



Home

[What's New](#)

[2000 Results](#)

[ACM](#)

[Membership](#)

[ACM Chapters](#)

[IBM Sponsor Site](#)

[Upsilon Pi Epsilon](#)

[Other Contests](#)

Regional Contest Info Finder...



The 2000 Regionals are over! [Check the results!](#) Advance registration for 2001 begins March 15, 2001.

Join us for the challenge, camaraderie, and the fun!

...for participants in the Central, Northeastern, and Southeastern ACM Regional Programming Contests!

If you participated in one of these contests, check out this special offer of [free ACM student membership!](#) The offer expires soon so don't delay.

Coming March 7-11, 2001...

The 25th Annual ACM International Collegiate Programming Contest World Finals sponsored by IBM!!

From a field of thousands of teams competing at 89 university sites on six continents, sixty-four teams will advance to the 2001 ACM World Finals to be held in Vancouver, B.C., Canada. Scholarships, prizes, and bragging rights are at stake for some of the world's finest university students of computing.



Problem A – *u* Calculate *e*

Background

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*.

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.

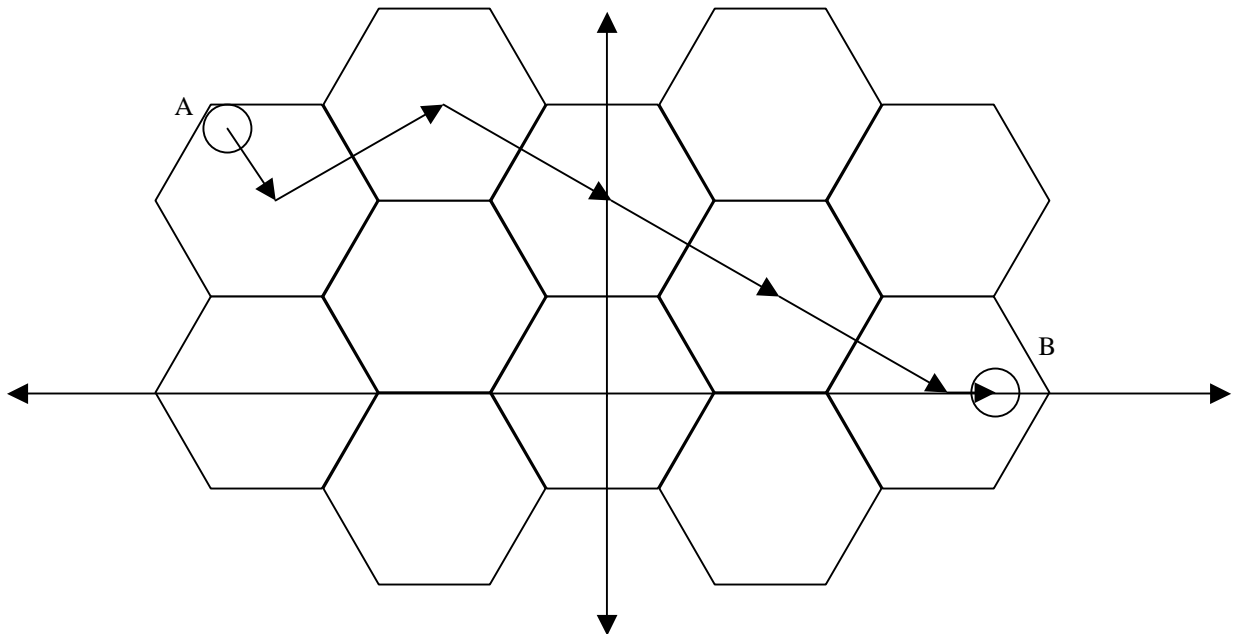
Example

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

Problem Bee

Background

Imagine a perfectly formed honeycomb, spanning the infinite Cartesian plane. It is an interlocking grid composed of congruent equilateral hexagons. One hexagon is located so that its center is at the origin and has two corners on the X-axis. A bee must be very careful about how it travels in order not to get lost in the infinite plane. To get from an arbitrary point A to another arbitrary point B, it will first head from A to the exact center of the hexagon in which A is located. Then, it will travel in a straight line to the exact center of an adjacent hexagon. It will move from center to adjacent center until it has reached the hexagon containing point B. At the destination hexagon, it will move from the center to point B. In all cases the bee will take a path of minimal distance that obeys the rules. The figure below demonstrates one possible minimal path from point A to point B.



