Univerza v Ljubljani

Prvenstvo
UNIVERZE V LJUBLJANI
v
programiranju

2002

finale

Pokrovitelji:

UNIVERZA V LJUBLJANI
Fakulteta za matematiko in fiziko

NAVODILA

Prijava: Prijavite se lahko le na en računalnik. Če bodo med tekmovanjem z njim slučajno težave, javite vodstvu tekmovanja. V primeru, da boste hkrati prijavljeni na dveh računalnikih, bo ekipa diskvalificirana.

Uporabniško ime je sestavljeno iz besede acm in številke računalnika.

Primer: uporabniško ime na računalniku št. 1 je: acm01

uporabniško ime na računalniku št. 12 je: acm12

Geslo je sporočeno pred začetkom tekmovanja.

Vaše domače področje na disku je `/home/acmXX` kjer je XX številka računalnika

Priprava delovnega okolja

Po uspešni prijavi morate odpreti dva terminala in program Netscape.

Terminal št.1: v njem boste pisali programe, prevajali programe in pošiljali programe sodniku.

Terminal št.2: v njem boste dobivali obvestila o pravilnosti rešitev in pošiljali morebitna vprašanja sodniku.

Program Netscape: bo omogočal pregled trenutne uspešnosti ekip

Pisanje programov:

Na voljo so urejevalniki besedil: *vi, joe, jed, mcedit, emacs*, *gedit in kedit*

Ime programov je točno določeno pri opisu nalog. Pazite na `return 0,` ki mora zaključiti vsak uspešno izveden C program.

Prevajanje programov:

Za prevajanje programov morate uporabljati naslednje prevajalnike:

```
C       (*.c)         gcc    primer:   gcc -o program program.c
C++     (*.cpp)       g++    primer:   g++ -o program program.cpp
Pascal (*.pas)        gpc    primer:   gpc -o program program.pas
```

Pošiljanje programov:

Za pošiljanje programov sodniku uporabljate ukaz `submit`.

Primer: `submit program.c`

Sporočila o napakah – PE in WA

Zaradi različnih razlogov je zelo težko razlikovati med `Presentation Error` in `Wrong Answer`. Ker preverjanje poteka več ali manj avtomatsko, program pogosto ne zna razlikovati med tem, ali ste le pozabili piko v odgovoru, ali pa ste napačno zračunali. Zato tudi `Wrong Answer` včasih lahko pomeni

le manjkajoč znak v odgovoru. Praviloma naj bi sporočilo `PE` dobili le v primeru napačno postavljenih presledkov ali ločil in praznih vrstic, ni pa to nujno!

Komunikacija sodniki – ekipa:

V terminalu 2 izvedete ukaz: `telnet svarun`

Za prijavo na računalnik svarun uporabite isto uporabniško ime in geslo, kot ste ga uporabili za prijavo na vaš računalnik. V tem terminalskem oknu se bodo izpisovala sodnikova sporočila vam. Če želite poslati sporočilo sodniku, morate v tem terminalu napisati ukaz:

```
write sodnik
```

in potem sporočilo, ki ga zaključite s kombinacijo tipk CTRL + D na začetku prazne vrstice. Prosimo, da **ne** pošiljate sporočil tipa `... Sedaj smo poslali program ...` in `... Naš program zagotovo dela, pa dobimo WA ...` Ekipa, ki bo pošiljala tovrstna sporočila, je lahko tudi diskvalificirana.

Trenutni rezultati:

Program Netscape: bo omogočal pregled trenutne uspešnosti ekip

Po zagonu programa greste na naslov:

http://svarun.fmf.uni-lj.si/index.html

Kadar želite videti trenutno stanje, morate narediti »reload«.

Pritožbe:

So možne le takoj po zaključku tekmovanja. Kasneje se rezultati lahko spremenijo le v primeru drastične sodniške napake.

Preklapljanje med angleško in slovensko tipkovnico:

(Velja samo za okolje X windows)

Prednastavljena je slovenska tipkovnica. Za preklop na angleško tipkovnico v terminalu izvedete ukaz

```
setxkbmap us
```

za preklop nazaj na slovensko pa:

```
setxkbmap sl
```

Izbrana tipkovnica potem velja za celotno X okolje.

