Problem SP2000-A Most wanted word

Write a program to find the most frequent word in a file of text. A word is any non-empty continuous sequence of alphabetic characters. Case is not regarded as significant, so the words "bother" and "BOTHeR" should be considered the same word.

Any non-alphabetic characters (including control characters such as newlines) can be used to separate words. Thus "isn't" is counted as two words, the second of which is a one letter word consisting only of the letter "t".

If more than one word appears with the maximum frequency, then the first word to reach the maximum frequency is required.

Input will consist of a set of paragraphs each terminated by a single line containing only the character '#' which will not otherwise occur anywhere in the text. No paragraph will contain more than 2000 different words, and no word will be more than 20 letters long. The input is terminated by a test case with no words at all. This case should produce no output.

For each paragraph, your program should print one line containing the frequency of the most frequent word, right justified in a field of width 4, followed by a space and the most frequently occurring word itself, entirely in lower case.

Sample Input

```
This is a simple file of test data, which should
not cause your program any trouble. Do note that
it contains several punctuation characters. Of course,
this is not really a problem, because such characters
are treated in the same way as spaces.
#
Don't use contractions; it isn't nice.
#
aBc def AbC def dfe ABC
#
123
#
```

Sample Output

```
   2 of
   2 t
   3 abc
```

Problem SP2000-B Tag Checker

Markup languages such as HTML use tags to highlight sections with special significance. In this way, a sentence in boldface can be indicated thus:
<B>This is a sentence in boldface</B>

Typically every tag has an opening tag of the form <TAG> and a closing tag of the form </TAG>, so that portions of text can be bracketed as above. Tags can then be combined to achieve more than one effect on a particular piece of text simply by nesting them properly, for instance:
<CENTER><B>This text is centred and in boldface</B></CENTER>

Two of the most common mistakes when tagging text are:

- getting the nesting wrong:
  <B><CENTER>This should be centred boldface, but the tags are wrongly nested</B></CENTER>
- forgetting a tag:
  <B><CENTER>This should be centred boldface, but there is a missing tag</CENTER>

Write a program to check that all the tags in a given piece of text (a paragraph) are correctly nested,and that there are no missing or extra tags. An opening tag for this problem is enclosed by angle brackets, and contains exactly one upper case letter, for example <T>, <X>, <S>. The corresponding closing tag will be the same letter preceded by the symbol /; for the examples above these would be </T>, </X>, </S>.

The input will consist of any number of paragraphs. Each paragraph will consist of a sequence of tagged sentences, over as many lines as necessary, and terminating with a # which will not occur elsewhere in the text. The input will never break a tag between two lines and no line will be longer than 80 characters. The input will be terminated by an empty paragraph, i.e. a line containing only a single #.

If the paragraph is correctly tagged then output the line “Correctly tagged paragraph”, otherwise output a line of the form “Expected <expected> found <unexpected>” where <expected> is the closing tag matching the most recent unmatched tag and <unexpected> is the closing tag encountered. If either of these is the end of paragraph, i.e. there is either an unmatched opening tag or no matching closing tag at the end of the paragraph, then replace the tag or closing tag with #. These points are illustrated in the examples below which should be followed exactly as far as spacing is concerned.

Sample Input

```
The following text<C><B>is centred and in boldface</B></C>#
<B>This <\g>is <B>boldface</B> in <<*> a</B> <\6> <<d>sentence#
<B><C> This should be centred and in boldface, but the
tags are wrongly nested </B></C>#
<B>This should be in boldface, but there is an extra closing
tag</B></C>#
<B><C>This should be centred and in boldface, but there is
a missing closing tag</C>#
#
```

Sample Output

```
Correctly tagged paragraph
Correctly tagged paragraph
Expected </C> found </B>
Expected # found </C>
Expected </B> found #
```

Problem SP2000-C Bouncy Balls

The Department of Defence of a certain country (No, not Australia or New Zealand), in conjunction with the Department of Police, have devised a brilliant method of saving money on crowd control — really bouncy rubber bullets. They had noticed that the rubber bullets they had been using were largely being wasted — those that hit anyone or anything usually just fell to the ground, whereas if they were really, really bouncy, they would bounce off and possibly hit several more people before their energy was spent.

