English alphabetical characters, showing one inferred car number. You can assume $1 \leqslant n \leqslant 1000$.

Standard Output

You should report the car numbers that have violated the law, one in each line, in the same order that they appear in the input. If the same car number has violated the law multiple times, all of the violation cases should be reported. The wrong car numbers, which do not meet one or more of the above requirements, should not be listed in the output.

Sample Input	Sample Output
5 8835R551 4352S132 2241X223 55123456 9914t521	8835R551 2241X223