Final term <u>BONUS Projects</u> for

C-Language Programming

Level 1 Electrical Engineering

Notes:

- Group Work is appreciated so it is accepted to have a group working on the same project.
- The number of students in the group may not exceed 10 students at any case.
- Each group <u>MUST</u> select 30 different problems from the first group (Group A) and 10 different problems from the additional problems (Group B) totalling 40 different problems from the ones outlined afterwards in both Groups A and B. These problems are renowned worldwide and are famous among all the people studying the C-language in the whole world.
- The average number of problems to be solved by each student is then *FOUR* problems only.
- Each one of the students in the same group must be able to defend the whole cause by himself (i.e., must understand everything about the 40 problems), though each one of them is asked thoroughly and separately.
- Projects will be marked out of 100.
- Please submit the following by the due date as explained later:

i)	Computer Written report	20% of mark
ii)	The explanation of the algorithm used	10% of mark
iii)	Program Flow Chart	20% of mark
iv)	Computer working source Code	20% of mark
v)	Sample input test cases and corresponding outputs	10% of mark
vi)	Power Point presentation detailing all the above	20% of mark

- **A detailed Flow Chart is very advantageous for the program you are solving.**
- **ALL students from first year MUST hand-in their projects. You must hand it in person in order to be evaluated on the spot and given a mark.**
- **If you leave the report to be handed in by a third party, it will be graded from half the original mark.**
- **If you are travelling or will not be available at the set dates, make sure to hand in the project earlier. Please be vigilant enough to coordinate with one of the supervisors APRIORI.**
- **If two or more reports from different groups are found to be identical or copied, each of them will be graded from half the mark (or the ratio of their number). In other words, the maximum mark will be divided between the students having given their report to be copied or having presented a copied report.**
- **The due date is Saturday the 27th of May 2023.**

Group A
Proposed Problems

Problem No. 1: The Blocks Problem

Background

Many areas of Computer Science use simple, abstract domains for both analytical and empirical studies. For example, an early AI study of planning and robotics (STRIPS) used a block world in which a robot arm performed tasks involving the manipulation of blocks.

In this problem you will model a simple block world under certain rules and constraints. Rather than determine how to achieve a specified state, you will ``program" a robotic arm to respond to a limited set of commands.

The Problem

The problem is to parse a series of commands that instruct a robot arm in how to manipulate blocks that lie on a flat table. Initially there are n blocks on the table (numbered from 0 to n-1) with block b_i adjacent to block b_{i+1} for all $0 \le i < n-1$ as shown in the diagram below:

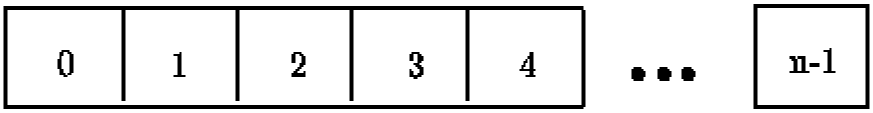


Figure: Initial Blocks World

The valid commands for the robot arm that manipulates blocks are:

move a onto b

where a and b are block numbers, puts block a onto block b after returning any blocks that are stacked on top of blocks a and b to their initial positions.

move a over b

where a and b are block numbers, puts block a onto the top of the stack containing block b, after returning any blocks that are stacked on top of block a to their initial positions.

pile a onto b

where a and b are block numbers, moves the pile of blocks consisting of block a, and any blocks that are stacked above block a, onto block b. All blocks on top of block b are moved to their initial positions prior to the pile taking place. The blocks stacked above block a retain their order when moved.

pile a over b

where a and b are block numbers, puts the pile of blocks consisting of block a, and any blocks that are stacked above block a, onto the top of the stack containing block b. The blocks stacked above block a retain their original order when moved.

quit

terminates manipulations in the block world.

Any command in which $a = b$ or in which a and b are in the same stack of blocks is an illegal command. All illegal commands should be ignored and should have no affect on the configuration of blocks.

The Input

The input begins with an integer *n* on a line by itself representing the number of blocks in the block world. You may assume that $0 < n < 25$.

The number of blocks is followed by a sequence of block commands, one command per line. Your program should process all commands until the quit command is encountered.

You may assume that all commands will be of the form specified above. There will be no syntactically incorrect commands.

The Output

The output should consist of the final state of the blocks world. Each original block position numbered *i* ($0\leq i<n$ where *n* is the number of blocks) should appear followed immediately by a colon. If there is at least a block on it, the colon must be followed by one space, followed by a list of blocks that appear stacked in that position with each block number separated from other block numbers by a space. Don't put any trailing spaces on a line.

There should be one line of output for each block position (i.e., *n* lines of output where *n* is the integer on the first line of input).

Sample Input

```
10
move 9 onto 1
move 8 over 1
move 7 over 1
move 6 over 1
pile 8 over 6
pile 8 over 5
move 2 over 1
move 4 over 9
quit
```

Sample Output

```
 0: 0
 1: 1 9 2 4
 2:
 3: 3
 4:
 5: 5 8 7 6
 6:
 7:
 8:
 9:
```

Problem No. 2: Arbitrage

Background

The use of computers in the finance industry has been marked with controversy lately as programmed trading -- designed to take advantage of extremely small fluctuations in prices -- has been outlawed at many Wall Street firms. The ethics of computer programming is a fledgling field with many thorny issues.

The Problem

Arbitrage is the trading of one currency for another with the hopes of taking advantage of small differences in conversion rates among several currencies in order to achieve a profit. For example, if $1.00 in U.S. currency buys 0.7 British pounds currency, £1 in British currency buys 9.5 French francs, and 1 French franc buys 0.16 in U.S. dollars, then an arbitrage trader can start with $1.00 and earn 1*0.7*9.5*0.16=1.064 dollars thus earning a profit of 6.4 percent.