Input

Input will be in the form of 5 floating point numbers per line. The first number will be the length, in centimeters, of the sides of the hexagons. The next two numbers will be the x and y coordinates of point A, followed by the x and y coordinates for point B. The input will be terminated by a line containing five zeroes. Neither point A nor point B will ever be exactly on a border between hexagons.

Output

For each line of the input, output the minimum length of a path from point A to point B, in centimeters, to the nearest .001 centimeters.

Example

Input	Output
1.0 -3.2 2.2 3.3 0	7.737
9 1 4 5 1	5.000
0.1 .09 0 .21 0	0.526
0 0 0 0 0	

Problem C – Digital Roots

Background

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.

Input

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

Output

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

Example

Input	Output
24	6
39	3
0	

Problem D – Scramble Sort

Background

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 n^{th} element in the list is a number it must remain a number, and if it is a word it must remain a word.

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.

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.

Example

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.
.
```

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 – A Well-Formed Problem

Background

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.**

Input

The input file 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.

Output

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

Example

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-list>
<?end?>
```

Output

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

Problem F – Entropy

Background

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.



Sponsored by IBM®

Input

The input file 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.

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.

Example

Input	Output
AAAAABCD	64 13 4.9
THE_CAT_IN_THE_HAT	144 51 2.8
END	

Problem G – N-Credible Mazes

Background

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.

Input

The input file 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 $2n$ 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 $2n$ 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 file. The end of the input is signalled by space dimension of zero. No further data will follow this terminating zero.

Output

For each maze output its 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.

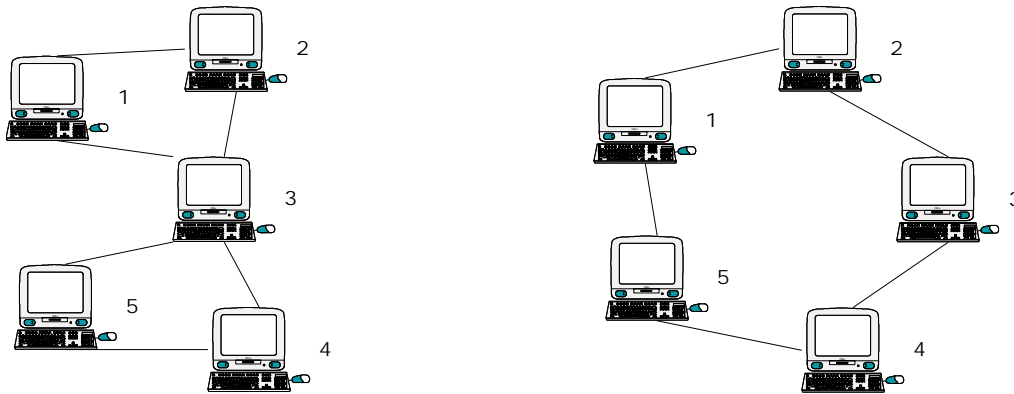
Example

Input	Output
<pre>2 0 0 2 2 0 0 0 1 0 1 0 2 0 2 1 2 1 2 2 2 -1 3 1 1 1 1 2 3 1 1 2 1 1 3 1 1 3 1 2 3 1 1 1 1 1 0 1 1 0 1 0 0 1 0 0 0 0 0 -1 0</pre>	<pre>Maze #1 can be travelled Maze #2 cannot be travelled</pre>

Problem H – SPF

Background

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.



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 file should be ignored.

Output

For each network in the input, you will output its number in the file, followed by a list of any SPF nodes that exist. The first network in the file 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.



Example

Input	Output
1 2 5 4 3 1 3 2 3 4 3 5 0	Network #1 SPF node 3 leaves 2 subnets
1 2 2 3 3 4 4 5 5 1 0	Network #2 No SPF nodes
1 2 2 3 3 4 4 6 6 3 2 5 5 1 0	Network #3 SPF node 2 leaves 2 subnets SPF node 3 leaves 2 subnets
0	