They decided to test this idea by building a special circular test rig. The bullet would be fired into the rig horizontally and at some predetermined angle to the tangent to the rig at that point. It would have sufficient energy to travel some considerable distance before stopping. (You may assume cartoon physics, i.e. that it travels horizontally until it reaches the end of its travel, at which time it drops to the floor.) However, as so often happens with lucrative defence contracts, the contractor made off with the money, so they decided to simulate the whole process on a computer. This is where you come in.

Write a program that will read in details of a test rig and a series of test firings and determine how many times the rubber bullet would bounce before it stops. You may assume that the bullet is a point and that, because of problems in determining the exact sequence of events, any test firing where the bullet stops within 1 mm of the rig is deleted from consideration.

Input will be a series of tests, each test consisting of a series of test firings. Each test starts with an integer specifying the radius of the test rig in millimetres and a value of 0 for the radius terminates the input. Each test firing occurs on a line by itself and consists of a distance in millimetres (between 100 and 10000 inclusive) that the bullet will travel, and an angle, in degrees, (between 10 and 170 inclusive, where 90 means directly towards the centre of the rig). The series of test firings will be terminated by a line containing two zeroes (0 0).

For each test rig, output a line with the words “Test Rig” followed by a space and then the number of the test rig (a running number starting at 1) followed by a series of lines, one for each test firing for that rig with each line giving the number of times a bullet bounces off a wall before it stops. This number is to be written without any leading or trailing spaces. A blank line should appear between test rigs. Follow the example given below.

Sample Input

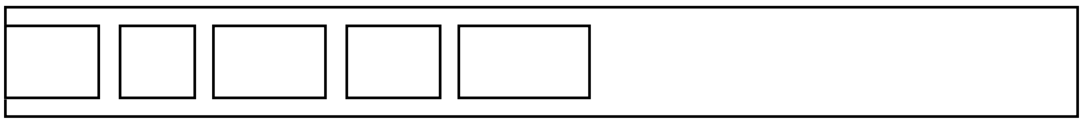
```
100
1000 23
1200 47
0 0
0
```

Sample Output

```
Test Rig 1
12
8
```

Problem SP2000-D Best Image Layout

Lazy Larry’s Leisure Photo Shop specialises in taking shots of people in leisure situations — at the beach, skiing, in cable cars and so on. When the photographer returns to the office at the end of a shoot, she has anything up to 50 images which she wishes to display. She has found that the most practical solution is to lay them out in a single row in a strip across the top of the screen with the first image hard up against the left of the strip, the rest of the images in order from left to right, and the last image as far right as necessary or possible.



The images are all of the same height, which comfortably fits into the strip of screen space, but have varying widths. The photographer can vary the width of the strip within limits, largely dictated by the size of the screen she is working on and other applications that may be running. When there is enough room, the images should be packed from left to right with a fixed spacing. When there is not enough room the spacing has to be reduced until it reaches zero. Thereafter the images must be allowed to overlap.

Write a program that will read in details of the size of the strip and a series of images and that will determine the best layout according to the following rules:

1. Pack the images starting at the left margin as shown in the diagram above with *SPACING* pixels between images. This is the ideal case.
2. Place the first image hard up against the left edge of the strip, the last image hard up against the right edge of the strip and then place the left edges of the other images on the pixel boundaries closest to the positions that they would occupy were it possible to contract the spacing in a smooth continuous manner. Round 0.5 up to the next integer.
3. Place the last image hard up against the right edge of the strip and overlap the remaining images so that their left boundaries remain in order from left to right. Allocate each image as many pixels as possible so that all images have the same proportion (truncated to one pixel) of their width visible. If there are still pixels left to be allocated, allocate them to images in descending order of the fractional portions of their desired display width. In case of a tie, allocate the ‘extra’ pixel to the leftmost image.
4. Placement is impossible if one or more images are not displayable, i.e. do not have at least one pixel allocated to them.