You will write a program that determines whether a sequence of currency exchanges can yield a profit as described above.

To result in successful arbitrage, a sequence of exchanges must begin and end with the same currency, but any starting currency may be considered.

The Input

The input file consists of one or more conversion tables. You must solve the arbitrage problem for each of the tables in the input file.

Each table is preceded by an integer *n* on a line by itself giving the dimensions of the table. The maximum dimension is 20; the minimum dimension is 2.

The table then follows in row major order but with the diagonal elements of the table missing (these are assumed to have value 1.0). Thus the first row of the table represents the conversion rates between country 1 and *n*-1 other countries, i.e., the amount of currency of country *i* ($2 \le i \le n$) that can be purchased with one unit of the currency of country 1.

Thus each table consists of *n*+1 lines in the input file: 1 line containing *n* and *n* lines representing the conversion table.

The Output

For each table in the input file you must determine whether a sequence of exchanges exists that results in a profit of more than 1 percent (0.01). If a sequence exists you must print the sequence of exchanges that results in a profit. If there is more than one sequence that results in a profit of more than 1 percent you must print a sequence of minimal length, i.e., one of the sequences that uses the fewest exchanges of currencies to yield a profit.

Because the IRS (United States Internal Revenue Service) notices lengthy transaction sequences, all profiting sequences must consist of *n* or fewer transactions where *n* is the dimension of the table giving conversion rates. The sequence 1 2 1 represents two conversions.

If a profiting sequence exists you must print the sequence of exchanges that results in a profit. The sequence is printed as a sequence of integers with the integer *i* representing the i^{th} line of the conversion table (country *i*). The first integer in the sequence is the country from which the profiting sequence starts. This integer also ends the sequence.

If no profiting sequence of *n* or fewer transactions exists, then the line

no arbitrage sequence exists

should be printed.

Sample Input

3

1.2 .89

```
.88 5.1
1.1 0.15
4
3.1    0.0023    0.35
0.21   0.00353   8.13
200    180.559   10.339
2.11   0.089     0.06111
2
2.0
0.45
```

Sample Output

```
1 2 1
1 2 4 1
no arbitrage sequence exists
```

Problem No. 3: The Cat in the Hat

Background

The Cat in the Hat is a nasty creature, But the striped hat he is wearing has a rather nifty feature. With one flick of his wrist he pops his top off. Do you know what's inside that Cat's hat? A bunch of small cats, each with its own striped hat. Each little cat does the same as line three, all except the littlest ones, who just say ``Why me?" Because the littlest cats have to clean all the grime, and they're tired of doing it time after time!

The Problem

A clever cat walks into a messy room which he needs to clean. Instead of doing the work alone, it decides to have its helper cats do the work. It keeps its (smaller) helper cats inside its hat. Each helper cat also has helper cats in its own hat, and so on. Eventually, the cats reach a smallest size. These smallest cats have no additional cats in their hats. These unfortunate smallest cats have to do the cleaning.

The number of cats inside each (non-smallest) cat's hat is a constant, *N*. The height of these cats-in-a-hat is 1/(N+1) times the height of the cat whose hat they are in.

The smallest cats are of height one; these are the cats that get the work done.

All heights are positive integers.

Given the height of the initial cat and the number of worker cats (of height one), find the number of cats that are not doing any work (cats of height greater than one) and also determine the sum of all the cats' heights (the height of a stack of all cats standing one on top of another).

The Input

The input consists of a sequence of cat-in-hat specifications. Each specification is a single line consisting of two positive integers, separated by white space. The first integer is the height of the initial cat, and the second integer is the number of worker cats.

A pair of 0's on a line indicates the end of input.

The Output

For each input line (cat-in-hat specification), print the number of cats that are not working, followed by a space, followed by the height of the stack of cats. There should be one output line for each input line other than the ``0 0'' that terminates input.

Sample Input

```
216 125
5764801 1679616
0 0
```

Sample Output

```
31 671
335923 30275911
```

Problem No. 4: Searching Quickly

Background

Searching and sorting are part of the theory and practice of computer science. For example, binary search provides a good example of an easy-to-understand algorithm with sub-linear complexity. Quicksort is an efficient O(n log n) [average case] comparison based sort.

KWIC-indexing is an indexing method that permits efficient ``human search'' of, for example, a list of titles.

The Problem

Given a list of titles and a list of ``words to ignore'', you are to write a program that generates a KWIC (Key Word In Context) index of the titles. In a KWIC-index, a title is listed once for each keyword that occurs in the title. The KWIC-index is alphabetized by keyword.

Any word that is not one of the ``words to ignore'' is a potential keyword.

For example, if words to ignore are ``the, of, and, as, a'' and the list of titles is:

Descent of Man
The Ascent of Man
The Old Man and The Sea
A Portrait of The Artist As a Young Man

A KWIC-index of these titles might be given by:

```
               a portrait of the ARTIST as a young man
                         the ASCENT of man
                            DESCENT of man
                    descent of MAN
                  the ascent of MAN
                      the old MAN and the sea
   a portrait of the artist as a young MAN
                         the OLD man and the sea
                          a PORTRAIT of the artist as a young man
              the old man and the SEA
       a portrait of the artist as a YOUNG man
```

The Input

The input is a sequence of lines, the string :: is used to separate the list of words to ignore from the list of titles. Each of the words to ignore appears in lower-case letters on a line by itself and is no more than 10 characters in length. Each title appears on a line by itself and may consist of mixed-case (upper and lower) letters. Words in a title are separated by whitespace. No title contains more than 15 words.

