Problem A: Ananagrams

Most crossword puzzle fans are used to *anagrams*—groups of words with the same letters in different orders—for example OPTS, SPOT, STOP, POTS and POST. Some words however do not have this attribute, no matter how you rearrange their letters, you cannot form another word. Such words are called *ananagrams*, an example is QUIZ.

Obviously such definitions depend on the domain within which we are working; you might think that ATHENE is an ananagram, whereas any chemist would quickly produce ETHANE. One possible domain would be the entire English language, but this could lead to some problems. One could restrict the domain to, say, Music, in which case SCALE becomes a *relative ananagram* (LACES is not in the same domain) but NOTE is not since it can produce TONE.

Write a program that will read in the dictionary of a restricted domain and determine the relative ananagrams. Note that single letter words are, ipso facto, relative ananagrams since they cannot be "rearranged" at all. The dictionary will contain no more than 1000 words.

Input will consist of a series of lines. No line will be more than 80 characters long, but may contain any number of words. Words consist of up to 20 upper and/or lower case letters, and will not be broken across lines. Spaces may appear freely around words, and at least one space separates multiple words on the same line. Note that words that contain the same letters but of differing case are considered to be anagrams of each other, thus tIeD and EdiT are anagrams. The file will be terminated by a line consisting of a single `#`.

Output will consist of a series of lines. Each line will consist of a single word that is a relative ananagram in the input dictionary. Words must be output in lexicographic (case-sensitive) order. There will always be at least one relative ananagram.

Sample input

```
ladder came tape soon leader acme RIDE lone Dreis peat
 ScAlE orb  eye  Rides dealer  NotE derail LaCeS  drIed
noel dire Disk mace Rob dries
#
```

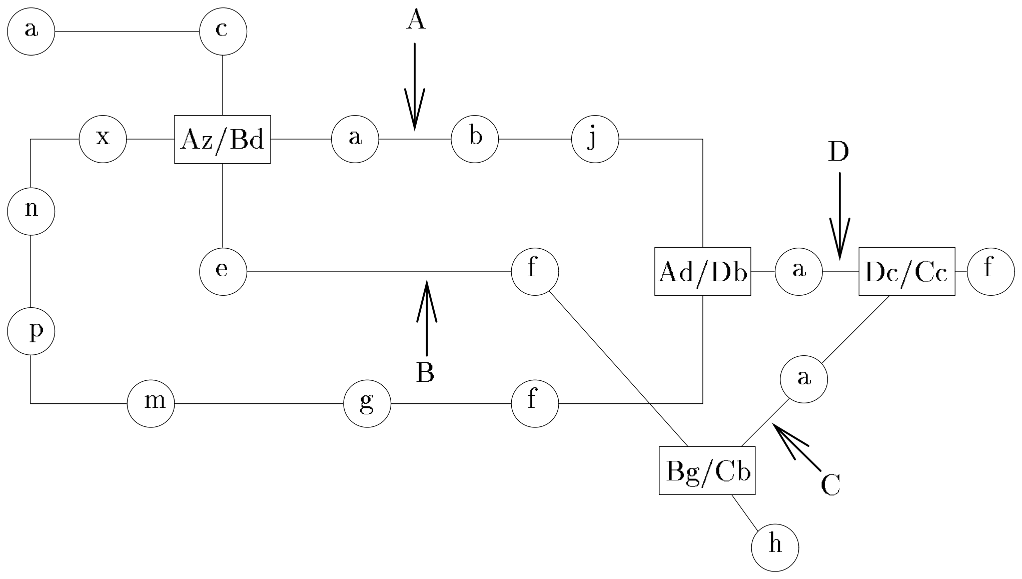
Sample output

```
Disk
NotE
derail
drIed
eye
ladder
soon
```

Problem B: Route Finding

Many cities provide a comprehensive public transport system, often integrating bus routes, suburban commuter train services and underground railways. Routes on such systems can be categorised according to the stations or stops along them. We conventionally think of them as forming lines (where the vehicle shuttles from one end of the route to the other and returns), loops (where the two ends of the "branch" are the same and vehicles circle the system in both directions) and connections, where each end of the route connects with another route. Obviously all of these can be thought of as very similar, and can connect with each other at various points along their routes. Note that vehicles can travel in both directions along all routes, and that it is only possible to change between routes at connecting stations.