Input will be a sequence of problems, terminated by a line of three (3) zeroes (0 0 0). The first line of each problem has three positive integers specifying *WIDTH*, *SPACING*, and *NIMAGES*. *WIDTH* is the width of the available screen area in pixels, *SPACING* is the maximum spacing to be used between images and *NIMAGES* is the number of images to be placed. This is followed by *NIMAGES* integers specifying, in order, the widths of the images, on one or more lines. You may assume that *SPACING* ≤ 10, that 0 < *NIMAGES* ≤ 50 and that the image widths range from 10 pixels to no more than *WIDTH*/2.

Output for each problem is a line with the words ‘Problem number’ followed by a space and the problem number followed by a line containing either the word ‘IMPOSSIBLE’ if placement is impossible, or *NIMAGES* integers, separated by single spaces, specifying the position at which the leftmost pixel of each image (in the order given in the input) should be placed. Pixels are numbered from 0 to *WIDTH-1*.

Sample Input

```
380 8 5
35 28 43 35 5
0 0 0
```

Sample Output

```
Problem number 1
0 43 79 130 173
```

Problem SP2000-E Swiss Draw

Many sports and games hold tournaments to determine at least a winner and, very often, a ranking or ordering as well. In two player games (such as Tennis, Chess and Scrabble), the two most common forms of tournament are‘knockout’ (usually based on an initial ranking or ‘seeding’) and ‘round robin’ (where everybody plays everybody else). The disadvantage in knockout is that a promising newcomer could meet a very much stronger player early in the tournament and not reach their true position. Round Robin eliminates this but at a huge cost in time — a Round Robin involving 128 players needs 127 rounds whereas it would take only 7 rounds in a knockout competition.

An alternative known as Swiss Draw is very popular in games such as Scrabble. To maximize competition, any one player will play any other player no more than once. After each round, players are ranked on the number of games they have won, where a draw is equal to half a win (more is better) and, within that, by ‘spread’ — the cumulative difference between their scores and their opponents’ scores (again bigger is better). If by chance two or more players tie in this ranking then they appear in inverse order of their previous ranking, i.e. the initially lower-ranked players move ahead. In each round each player either plays someone above them or the highest ranked player below them that allows everyone to play someone they have not played before. The input will specify the (usually random) ordering before the first game.

Write a program to determine the final ranking of a group of Scrabble players, given the initial draw and the scores for each individual for each round.

Input will consist of one or more scenarios. The first line of each scenario will consist of two integers, P and R, ($16 \le P \le 64$, $4 \le R \le P/4$) denoting the number of players (a multiple of two) and the number of rounds respectively. This will be followed by P lines, each line consisting of a name (a string of 1 through 20 alphabetic characters without any spaces) followed by R integers (separated from each other and the name by at least one space) representing the R scores for that individual. The list will be in the initial order of play, thus in the first round player 2n+1 played player 2n+2 ($0 \le n < P / 2$). Input will be terminated by a line containing two zeroes (i.e. P and R both zero).

Output will consist of a list of all the players ranked according to the above criteria, together with the number of wins and the spread. Note that a draw is counted as half a win, so indicate an odd number of draws by a plus sign (+) after the number of wins. The name is left justified in a field of width 20, the number of wins is right justified in a field of width 3, specification of draws occupies 1 character position and the spread is right justified in a field of width 6. Leave one blank line between scenarios.

Sample Input

```
16 4
Absalom  280 334 319 426
Betsheba 374 514 459 417
Carolyne 318 415 445 481
Davidian 402 361 375 278
Eleanor  425 302 447 522
Frances  425 513 306 327
Gabriel  330 337 365 398
Hermione 539 254 442 450
Ishmael  485 305 540 522
Jeremiah 288 295 367 476
Kenneth  532 304 452 445
Laurence 426 437 260 474
Meredith 438 489 274 475
Nicholas 307 357 380 482
Octavia  426 498 305 497
Patricia 333 253 370 412
0 0
```

Sample Output

```
Ishmael                4    619
Meredith               3    247
Carolyne               3    186
Betsheba               3    158
Kenneth                3    126
Eleanor                2+   -11
Hermione               2    264
Laurence               2    -93
Nicholas               2   -114
Davidian               2   -150
Frances                1+  -119
Octavia                1    -85
Absalom                1   -187
Jeremiah               1   -188
Gabriel                1   -338
Patricia               0   -315
```

Problem SP2000-F Calypso

Calypso is a fun-filled family card game that has been propagated at the training camps for the Australian Maths Olympiad teams. The game is designed for four players and will be described in that way, although I am sure you can adapt it to a different number. I will start with a complete description, since I suspect most of you will never have heard of it and you may want to play it some time (possibly with your coach as a fourth on the way to Vancouver next year).