There will be no more than 50 words to ignore, no more than 200 titles, and no more than 10,000 characters in the titles and words to ignore combined. No characters other than 'a'-'z', 'A'-'Z', and white space will appear in the input.

The Output

The output should be a KWIC-index of the titles, with each title appearing once for each keyword in the title, and with the KWIC-index alphabetized by keyword. If a word appears more than once in a title, each instance is a potential keyword.

The keyword should appear in all upper-case letters. All other words in a title should be in lower-case letters. Titles in the KWIC-index with the same keyword should appear in the same order as they appeared in the input file. In the case where multiple instances of a word are keywords in the same title, the keywords should be capitalized in left-to-right order.

Case (upper or lower) is irrelevant when determining if a word is to be ignored.

The titles in the KWIC-index need NOT be justified or aligned by keyword, all titles may be listed left-justified.

Problem No. 5: Numbering Paths

Background

Problems that process input and generate a simple ``yes'' or ``no'' answer are called decision problems. One class of decision problems, the NP-complete problems, are not amenable to general efficient solutions. Other problems may be simple as decision problems, but enumerating all possible ``yes'' answers may be very difficult (or at least time-consuming).

This problem involves determining the number of routes available to an emergency vehicle operating in a city of one-way streets.

The Problem

Given the intersections connected by one-way streets in a city, you are to write a program that determines the number of different routes between each intersection. A route is a sequence of one-way streets connecting two intersections.

Intersections are identified by non-negative integers. A one-way street is specified by a pair of intersections. For example, *j k* indicates that there is a one-way street from intersection *j* to intersection *k*. Note that two-way streets can be modelled by specifying two one-way streets: *j k* and *k j*.

Consider a city of four intersections connected by the following one-way streets:

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

There is one route from intersection 0 to 1, two routes from 0 to 2 (the routes are $0 \rightarrow 1 \rightarrow 2$and $0 \rightarrow 2$), two routes from 0 to 3, one route from 1 to 2, one route from 1 to 3, one route from 2 to 3, and no other routes.

It is possible for an infinite number of different routes to exist. For example if the intersections above are augmented by the street $3 \quad 2$, there is still only one route from 0 to 1, but there are infinitely many different routes from 0 to 2. This is because the street from 2 to 3 and back to 2 can be repeated yielding a different sequence of streets and hence a different route. Thus the route $0 \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 3 \rightarrow 2$is a different route than $0 \rightarrow 2 \rightarrow 3 \rightarrow 2$.

The Input

The input is a sequence of city specifications. Each specification begins with the number of one-way streets in the city followed by that many one-way streets given as pairs of intersections. Each pair *j k* represents a one-way street from intersection *j* to intersection *k*. In all cities, intersections are numbered sequentially from 0 to the ``largest'' intersection. All integers in the input are separated by whitespace. The input is terminated by end-of-file.

There will never be a one-way street from an intersection to itself. No city will have more than 30 intersections.

The Output

For each city specification, a square matrix of the number of different routes from intersection *j* to intersection *k* is printed. If the matrix is denoted *M*, then *M*[*j*][*k*] is the number of different routes from intersection *j* to intersection *k*. The matrix *M* should be printed in row-major order, one row per line. Each matrix should be preceded by the string ``matrix for city *k*'' (with *k* appropriately instantiated, beginning with 0).

If there are an infinite number of different paths between two intersections a -1 should be printed. DO NOT worry about justifying and aligning the output of each matrix. All entries in a row should be separated by whitespace.

Sample Input

```
7 0 1 0 2 0 4 2 4 2 3 3 1 4 3
5
0 2
0 1 1 5 2 5 2 1
9
0 1 0 2 0 3
0 4 1 4 2 1
2 0
3 0
3 1
```

Sample Output

```
matrix for city 0
0 4 1 3 2
0 0 0 0 0
0 2 0 2 1
0 1 0 0 0
0 1 0 1 0
matrix for city 1
```

```
0 2 1 0 0 3
0 0 0 0 0 1
0 1 0 0 0 2
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
matrix for city 2
-1 -1 -1 -1 -1
0 0 0 0 1
-1 -1 -1 -1 -1
-1 -1 -1 -1 -1
0 0 0 0 0
```

Problem No. 6: Ugly Numbers

Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence
1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ...
shows the first 11 ugly numbers. By convention, 1 is included.
Write a program to find and print the 1500'th ugly number.

Input and Output

There is no input to this program. Output should consist of a single line as shown below, with <number> replaced by the number computed.

Sample output

```
The 1500'th ugly number is <number>.
```

Problem No. 7: Student Grant

The Government of Impecunia has decided to discourage tertiary students by making the payments of tertiary grants a long and time-consuming process. Each student is issued a student ID card which has a magnetically encoded strip on the back which records the payment of the student grant. This is initially set to zero. The grant has been set at $40 per year and is paid to the student on the working day nearest to his birthday. Thus on any given working day up to 25 students will appear at the nearest office of the Department of Student Subsidies to collect their grant.

The grant is paid by an Automatic Teller Machine which is driven by a reprogrammed 8085 chip originally designed to run the state slot machine. The ATM was built in the State Workshops and is designed to be difficult to rob. It consists of an interior vault where it holds a large stock of $1 coins and an output store from which these coins are dispensed. To limit possible losses it will only move coins from the vault to the output store when that is empty. When the machine is switched on in the morning, with an empty output store, it immediately moves 1 coin into the output store. When that has been dispensed it will then move 2 coins, then 3, and so on until it reaches some preset limit k. It then recycles back to 1, then 2 and so on.

The students form a queue at this machine and, in turn, each student inserts his card. The machine dispenses what it has in its output store and updates the amount paid to that student by writing the new total on the card. If the student has not received his full grant, he removes his card and rejoins the queue at the end. If the amount in the store plus what the student has already received comes to more than $40, the machine only pays out enough to make the total up to $40. Since this fact is recorded on the card, it is pointless for the student to continue queuing and he leaves. The amount remaining in the store is then available for the next student.