To simplify matters, each route is given a designation letter from the set 'A' to 'Z', and each station along a route will be designated by another letter from the set 'a' to 'z'. Connecting stations will have more than one designation. Thus an example could be:



A common problem in such systems is finding a route between two stations. Once this has been done we wish to find the "best" route, where "best" means "shortest time".

Write a program that will read in details of such a system and then will find the fastest routes between given pairs of stations. You can assume that the trip between stations always takes 1 unit of time and that changing between routes at a connecting station takes 3 units of time.

Input will consist of two parts. The first will consist of a description of a system, the second will consist of pairs of stations. The description will start with a number between 1 and 26 indicating how many routes there are in the system. This will be followed by that many lines, each describing a single route. Each line will start with the route identifier followed by a ':' followed by the stations along that route, in order. Connections will be indicated by an '=' sign followed by the complete alternative designation. All connections will be identified at least once, and if there are more than two lines meeting at a connection, some or of all the alternative designations may be identified together,. That is, there may be sequences such as '...hc=Bg=Cc=Abd...'. If the route forms a loop then the last station will be the same as the first. This is the only situation in which station letters will be repeated. The next portion of the input file will consist of a sequence of lines each containing two stations written contiguously. The file will be terminated by a line consisting of a single **#**.

Output will consist of a series of lines, one for each pair of stations in the input. Each line will consist of the time for the fastest route joining the two stations, right justified in a field of width 3, followed by a colon and a space and the sequence of stations representing the shortest journey. Follow the example shown below. Note that there will always be only one fastest route for any given pair of stations and that the route must start and finish at the named stations (not at any synonyms thereof), hence the time for the route must include the time for any inter-station transfers.

The example input below refers to the diagram given above.

Sample input

```
4
A:fgmpnxzabjd=Dbf
D:b=Adac=Ccf
B:acd=Azefg=Cbh
C:bac
AgAa
AbBh
BhDf
#
```

Sample output

```
  5: Agfdjba
  9: Abaz=Bdefgh
 10: Bhg=Cbac=Dcf
```

Problem C: Calendar

Most of us have a calendar on which we scribble details of important events in our lives—visits to the dentist, the Regent 24 hour book sale, Programming Contests and so on. However there are also the fixed dates: partner's birthdays, wedding anniversaries and the like; and we also need to keep track of these. Typically we need to be reminded of when these important dates are approaching—the more important the event, the further in advance we wish to have our memories jogged.

Write a program that will provide such a service. The input will specify the year for which the calendar is relevant (in the range 1901 to 1999). Bear in mind that, within the range specified, all years that are divisible by 4 are leap years and hence have an extra day (February 29th) added. The output will specify "today's" date, a list of forthcoming events and an indication of their relative importance.

The first line of input will contain an integer representing the year (in the range 1901 to 1999). This will be followed by a series of lines representing anniversaries or days for which the service is requested. An anniversary line will consist of the letter '`A`'; three integer numbers (D, M, P) representing the date, the month and the importance of the event; and a string describing the event, all separated by one or more spaces. P will be a number between 1 and 7 (both inclusive) and represents the number of days before the event that the reminder service should start. The string describing the event will always be present and will start at the first non-blank character after the priority. A date line will consist of the letter '`D`' and the date and month as above. All anniversary lines will precede any date lines. No line will be longer than 255 characters in total. The file will be terminated by a line consisting of a single `#`.

Output will consist of a series of blocks of lines, one for each date line in the input. Each block will consist of the requested date followed by the list of events for that day and as many following days as necessary. The output should specify the date of the event (D and M), right justified in fields of width 3, and the relative importance of the event. Events that happen today should be flagged as shown below, events that happen tomorrow should have P stars, events that happen the day after tomorrow should have P-1 stars, and so on. If several events are scheduled for the same day, order them by relative importance (number of stars). If there is still a conflict, order them by their appearance in the input stream. Follow the format used in the example below. Leave 1 blank line between blocks.

Sample input

```
1993
A 23 12 5 Partner's birthday
A 25 12 7    Christmas
A 20 12 1 Unspecified Anniversary
D 20 12
#
```

Sample output

```
Today is: 20 12
 20 12 *TODAY* Unspecified Anniversary
 23 12 ***     Partner's birthday
 25 12 ***     Christmas
```

Problem D: Word Crosses

A *word cross* is formed by printing a pair of words, the first horizontally and the second vertically, so that they share a common letter. A *leading word cross* is one where the common letter is as near as possible to the beginning of the horizontal word, and, for this letter, as close as possible to the beginning of the vertical word. Thus DEFER and PREFECT would cross on the first 'E' in each word, PREFECT and DEFER would cross on the 'R'. *Double leading word crosses* use two pairs of words arranged so that the two horizontal words are on the same line and each pair forms a leading word cross.