The game is played with a standard deck of 52 cards where each card has a *suit*:– Spades (S), Hearts (H), Diamonds (D) or Clubs (C)— and a *value*:– 2–9, Jack (J), Queen (Q), King (K) and Ace (A).The cards are ranked (from lowest to highest) in the order given. One person is designated as the *dealer*, after each deal this role moves clockwise. To make the following description easier, assume that the four players are Amy, Bob, Carol and Dave and that they are sitting in that order clockwise around a table. Dave is the first dealer, Amy will deal next and so on.

Dave deals one card to each player face down, starting with Amy, and continues until all cards have been dealt (each player now has 13 cards). Amy now designates how many *tricks* she expects to take, a number between 0 and 13, (tricks will be explained later) and her personal *trump suit*, then Bob does likewise and so on. Dave as the last bidder must bid for a number of tricks such that the total number of tricks bid (by all four players) does not equal 13. After this has been done and recorded the play begins.

Amy then *leads* to the first trick, i.e. she lays a card face up on the table. Each player in clockwise then plays a card and the four cards constitute a trick. Each player **must** *follow suit* if possible, i.e. play a card of the same suit as that led. Players who cannot follow suit may play one of their personal trumps or just discard one of their other cards. The winner of the trick collects it and leads to the next trick. The winner is determined as follows. If a trick has not been trumped (the leader did not lead anyone’s personal trump suit and everyone either followed suit or discarded) then the highest card in the led suit wins. If the trick has been trumped, then the highest trump wins. If there are two or more trumps of equal value (remember that everyone has their own trump suit) then the first highest wins. For example, assume that at some stage Dave led a low diamond, that neither Amy or Bob have any diamonds left, and that Amy’s trump suit is Hearts and Bob’s is Spades. If Amy plays HK and Dave plays SA, he would win. If Amy had played HA, she would have won, even if Dave had still played SA. When all 13 tricks have been played, players score the number of tricks they won. In addition, players who achieved their target (won the number of tricks they bid for) score an extra 10 points.

In order to simulate this game, we will make the following modifications. Bids will consist of only the desired trump suit, based on length. If two or more suits are equally long, value the suits by allocating 5 points to an Ace, 4 points to a King, 3 points to a queen, two points to a Jack and 1 point to anything else and bid the highest valued suit. If there is still a tie, bid the highest ranked suit in the order (from highest to lowest) S, H, D, C. At the end of a deal each player will merely score the number of tricks they won, without any bonuses.

When leading, lead the highest card in your trump suit if possible, otherwise lead the highest card in the highest ranking suit that is not someone else’s trump suit. If this is not possible, lead the lowest ranked card in your hand (choosing the lowest ranked suit in case of a tie). When playing, always attempt to win the trick if you can, otherwise play as cheaply as possible. Thus you will play the highest card in the suit led, unless a higher card has already been played, in which case play your lowest card in that suit. If you cannot follow suit, play your highest trump if you can and if it could win. If you have no trumps, or your highest trump could not win the trick, then discard the lowest card in your hand. If you have two lowest cards, play the one from the lowest ranking suit (using the ranking given above).

Consider that the deck is (in order from the top of the deck to the bottom):

```
C8 HK D6 ST DT H5 S7 C9 DQ DK SA HA D2 S8 CT H8 SJ SQ S4 D8 D7 C3 SK
H6 HT H4 HQ S2 C6 C2 H9 DJ C7 CK CQ H2 CA DA CJ D5 S3 D3 S6 D9 H3 D4
S5 C4 H7 C5 HJ S9
```