Write a program that will read in values of N (the number of students, 1 ≤N≤ 25) and k (the limit for that machine, 1 ≤k≤ 40) and calculate the order in which the students leave the queue.

Input and Output

Input will consist of a series of lines each containing a value for N and k as integers. The list will be terminated by two zeroes (0 0).

Output will consist of a line for each line of input and will contain the list of students in the order in which they leave the queue. Students are ordered according to their position in the queue at the start of the day. All numbers must be right justified in a field of width 3.

Sample input

```
5 3
0 0
```

Sample output

```
  1  3  5  2  4
```

Problem No. 8: Dollars

New Zealand currency consists of $100, $50, $20, $10, and $5 notes and $2, $1, 50c, 20c, 10c and 5c coins. Write a program that will determine, for any given amount, in how many ways that amount may be made up. Changing the order of listing does not increase the count. Thus 20c may be made up in 4 ways: 1*20c, 2*10c, 10c+2*5c, and 4*5c.

Input

Input will consist of a series of real numbers no greater than $300.00 each on a separate line. Each amount will be valid, that is will be a multiple of 5c. The file will be terminated by a line containing zero (0.00).

Output

Output will consist of a line for each of the amounts in the input, each line consisting of the amount of money (with two decimal places and right justified in a field of width 6), followed by the number of ways in which that amount may be made up, right justified in a field of width 17.

Sample input

```
0.20
2.00
0.00
```

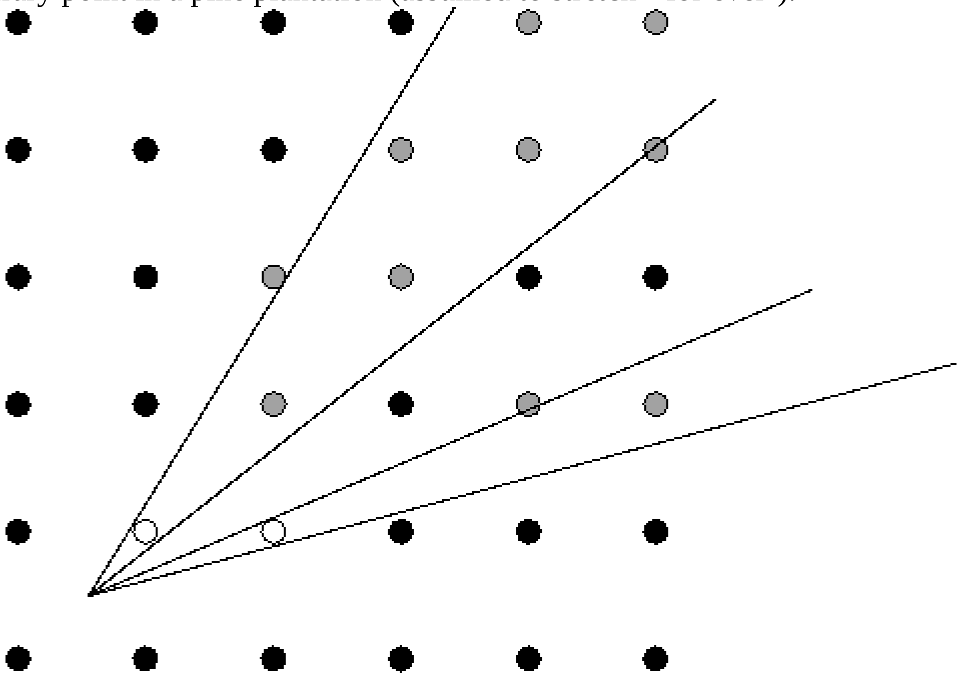
Sample output

```
  0.20                4
```

```
2.00       293
```

Problem No. 9: Forests

The saying ``You can't see the wood for the trees" is not only a cliche, but is also incorrect. The real problem is that you can't see the trees for the wood. If you stand in the middle of a ``wood" (in NZ terms, a patch of bush), the trees tend to obscure each other and the number of distinct trees you can actually see is quite small. This is especially true if the trees are planted in rows and columns (as in a pine plantation), because they tend to line up. The purpose of this problem is to find how many distinct trees you can see from an arbitrary point in a pine plantation (assumed to stretch ``for ever").



You can only see a distinct tree if no part of its trunk is obscured by a nearer tree--that is if both sides of the trunk can be seen, with a discernible gap between them and the trunks of all trees closer to you. Also, you can't see a tree if it is apparently ``too small". For definiteness, ``not too small" and ``discernible gap" will mean that the angle subtended at your eye is greater than 0.01 degrees (you are assumed to use one eye for observing). Thus the two trees marked ○obscure at least the trees marked ●from the given view point.

Write a program that will determine the number of trees visible under these assumptions, given the diameter of the trees, and the coordinates of a viewing position. Because the grid is infinite, the origin is unimportant, and the coordinates will be numbers between 0 and 1.

Input

Input will consist of a series of lines, each line containing three real numbers of the form 0.nn. The first number will be the trunk diameter--all trees will be assumed to be cylinders of exactly this diameter, with their centres placed exactly on the points of a rectangular grid with a spacing of one unit. The next two numbers will be the x and y coordinates of the observer. To avoid potential problems, say by being too close to a tree, we will guarantee that *diameter≤x,y≤(1-diameter)*. To avoid problems with trees being too small you may assume that *diameter≥0.1*. The file will be terminated by a line consisting of three zeroes.

Output

Output will consist of a series of lines, one for each line of the input. Each line will consist of the number of trees of the given size, visible from the given position.