Syntax

Program SYNTAX.C, SYNTAX.CPP, SYNTAX.PAS

In the land of Hedonia the official language is Hedonian. A Hedonian professor had noticed that many of her students still did not master the syntax of Hedonian well. Tired of correcting the many syntactical mistakes, she decided to challenge the students and asked them to write a program that could check the syntactical correctness of any sentence they wrote. Similar to the nature of Hedonians, the syntax of Hedonian is also pleasantly simple. Here are the rules:

0. The only characters in the language are the characters p through z and N, C, D, E, and I.

1. Every character from p through z is a correct sentence.

2. If s is a correct sentence, then so is Ns.

3. If s and t are correct sentences, then so are Cst, Dst, Est, and Ist.

4. Rules 0. to 3. are the only rules to determine the syntactical correctness of a sentence.

You are asked to write a program that checks if sentences satisfy the syntax rules given in Rule 0. - Rule 4.

Input

The input consists of a number of sentences. Each sentence is ended by a new-line character. The collection of sentences is terminated by the end-of-file character. If necessary, you may assume that each sentence has at most 256 characters and at least 1 character.

Output:

The output consists of the answers YES for each well-formed sentence and NO for each not-well-formed sentence. The answers are given in the same order as the sentences. Each answer is followed by a new-line character, and the list of answers is followed by an end-of-file character.

Sample Input

```
Cp
Isz
NIsz
Cqpq
```

Sample Output:

```
NO
YES
YES
NO
```

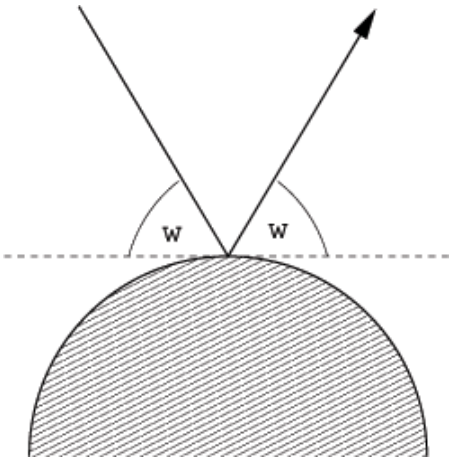
Reflections

Program REFLECT.C, REFLECT.CPP, REFLECT.PAS

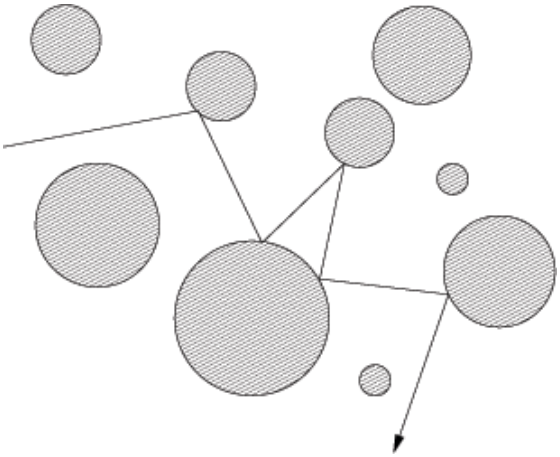
Rendering realistic images of imaginary environments or objects is an interesting topic in computer graphics. One of the most popular methods for this purpose is ray-tracing.

To render images using ray-tracing, one computes (traces) the path that rays of light entering a scene will take. We ask you to write a program that computes such paths in a restricted environment.

For simplicity, we will consider only two-dimensional scenes. All objects in the scene are totally reflective (mirror) spheres. When a ray of light hits such a sphere, it is reflected such that the angle of the incoming ray and the leaving ray against the tangent are the same:



The following figure shows a typical path that a ray of light may take in such a scene:



Your task is to write a program, that given a scene description and a ray entering the scene, determines which spheres are hit by the ray.

Input

The input consists of a series of scene descriptions. Each description starts with a line containing the number n (n <= 25) of spheres in the scene. The following n lines contain three integers x_i, y_i, r_i each, where (x_i, y_i) is the center, and $r_i > 0$ is the radius of the i-th sphere. Following this is a line containing four integers x, y, d_x, d_y, which describe the ray. The ray originates from the point (x, y) and initially points in the direction (d_x, d_y). At least one of d_x and d_y will be non-zero.

The spheres will be disjoint and non-touching. The ray will not start within a sphere, and never touch a sphere tangentially.

A test case starting with n = 0 terminates the input. This case should not be processed.

Output

For each scene first output the number of the scene. Then print the numbers of the spheres that the ray hits in its first ten deflections (the numbering of spheres is according to their order in the input).

If the ray hits at most ten spheres (and then heads towards infinity), print `inf` after the last sphere it hits. If the ray hits more than 10 spheres, print three points (`...`) after the tenth sphere.