Thus Dave will deal the C8 to Amy, HK to Bob, D6 to Carol and ST to himself and then continue. The resulting hands and bids are as follows:

```
  Amy  : C6 C7 C8 CA D2 D7 DT DQ H3 H7 HT S3 SJ  bids C
  Bob  : C2 C3 C5 CK D3 D4 DK DA H4 H5 HK S8 SQ  bids D
  Carol: CT CJ CQ D6 H9 HJ HQ S4 S5 S6 S7 SK SA  bids S
  Dave : C4 C9 D5 D8 D9 DJ H2 H6 H8 HA S2 S9 ST  bids H
```

Amy will lead to the first trick. In what follows, the leftmost card is the card led by the winner of the previous trick (indicated on the right of the previous line). The next three cards are the next three cards played. Note that the leftmost card is the card that was led, **not** necessarily the card played by Amy.

```
  Trick  1: CA C2 CT C4 Amy
  Trick  2: C8 C3 CJ C9 Amy
  Trick  3: C7 C5 CQ HA Dave
  Trick  4: H8 H3 H4 H9 Dave
  Trick  5: H6 H7 H5 HJ Dave
  Trick  6: H2 HT HK HQ Dave
  Trick  7: S2 SJ SQ SA Carol
  Trick  8: SK S9 S3 S8 Carol
  Trick  9: S7 ST D2 DA Bob
  Trick 10: DK D6 D5 D7 Bob
  Trick 11: D4 S6 D8 DT Carol
  Trick 12: S5 D9 C6 CK Amy
  Trick 13: DQ D3 S4 DJ Carol
```

Thus at the end of the deal Amy has won 3 tricks, Bob 2 and Carol and Dave 4 each.

Write a program to simulate playing this game. Input will consist of a series of decks (between 1 and 99, both numbers inclusive), each consisting of 4 lines of 13 cards without spaces as shown below and terminated by a line containing only '##'. For each deck in the input, output a line as shown below. After all decks have been processed output a summary line as shown.

Follow the spacing of the example exactly. The number of the deal is right justified in a field of width 3, the other numbers are right justified in fields of width 4.

Input

```
C8HKD6STDTH5S7C9DQDKSAHAD2
S8CTH8SJSQS4D8D7C3SKH6HTH4
HQS2C6C2H9DJC7CKCQH2CADACJ
D5S3D3S6D9H3D4S5C4H7C5HJS9
SAD3DTS4C9DAHJH6S8HACTD8H9
HTCQC5STHKHQS5D6C6D5H5H2H7
CKD7D4SJD2DQC3C8C4C2SQSKH3
DKD9H4CAH8S7CJS2S6C7DJS9S3
##
```

Output

```
Round  1:   3   2   4   4
Round  2:   4   3   3   3
            7   5   7   7
```

Problem SP2000-G Four in a Line

Four in a Line is a game similar to 3-dimensional noughts and crosses. It consists of a horizontal table on which 16 pegs, each of which can hold 4 beads, are arranged in a 4×4 grid. Each player has a supply of either green or red beads which are placed on the pegs in turn, starting with red. Obviously, as each bead is placed on a peg, it slides down as far as it can — until it either hits another bead or the supporting table. The winner is the first to get 4 beads of their colour in a line (hence the name). The line can be in any plane and in any orientation, as long as the four beads are all of the same colour and form a straight line.

As with most games, the interesting part comes towards the end, when each player (colour) is attempting to build a line and block the opponent's incipient lines. Write a program that will read in details of a game position and determine whether green (the next player) can be guaranteed to win the game within 5 *plies*. A *ply* is half a turn, in this situation placing one bead, thus 5 plies implies three moves by green and two by red.

Input consists of a number of games. Each game consists of 4 lines of characters, each line consisting of 4 blocks of 4 characters — 'R' for red, 'G' for green or '#' for empty — where each block represents the contents of a single peg with the left end representing the bottom. Thus the block 'GRR#' represents a peg with a green bead on the bottom with two red beads above it. Note that the entire state of the game is always given, thus the starting state would consist of 64 '#' characters arranged in 16 blocks of 4. You can assume that the position is valid, i.e. that there will be exactly one more red bead than green beads, and that there will not be any 'holes' in the description (the 'block' GR#G, for instance). There will be one blank line after each game and input will be terminated by a line containing only a single '#'.