Sample input

```
0.10 0.46 0.38
0 0 0
```

Sample output

```
128
```

Problem No. 10: Recycling

Kerbside recycling has come to New Zealand, and every city from Auckland to Invercargill has leapt on to the band wagon. The bins come in 5 different colours--red, orange, yellow, green and blue--and 5 wastes have been identified for recycling--Plastic, Glass, Aluminium, Steel, and Newspaper. Obviously there has been no coordination between cities, so each city has allocated wastes to bins in an arbitrary fashion. Now that the government has solved the minor problems of today (such as reorganising Health, Welfare and Education), they are looking around for further challenges. The Minister for Environmental Doodads wishes to introduce the ``Regularisation of Allocation of Solid Waste to Bin Colour Bill" to Parliament, but in order to do so needs to determine an allocation of his own. Being a firm believer in democracy (well some of the time anyway), he surveys all the cities that are using this recycling method. From these data he wishes to determine the city whose allocation scheme (if imposed on the rest of the country) would cause the least impact, that is would cause the smallest number of changes in the allocations of the other cities. Note that the sizes of the cities is not an issue, after all this is a democracy with the slogan ``One City, One Vote".

Write a program that will read in a series of allocations of wastes to bins and determine which city's allocation scheme should be chosen. Note that there will always be a clear winner.

Input and Output

Input will consist of a series of blocks. Each block will consist of a series of lines and each line will contain a series of allocations in the form shown in the example. There may be up to 100 cities in a block. Each block will be terminated by a line starting with `e'. The entire file will be terminated by a line consisting of a single #.

Output will consist of a series of lines, one for each block in the input. Each line will consist of the number of the city that should be adopted as a national example.

Sample input

```
r/P,o/G,y/S,g/A,b/N
r/G,o/P,y/S,g/A,b/N
r/P,y/S,o/G,g/N,b/A
r/P,o/S,y/A,g/G,b/N
```

```
e
r/G,o/P,y/S,g/A,b/N
r/P,y/S,o/G,g/N,b/A
r/P,o/S,y/A,g/G,b/N
r/P,o/G,y/S,g/A,b/N
ecclesiastical
#
```

Sample output

```
1
4
```

Problem No. 11: All Squares

Geometrically, any square has a unique, well-defined centre point. On a grid this is only true if the sides of the square are an odd number of points long. Since any odd number can be written in the form 2k+1, we can characterise any such square by specifying k, that is we can say that a square whose sides are of length 2k+1 has size k. Now define a pattern of squares as follows.

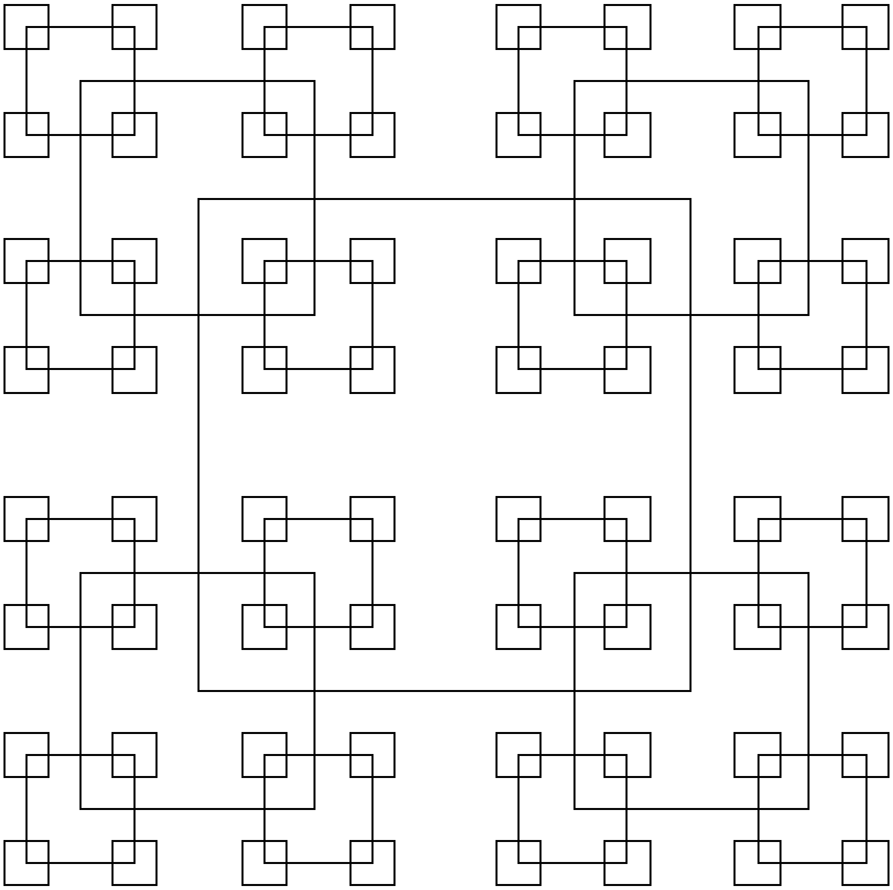
The largest square is of size k (that is sides are of length 2k+1) and is centred in a grid of size 1024 (that is the grid sides are of length 2049).

The smallest permissible square is of size 1 and the largest is of size 512, thus $1 \leq k \leq 512$.

All squares of size $k > 1$ have a square of size k div 2 centred on each of their 4 corners. (Div implies integer division, thus 9 div 2 = 4).

The top left corner of the screen has coordinates (0,0), the bottom right has coordinates (2048, 2048).

Hence, given a value of k, we can draw a unique pattern of squares according to the above rules. Furthermore any point on the screen will be surrounded by zero or more squares. (If the point is on the border of a square, it is considered to be surrounded by that square). Thus if the size of the largest square is given as 15, then the following pattern would be produced.