Write a program that will read in sets of four words and form them (if possible) into double leading word crosses.

Input will consist of a series of lines, each line containing four words (two pairs). A word consists of 1 to 10 upper case letters, and will be separated from its neighbours by at least one space. The file will be terminated by a line consisting of a single `#`.

Output will consist of a series of double leading word crosses as defined above. Leave exactly three spaces between the horizontal words. If it is not possible to form both crosses, write the message '`Unable to make two crosses`'. Leave 1 blank line between output sets.

Sample input

```
MATCHES CHEESECAKE PICNIC EXCUSES
PEANUT BANANA VACUUM  GREEDY
A  VANISHING   LETTER TRICK
#
```

Sample output

```
 C
 H
 E
 E
 S
 E          E
 C          X
MATCHES   PICNIC
 K          U
 E          S
            E
            S

Unable to make two crosses

V
A   LETTER
N     R
I     I
S     C
H     K
I
N
G
```

Problem E: Factors and Factorials

The factorial of a number N (written $N!$) is defined as the product of all the integers from 1 to N. It is often defined recursively as follows:

$$1! = 1$$

$$N! = N * (N - 1)!$$

Factorials grow very rapidly—5! = 120, 10! = 3,628,800. One way of specifying such large numbers is by specifying the number of times each prime number occurs in it, thus 825 could be specified as (0 1 2 0 1) meaning no twos, 1 three, 2 fives, no sevens and 1 eleven.

Write a program that will read in a number N ($2 \leq N \leq 100$) and write out its factorial in terms of the numbers of the primes it contains.

Input will consist of a series of lines, each line containing a single integer N. The file will be terminated by a line consisting of a single 0.

Output will consist of a series of blocks of lines, one block for each line of the input. Each block will start with the number N, right justified in a field of width 3, and the chracters '`!`', space, and '`=`'. This will be followed by a list of the number of times each prime number occurs in N!. These should be right justified in fields of width 3 and each line (except the last of a block, which may be shorter) should contain fifteen numbers. Any lines after the first should be indented. Follow the layout of the example shown below exactly.

Sample input

```
5
53
0
```

Sample output

```
  5! =  3  1  1
 53! = 49 23 12  8  4  4  3  2  2  1  1  1  1  1  1
        1
```

Problem F: Traffic Lights

One way of achieving a smooth and economical drive to work is to 'catch' every traffic light, that is have every signal change to green as you approach it. One day you notice as you come over the brow of a hill that every traffic light you can see has just changed to green and that therefore your chances of catching every signal is slight. As you wait at a red light you begin to wonder how long it will be before all the lights again show green, not necessarily all turn green, merely all show green simultaneously, even if it is only for a second.

Write a program that will determine whether this event occurs within a reasonable time. Time is measured from the instant when they all turned green simultaneously, although the initial portion while they are all still green is excluded from the reckoning.

Input will consist of a series of scenarios. Data for each scenario will consist of a series of integers representing the cycle times of the traffic lights, possibly spread over many lines, with no line being longer than 100 characters. Each number represents the cycle time of a single signal. The cycle time is the time that traffic may move in one direction; note that the last 5 seconds of a green cycle is actually orange. Thus the number 25 means a signal that (for a particular direction) will spend 20 seconds green, 5 seconds orange and 25 seconds red. Cycle times will not be less than 10 seconds, nor more than 90 seconds. There will always be at least two signals in a scenario and never more than 100. Each scenario will be terminated by a zero (0). The file will be terminated by a line consisting of three zeroes (0 0 0).

Output will consist of a series of lines, one for each scenario in the input. Each line will consist of the time in hours, minutes and seconds that it takes for all the signals to show green again after at least one of them changes to orange. Follow the format shown in the examples. Time is measured from the instant they all turn green simultaneously. If it takes more than five hours before they all show green simultaneously, the message "`Signals fail to synchronise in 5 hours`" should be written instead.