Output a blank line after each test case.

Sample Input

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

Sample Output

```
Scene 1
1 2 1 3 inf

Scene 2
2 1 2 1 2 1 2 1 2 1 ...
```

Superior Search

Program SEARCH.C, SEARCH.CPP, SEARCH.PAS

Clearly the economy is bound to pick up again soon. As a forward-thinking Internet entrepreneur, you think that the 'Net will need a new search engine to serve all the people buying new computers. Because you're frustrated with the poor results most search engines produce, your search engine will be better.

You've come up with what you believe is an innovative approach to document matching. By giving weight to the number of times a term appears in both the search string and in the document being checked, you believe you can produce a more accurate search result.

Your program will be given a search string, followed by a set of documents. You will calculate the score for each document and print it to output in the order the document appears in the input file. To calculate the score for a document you must first calculate the term score for each term appearing in the search string. A term score is the number of times a term occurs in the search string multiplied by the number of times it occurs in the document. The document score is the sum of the square roots of each term score.

Input

The input file consists of a set of documents separated by single lines containing only ten dashes, `"----------"`. No line will be longer than 250 characters. No document will be longer than 100 lines. The first document is the search string. The input file terminates with two lines of ten dashes in a row.

The input documents will use the full ASCII character set. You must parse each document into a set of terms.

Terms are separated by whitespace in the input document. Comparisons between terms are case-insensitive. Punctuation is removed from terms prior to comparisons, e.g. “`don't`” becomes “`dont`”. The resulting terms should contain only the characters {[a-z],[0- 9]}. A term in the input consisting only of punctuation should be ignored. You may assume the search string and each document will have at least one valid term.

Output

The output is a series of scores, one per line, rounded to two decimal places. The scores are printed in the order the documents occur in the input. No other characters may appear in the output.

Sample Input

```
fee fi fo fum
----------
fee, fi, fo! fum!!
----------
fee fee fi, me me me
----------
----------
```

Sample Output

```
4.00
2.41
```

Continued Fractions

Program FRACTION.C, FRACTION.CPP, FRACTION.PAS

Let b_0, b_1, b_2,..., b_n be integers with $b_k > 0$ for $k > 0$. The *continued fraction* of order n with coeficients b_1, b_2,..., b_n and the initial term b_0 is defined by the following expression

$$b_0 + \cfrac{1}{b_1 + \cfrac{1}{b_{2+\cdots+\frac{1}{b_n}}}}$$

which can be abbreviated as $[b_0;b_1,\ldots, b_n]$.

An example of a continued fraction of order $n = 3$ is [2;3, 1, 4]. This is equivalent to

$$2 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{4}}} = \frac{43}{19}$$

Write a program that determines the expansion of a given rational number as a continued fraction. To ensure uniqueness, make $b_n > 1$.

Input

The input consists of an undetermined number of rational numbers. Each rational number is defined by two integers, numerator and denominator.

Output

For each rational number given in the input, you should output the corresponding continued fraction.

Sample Input

```
43 19
1 2
```

Sample Output

```
[2;3,1,4]
[0;2]
```

Crypt Kicker II

Program CRYPT.C, CRYPT.CPP, CRYPT.PAS

A common but insecure method of encrypting text is to permute the letters of the alphabet. That is, in the text, each letter of the alphabet is consistently replaced by some other letter. So as to ensure that the encryption is reversible, no two letters are replaced by the same letter.

A common method of cryptanalysis is the known plaintext attack. In a known plaintext attack, the cryptanalist manages to have a known phrase or sentence encrypted by the enemy, and by observing the encrypted text then deduces the method of encoding.

Your task is to decrypt several encrypted lines of text, assuming that each line uses the same set of replacements, and that one of the lines of input is the encrypted form of the plaintext
`the quick brown fox jumps over the lazy dog`

Input

The input begins with a single positive integer on a line by itself indicating the number of the cases following, each of them as described below. This line is followed by a blank line, and there is also a blank line between two consecutive inputs.

The input consists of several lines of input. Each line is encrypted as described above. The encrypted lines contain only lower case letters and spaces and do not exceed 80 characters in length. There are at most 100 input lines.

Output

For each test case, the output must follow the description below. The outputs of two consecutive cases will be separated by a blank line.