Write a program that will read in a value of k and the coordinates of a point, and will determine how many squares surround the point.

Input and Output

Input will consist of a series of lines. Each line will consist of a value of k and the coordinates of a point. 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 line of the input. Each line will consist of the number of squares containing the specified point, right justified in a field of width 3.

Sample input

```
500 113 941
0 0 0
```

Sample output

```
  5
```

Problem No. 12: 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: relatives' 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.

Input

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

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 No. 13: 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

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

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
```

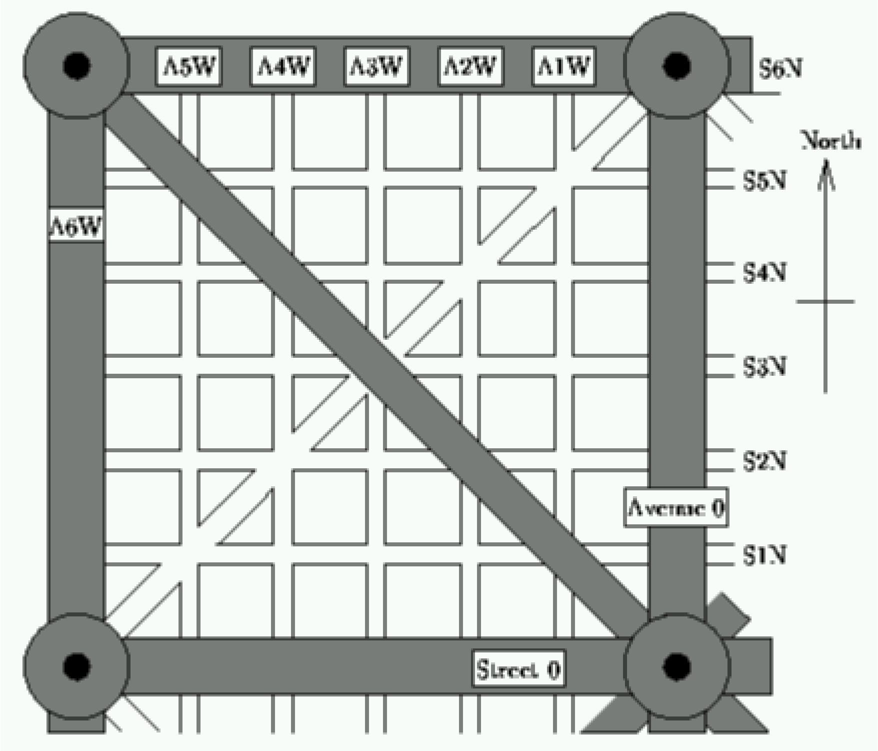
Sample output

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

Problem No. 14: 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 ≤ 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 Twenty five. 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

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

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

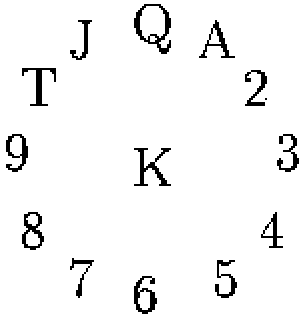
```
A2W S1N E
TURN SHARP LEFT
GO 1
TURN RIGHT
TURN LEFT
TURN SHARP LEFT
GO 1
TURN LEFT
STOP
A2W S1N W
GO STRAIGHT 2
TURN LEFT
GO ON 2
TURN HALF LEFT
TURN LEFT
GO 2
STOP
END
```

Sample output

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

Problem No. 15: Clock Patience

Card sharp Albert (Foxy) Smith is writing a book on patience games. To double check the examples in the book, he is writing programs to find the optimal play of a given deal. The description of Clock Patience reads as follows: ``The cards are dealt out (face down) in a circle, representing a clock, with a pile in each hour position and an extra pile in the centre of the clock. The first card goes face down on one o'clock, the next on two, and so on clockwise from there, with each thirteenth card going to the centre of the clock. This results in thirteen piles, with four cards face down in each pile.



The game then starts. The top card of the `king' pile (the last card dealt) is exposed to become the current card. Each move thereafter consists of placing the current card face up beneath the pile corresponding to its value and exposing the top card of that pile as the new current card. Thus if the current card is an Ace it is placed under the `one' pile and the top card of that pile becomes the current card. The game ends when the pile indicated by the current card has no face down cards in it. You win if the entire deck is played out, i.e. exposed."

Write a program that will read in a number of shuffled decks, and play the game.

Input and Output

The input will consist of decks of cards arranged in four lines of 13 cards, cards separated by a single blank. Each card is represented by two characters, the first is the rank (A, 2, 3, 4, 5, 6, 7, 8, 9, T, J, Q, K) followed by the suit (H, D, C, S). The input will be terminated by a line consisting of a single #. The deck is listed from bottom to top, so the first card dealt is the last card listed.

The output will consist of one line per deck. Each line will contain the number of cards exposed during the game (2 digits, with a leading zero if necessary), a comma, and the last card exposed (in the format used in the input).

Sample Input

```
TS QC 8S 8D QH 2D 3H KH 9H 2H TH KS KC
9D JH 7H JD 2S QS TD 2C 4H 5H AD 4D 5D
6D 4S 9S 5S 7S JS 8H 3D 8C 3S 4C 6S 9C
AS 7C AH 6H KD JC 7D AC 5C TC QD 6C 3C
#
```

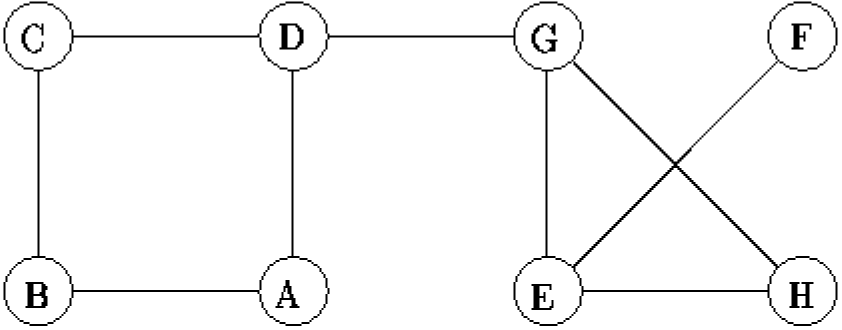
Sample output