Sample input

```
19 20    0
30
  25     35  0
0 0 0
```

Sample output

```
00:00:40
00:05:00
```

Problem G: Beggar My Neighbour

"Beggar My Neighbour" (sometimes known as "Strip Jack Naked") is a traditional card game, designed to help teach beginners something about cards and their values. A standard deck is shuffled and dealt face down to the two players, the first card to the non-dealer, the second to the dealer, and so on until each player has 26 cards. The dealer receives the last card. The non-dealer starts the game by playing the top card of her deck (the second last card dealt) face up on the table. The dealer then covers it by playing her top card face up. Play continues in this fashion until a "face" card (Ace, King, Queen or Jack) is played. The next player must then "cover" that card, by playing one card for a Jack, two for a Queen, three for a King and four for an Ace. If a face card is played at any stage during this sequence, play switches and the other player must cover that card. When this sequence has ended, the player who exposed the last face card takes the entire heap, placing it face down under her existing deck. She then starts the next round by playing one card face up as before, and play continues until one player cannot play when called upon to do so, because they have no more cards.

Write a program that will simulate playing this game. Remember that a standard deck (or pack) of cards contains 52 cards. These are divided into 4 suits—Spades (♠), Hearts (♡), Diamonds (♢) and Clubs (♣). Within each suit there are 13 cards—Ace (A), 2–9, Ten (T), Jack (J), Queen (Q) and King (K).

Input will consist of a series of decks of cards. Each deck will give the cards in order as they would be dealt (that is in the example deck below, the non-dealer would start the game by playing the H2). Decks will occupy 4 lines with 13 cards on each. The designation of each card will be the suit (S, H, D, C) followed by the rank (A, 2–9, T, J, Q, K). There will be exactly one space between cards. The file will be terminated by a line consisting of a single `#`.

Output will consist of a series of lines, one for each deck in the input. Each line will consist of the number of the winning player (1 is the dealer, 2 is the first to play) and the number of cards in the winner's hand (ignoring any on the stack), right justified in a field of width 3.