Decrypt each line and print it to standard output. If there is more than one possible decryption (several lines can be decoded to the key sentence), use the first line found for decoding. If decryption is impossible, output a single line "`No solution.`"

Sample Input

```
1

vtz ud xnm xugm itr pyy jttk gmv xt otgm xt xnm puk ti xnm fprxq
xnm ceuob lrtzv ita hegfd tsmr xnm ypwq ktj
frtjrpgguvj otvxmdxd prm iev prmvx xnmq
```

Sample Output

```
now is the time for all good men to come to the aid of the party
the quick brown fox jumps over the lazy dog
programming contests are fun arent they
```

John's Tasks

Program TASKS.C, TASKS.CPP, TASKS.PAS

John's boss likes to give his employees a lot of tasks at once, to save himself the trouble of leaving his office too many times. Unfortunately, the tasks he gives are not independent so the execution of one task is only possible if some other tasks have already been executed. Of course the boss won't bother to sort the tasks... therefore John must do it on his own. To make things even more complicated, it's sometimes impossible to sort the tasks (because one task depends on another which again depends on the first).

Input

The input will consist of several instances of the problem. Each instance begins with a line containing two integers, **1 <= n <= 100** and **m**. **n** is the number of tasks (numbered from **1** to **n**) and **m** is the number of direct precedence relations between tasks. After this, there will be **m** lines with two integers **i** and **j**, representing the fact that task **i** must be executed before task **j**. An instance with **n = m = 0** will finish the input.

Output

For each instance, print a line with n integers representing the tasks in a possible order of execution. If there's more than one solution, print lexicographicaly smallest one. If it's impossible to find a good ordering, print "`No solution.`"

Sample Input

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

Sample Output

```
1 2 3 4 5
No solution.
```

Poker Hands

Program POKER.C, POKER.CPP, POKER.PAS

A poker deck contains **52** cards - each card has a suit which is one of clubs, diamonds, hearts, or spades (denoted **C**, **D**, **H**, and **S** in the input data). Each card also has a value which is one of **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **10**, **jack**, **queen**, **king**, **ace** (denoted **2, 3, 4, 5, 6, 7, 8, 9, T, J, Q, K, A**). For scoring purposes, the suits are unordered while the values are ordered as given above, with 2 being the lowest and ace the highest value.

A poker hand consists of **5** cards dealt from the deck. Poker hands are ranked by the following partial order from lowest to highest

- **High Card:** Hands which do not fit any higher category are ranked by the value of their highest card. If the highest cards have the same value, the hands are ranked by the next highest, and so on.
- **Pair: 2** of the **5** cards in the hand have the same value. Hands which both contain a pair are ranked by the value of the cards forming the pair. If these values are the same, the hands are ranked by the values of the cards not forming the pair, in decreasing order.
- **Two Pairs:** The hand contains **2** different pairs. Hands which both contain **2** pairs are ranked by the value of their highest pair. Hands with the same highest pair are ranked by the value of their other pair. If these values are the same the hands are ranked by the value of the remaining card.
- **Three of a Kind:** Three of the cards in the hand have the same value. Hands which both contain three of a kind are ranked by the value of the **3** cards.
- **Straight:** Hand contains **5** cards with consecutive values. Hands which both contain a straight are ranked by their highest card.
- **Flush:** Hand contains **5** cards of the same suit. Hands which are both flushes are ranked using the rules for High Card.
- **Full House: 3** cards of the same value, with the remaining **2** cards forming a pair. Ranked by the value of the **3** cards.
- **Four of a kind: 4** cards with the same value. Ranked by the value of the **4** cards.
- **Straight flush: 5** cards of the same suit with consecutive values. Ranked by the highest card in the hand.

Your job is to compare several pairs of poker hands and to indicate which, if either, has a higher rank.

Input

The input file contains several lines, each containing the designation of **10** cards: the first **5** cards are the hand for the player named **"Black"** and the next **5** cards are the hand for the player named **"White"**.

Output

For each line of input, print a line containing one of the following three lines:

```
Black wins.
White wins.
Tie.
```

Sample Input

```
2H 3D 5S 9C KD 2C 3H 4S 8C AH
2H 4S 4C 2D 4H 2S 8S AS QS 3S
2H 3D 5S 9C KD 2C 3H 4S 8C KH
2H 3D 5S 9C KD 2D 3H 5C 9S KH
```