```
44,KD
```

Problem No. 16: Network Wars

It is the year 2126 and comet Swift-Tuttle has struck the earth as predicted. The resultant explosion emits a large cloud of high energy neutrons that eliminates all human life. The accompanying electro-magnetic storm causes two unusual events: many of the links between various parts of the electronic network are severed, and some postgraduate AI projects begin to merge and mutate, in much the same way as animal life did several million years ago. In a very short time two programs emerge, Paskill and Lisper, which move through the network marking each node they visit: Paskill activates a modified Prolog interpreter and Lisper activates the `Hello World' program. However `Hello World' has mutated into an endless loop that so ties up the node that no other program, not even Lisper, can re-enter that node and the Prolog interpreter immediately reverse compiles (and destroys) any program that enters. However, Paskill knows which nodes it has visited and never tries to re-enter them. Thus if Lisper attempts to enter a node already visited by Paskill it will be annihilated; neither can enter a node already visited by Lisper, if either (or both) cannot move both will halt and if they ever arrive at a node simultaneously they annihilate each other. Both programs move through the network at the same speed.

Write a program to simulate these events. All nodes in the network are labelled with a single uppercase letter as shown below. When moving to the next node, Paskill searches alphabetically forwards from the current node, whereas Lisper searches alphabetically backwards from the current node, both wrapping round if necessary. Thus, (in the absence of the other) if Paskill enters the network below at A, it would visit the nodes in the order A, B, C, D, G, H, E, F; if Lisper enters the network at H it would visit them in the order H, G, E, F. Simulation stops when one or more of the above events occurs. If more than one event occurs, mention Paskill first.



Input

Input will consist of a series of lines. Each line will describe a network and indicate the starting nodes for the two programs. A network is described as a series of nodes separated by `;' and terminated by a period (`.'). Each node is described by its identifier, a `:' and one or more of the nodes connected to it. Each link will be mentioned at least once, as will each node, although not all nodes will be `described'. After the period will appear the labels of the starting nodes--first Paskill and then Lisper. No line will contain more than 255 characters. The file will be terminated by a line consisting of a single #.

Output

Output will consist of one line for each network. Each line will specify the terminating event and the node where it occurs. The terminating event is one or two of the following:

Lisper destroyed in node ?

{Paskill/Lisper} trapped in node ?
Both annihilated in node ?

Sample input

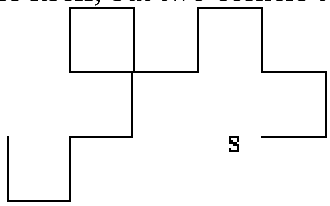
A:BD;C:BD;F:E;G:DEH;H:EG. A H
E:AB. A B
B:ACD. B D
A:B;B:C;D:E. A D
#

Sample output

Paskill trapped in node D Lisper trapped in node F
Both annihilated in node E
Lisper destroyed in node B
Lisper trapped in node E

Problem No. 17: Paper Folding

If a large sheet of paper is folded in half, then in half again, etc, with all the folds parallel, then opened up flat, there are a series of parallel creases, some pointing up and some down, dividing the paper into fractions of the original length. If the paper is only opened ``half-way'' up, so every crease forms a 90 degree angle, then (viewed end-on) it forms a ``dragon curve''. For example, if four successive folds are made, then the following curve is seen (note that it does not cross itself, but two corners touch):



Write a program to draw the curve which appears after *N* folds. The exact specification of the curve is as follows:

The paper starts flat, with the ``start edge'' on the left, looking at it from above.

The right half is folded over so it lies on top of the left half, then the right half of the new double sheet is folded on top of the left, to form a 4-thick sheet, and so on, for *N* folds.

Then every fold is opened from a 180 degree bend to a 90 degree bend.

Finally the bottom edge of the paper is viewed end-on to see the dragon curve.

From this view, the only unchanged part of the original paper is the piece containing the ``start edge'', and this piece will be horizontal, with the ``start edge'' on the left. This uniquely defines the curve. In the above picture, the ``start edge'' is the left end of the rightmost bottom horizontal piece (marked `s'). Horizontal pieces are to be displayed with the underscore character ``_'', and vertical pieces with the ``|'' character.

Input

Input will consist of a series of lines, each with a single number *N* ($1 \leq N \leq 13$). The end of the input will be marked by a line containing a zero.

Output

Output will consist of a series of dragon curves, one for each value of *N* in the input. Your picture must be shifted as far left, and as high as possible. Note that for large *N*, the picture will be greater than 80 characters wide, so it will look messy on the screen. The pattern for each different number of folds is terminated by a line containing a single `^'.

Sample input