Sample input

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

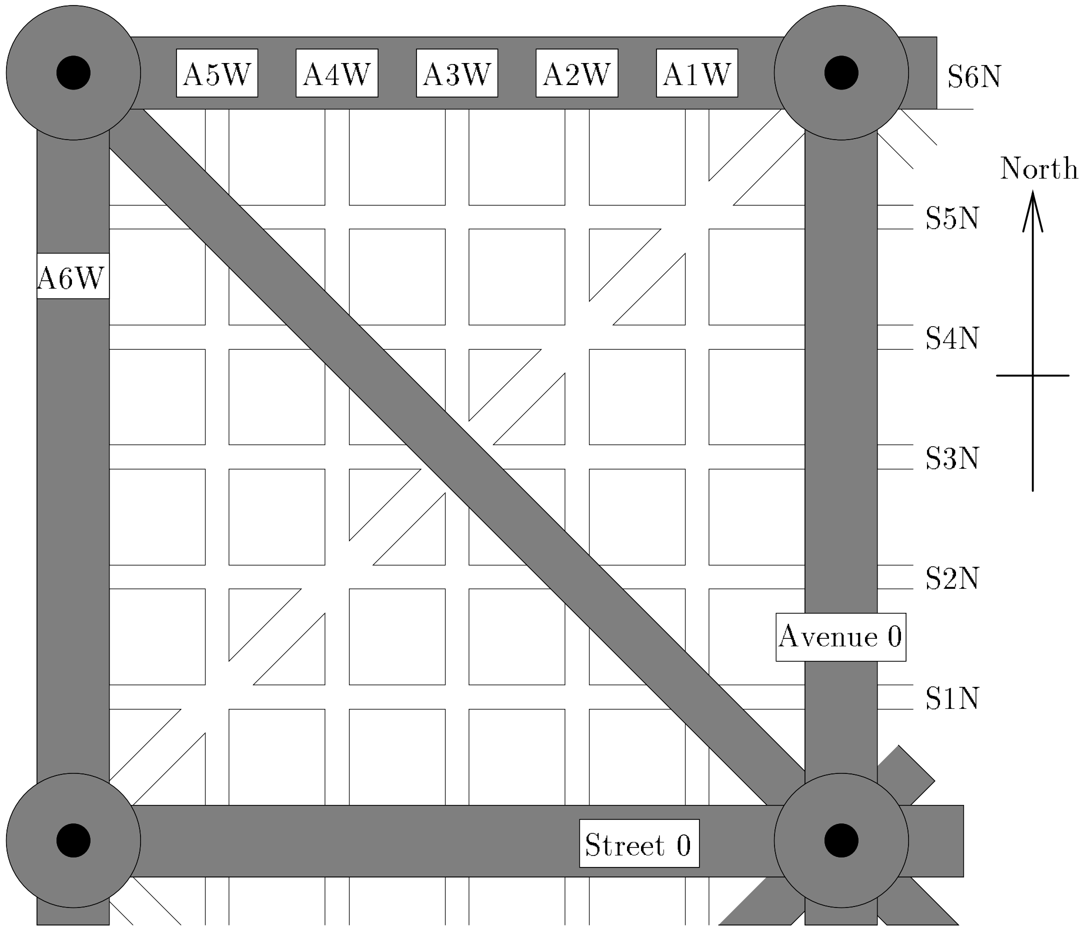
Sample output

```
1 44
```

Problem H: City Directions

When driving through a city, an intersection usually offers one the choice of going straight on or turning left or right through 90 degrees. However some cities have diagonal roads, thus at intersections involving these one may be able to turn through 45 degrees ("half") or through 135 degrees ("sharp").

Consider such a city with Avenues running north-south, Streets running east-west and Boulevards running diagonally. The central Avenue and Street are labelled Zero (A0 and S0). Other roads are labelled relative to these, thus A3W is the third avenue to the west of A0. There are 6 Boulevards—two passing through the centre of the city, and 4 others, one in each quadrant. The diagram below shows the northwest quadrant of a small version of such a city.



The roads marked in grey are considered to be throughways. These are elevated for most of their length, thus it is possible to cross them easily, however they always intersect each other at a circle, which is shared by all other roads that meet at that intersection. You may only enter or leave them by turning left (sharp left in the case of boulevards). You may not stop on them for any reason. There are no restrictions on turns for other roads.

This system allows a very simple method of determining one's current position and a way of arriving at one's destination. Position can be specified in terms of the last intersection you passed through (the numbers of the Avenue and Street that meet there) and your current heading, which can be one of: north (N), northeast (NE), east (E), southeast (SE), south (S), southwest (SW), west (W) and northwest (NW). Directions can then be given in terms of how many intersections to pass through and which turns to make. However, the locals have an infuriating habit of giving incorrect or invalid directions, although it cannot be determined whether this is deliberate or accidental. Directions should (but don't always) conform to the following simple grammar:

```
<command> ::= <turn_command> | <straight_command>
<turn_command> ::= TURN [HALF | SHARP] {LEFT | RIGHT}
<straight_command> ::= GO [STRAIGHT] n                    1 ≤ n ≤ 99
```

Write a program that will simulate driving through such a city, by tracking your position and heading as you follow a set of directions (commands). Each quadrant of the city will be 50 blocks by 50 blocks, thus the entire city will be 100 blocks by 100 blocks, the outer throughways will be labelled Fifty and the major and minor boulevards will cross at roads labelled Twentyfive. You will be told your starting position and heading and then given a series of directions. If a direction does not follow the above grammar, or would involve an illegal or impossible turn then ignore it. At no stage will directions take you out of the confines of the city.

Input will consist of a series of scenarios. Each scenario will consist of a position and a heading and will be followed by a series of directions (commands), each on a separate line. If either of the roads involved is one of the central roads (A0, S0), they will be labelled N or E as appropriate. Note that you may assume that you have just left the intersection specified. The GO ¡n¿ command means that you pass through ¡n¿ intersections. Each scenario will be terminated by a line consisting of the word STOP. The file will be terminated by a line consisting of the word END only. Input data will follow the format shown below, except that more than one space may occur where only one is shown. No line will be longer than 80 characters.

Output will consist of a series of lines, one for each scenario. Each line will consist of a position and a heading in the same format as the input. If the final stopping place is illegal, report '`Illegal stopping place`' as the answer.

Sample input

```
A2W S1N E
TURN SHARP LEFT
GO 1
TURN RIGHT
TURN LEFT
TURN SHARP LEFT
GO 1
TURN LEFT
STOP
A2W S1N W
GO STRAIGHT 2
TURN LEFT
GO ON 2
TURN HALF LEFT
TURN LEFT
GO 2
STOP
END
```

Sample output

```
A3W S1N E
Illegal stopping place
```