For each game description in the input, output a single line of the form "Green can win in N move(s)", where $1 \le N \le 3$, and where N is the smallest such number, or "Green cannot win in 3 moves". Use the singular form when N = 1 and the plural form otherwise.

Sample Input

```
GGG# RR## R### R###
#### #### #### ####
#### #### #### ####
#### #### #### ####

RRR# RGG# RGG# ####
#### #### #### ####
#### #### #### ####
#### #### #### ####

#
```

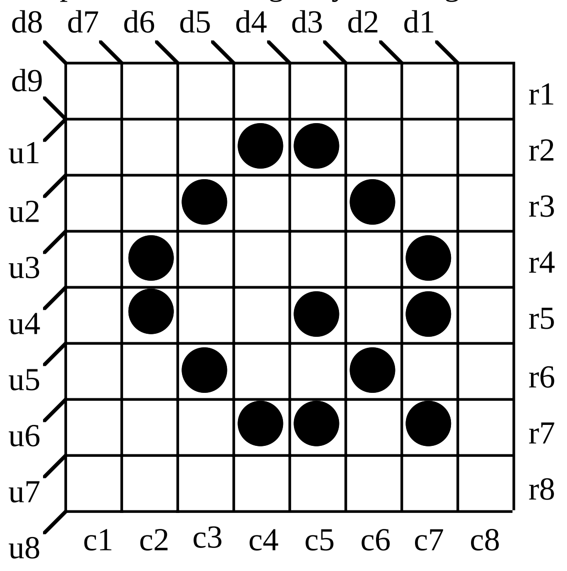
Sample Output

```
Green can win in 1 move
Green cannot win in 3 moves
```

Problem SP2000-H Discrete Digital Tomography

You are given a collection of sealed pizza boxes. The tops and bottoms of the boxes are covered internally with metal foil, but the edges are not. Inside, each box has been divided into *r* rows and *c* columns (both in the range 2 to 8 inclusive), after the manner of a chess board. Each of the thus-formed internal squares is either empty or contains a single widget. Widgets absorb beta radiation slightly, so by placing a beta emitter on one side of a box and a receiver on the other side you can tell how many widgets there are in the line of sight between emitter and receiver, but not where they are. By appropriate placement of the emitters and receivers we can determine the occupancies along various lines, i.e. how many widgets there are in each row (*r* numbers), column (*c* numbers), down diagonal (*r*+*c*-1 numbers), and up diagonal (*r*+*c*-1 numbers)

For example, the following 8 by 8 configuration will produce the numbers shown.



```
0 2 2 2 3 2 3 0               (r1..r8)
0 2 2 2 3 2 3 0               (c1..c8)
0 0 0 0 3 2 0 4 0 2 3 0 0 0 0 (d1..d15)
0 0 0 0 3 2 0 2 1 2 3 0 1 0 0 (u1..u15)
```

Write a program that will determine the arrangement of widgets in a pizza box, given a set of numbers such as those above. To make life a bit easier for you, the test data for this program will always have a unique solution. Also, the arrangement of widgets will be such that if any proper subset of the squares is revealed, there will always be at least one line with hidden squares such that either all the hidden squares in that line are empty or all the hidden squares in that line are occupied.

The input to the program is a sequence of problems, each consisting of five lines of integers. The first line of each problem contains r and c ($2 \le r \le 10$, $2 \le c \le 10$), line 2 contains r numbers giving the row occupancies, line 3 contains c numbers giving the column occupancies, line 4 contains r+c–1 numbers giving the down diagonal occupancies and line 5 contains r+c–1 numbers giving the up diagonal occupancies. A line of two zeroes (0 0) for r and c terminates the input.

The output of the program is a sequence of pictures, one per input problem except for the terminal one. The first line of the output contains the words “Pizza box” followed by a single space and the number of the problem (a running number starting at 1). The next r lines each contain c characters— either a ‘#’ for a full square or ‘–’ for an empty square. Leave a blank line between problems.

Sample Input

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

Sample Output

```
Pizza box 1
###
--#
--#
#-#

Pizza box 2
--------
---##---
--#--#--
-#----#-
-#--#-#-
--#--#--
---##-#-
--------
```