Sample Output

```
White wins.
Black wins.
Black wins.
Tie.
```

Frame Stacking

Program FRAMES.C, FRAMES.CPP, FRAMES.PAS

Consider the following 5 picture frames placed on an 9 x 8 array.

```
........   ........   ........   ........   .CCC....
EEEEEE..   ........   ........   ..BBBB..   .C.C....
E....E..   DDDDDD..   ........   ..B..B..   .C.C....
E....E..   D....D..   ........   ..B..B..   .CCC....
E....E..   D....D..   ....AAAA   ..B..B..   ........
E....E..   D....D..   ....A..A   ..BBBB..   ........
E....E..   DDDDDD..   ....A..A   ........   ........
E....E..   ........   ....AAAA   ........   ........
EEEEEE..   ........   ........   ........   ........
   1          2          3          4          5
```

Now place them on top of one another starting with 1 at the bottom and ending up with 5 on top. If any part of a frame covers another it hides that part of the frame below. Viewing the stack of 5 frames we see the following.

```
.CCC....
ECBCBB..
DCBCDB..
DCCC.B..
D.B.ABAA
D.BBBB.A
DDDDAD.A
E...AAAA
EEEEEE..
```

In what order are the frames stacked from bottom to top? The answer is EDABC. Your problem is to determine the order in which the frames are stacked from bottom to top given a picture of the stacked frames. Here are the rules:

1. The width of the frame is always exactly 1 character and the sides are never shorter than 3 characters.
2. It is possible to see at least one part of each of the four sides of a frame. A corner shows two sides.
3. The frames will be lettered with capital letters, and no two frames will be assigned the same letter.

Input

Each input block contains the height, h (h<=30) on the first line and the width w (w<=30) on the second. A picture of the stacked frames is then given as h strings with w characters each.

Input may contain multiple blocks of the format described above, without any blank lines in between. All blocks in the input must be processed sequentially.

Output

Write the solution to the standard output. Give the letters of the frames in the order they were stacked from bottom to top. If there are multiple possibilities for an ordering, list all such possibilities in alphabetical order, each one on a separate line. There will always be at least one legal ordering for each input block. List the output for all blocks in the input sequentially. Leave one blank line after each blocks.

Sample input

```
9
8
.CCC....
ECBCBB..
DCBCDB..
DCCC.B..
D.B.ABAA
D.BBBB.A
DDDDAD.A
E...AAAA
EEEEEE..
```

Sample Output

```
EDABC
```

Keeps Going and Going and...

Program GOING.C, GOING.CPP, GOING.PAS

Lazy functional languages like Haskell and Miranda support features that are not found in other programming languages, including infinite lists. Consider the following simple (and useful) recursive declaration:

```
letrec
  count n = cons n (count (n+1))
in
  count 0
```

The function cons constructs lists, so the above declaration creates the following structure:

```
  cons 0 (count 1)
= cons 0 (cons 1 (count 2))
= cons 0 (cons 1 (cons 2 ...))
= [0,1,2,...]
```

Lazy languages can do this because they only evaluate expressions that are actually used. If a program creates an infinite list and only looks at items 2 and 3 in it, the values in positions 0 and 1 are never evaluated and the list structure is only evaluated so far as the fourth node.

It is also possible to use more than one function to build an infinite list. Here is a declaration that creates the list `[''even'',''odd'',''even'',...]`:

```
letrec
  even = cons ''even'' odd
  odd = cons ''odd'' even
in
  even
```

There are also functions that manipulate infinite lists. The functions take and drop can be used to remove elements from the start of the list, returning the (removed) front elements or the remainder of the list, respectively. Another useful function is zip, which combines two lists like the slider on a zipper combines the teeth. For example,

```
zip (count 0) (count 10) = [0,10,1,11,2,12,...]
```

Your task is to implement a subset of this functionality.

Input

The first line of input consists of two positive integers, *n* and *m*. *n* is the number of declarations to follow and *m* is the number of test cases.

Each declaration takes the form *name* = *expr*. There are two forms of *expr*: **zip** *name1 name2* and $x_0\ x_1\ ...\ x_i$ *name3*. In the first case, *name* is the result of zipping *name1* and *name2* together. The other case defines the first $i + 1$ non-negative integers in the list name (where i is at least 0) and *name3* is the name of the list that continues it (mandatory - all lists will be infinite).

The test cases take the form *name s e*, where *s* and *e* are non-negative integers, $s <= e$ and $e - s < 250$.

No line of input will be longer than 80 characters. Names consist of a single capital letter.

Output

For each test case, print the integers in positions *s* to *e* of the list name. List elements are numbered starting with 0.

Sample Input

```
5 3
S = 4 3 2 1 A
O = 1 O
E = 0 E
A = zip E O
Z = zip Z S
A 43455436 43455438
S 2 5
Z 1 10
```

Sample Output

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