```
2
4
1
0
```

Sample output

```
|_
 _|
^
   _   _
  |_|_| |_
   _|    _|
|_|
^
_|
^
```

Problem No. 18: Circle Through Three Points

Your team is to write a program that, given the Cartesian coordinates of three points on a plane, will find the equation of the circle through them all. The three points will not be on a straight line.

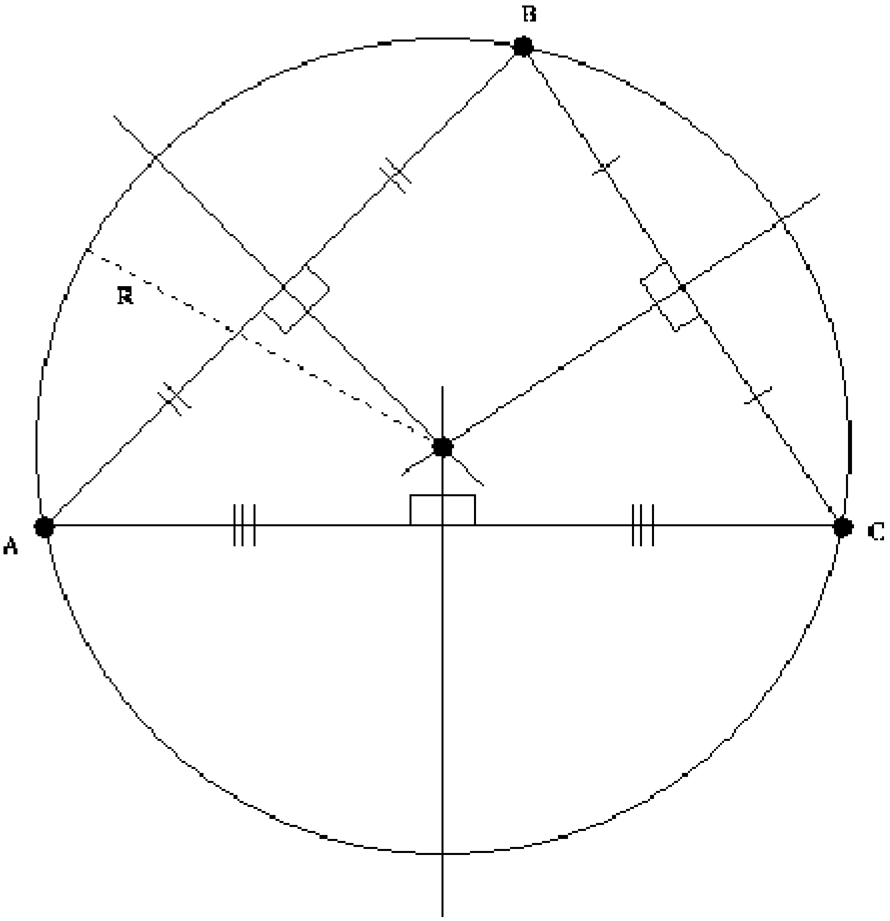
The solution is to be printed as an equation of the form

$$(x-h)^2+(y-k)^2=r^2 \qquad (1)$$

and an equation of the form

$$x^2+y^2+cx+dy+e=0 \qquad (2)$$

Each line of input to your program will contain the *x* and *y* coordinates of three points, in the order A_x, A_y, B_x, B_y, C_x, C_y. These coordinates will be real numbers separated from each other by one or more spaces.



Your program must print the required equations on two lines using the format given in the sample below. Your computed values for *h*, *k*, *r*, *c*, *d*, and *e* in Equations 1 and 2 above are to be printed with three digits after the decimal point. Plus and minus signs in the equations should be changed as needed to avoid multiple signs before a number. Plus, minus, and equal signs must be separated from the adjacent characters by a single space on each side. No other spaces are to appear in the equations. Print a single blank line after each equation pair.

Sample input

```
7.0 -5.0 -1.0 1.0 0.0 -6.0
1.0 7.0 8.0 6.0 7.0 -2.0
```

Sample output

```
(x - 3.000)^2 + (y + 2.000)^2 = 5.000^2
x^2 + y^2 - 6.000x + 4.000y - 12.000 = 0

(x - 3.921)^2 + (y - 2.447)^2 = 5.409^2
x^2 + y^2 - 7.842x - 4.895y - 7.895 = 0
```

Problem No. 19: Synchronous Design

The designers of digital integrated circuits (IC) are very concerned about the correctness of their designs because, unlike software, ICs cannot be easily tested. Real tests are not possible until the design has been finalized and the IC has been produced.

To simulate the behaviour of a digital IC and to more or less guarantee that the final chip will work, all of today's digital ICs are based on a *synchronous design*.

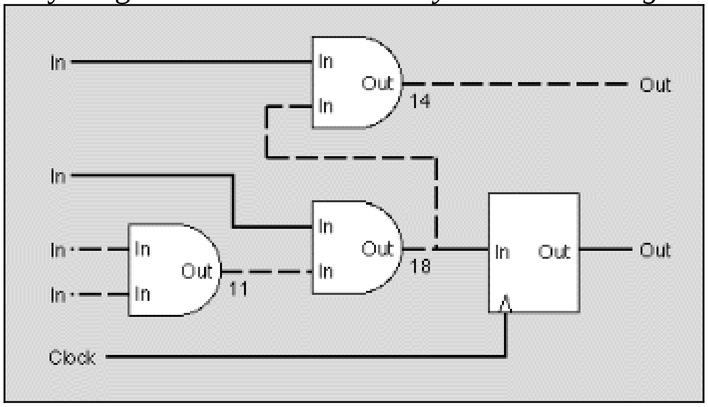


Figure: The critical path (dashed line) takes 43ns to settle

In a synchronous design, an external clock signal triggers the IC to go from a well defined and stable state to the next one. On the active edge of the clock, all input and output signals and all internal nodes are stable in either the high or low state. Between two consecutive edges of the clock, the signals and nodes are allowed to change and may take any intermediate state. The behaviour of a synchronous network is predictable and will not fail due to hazards or glitches introduced by irregularities of the real circuit.

To analyze whether an IC has a synchronous design, we distinguish between *synchronous* and "*asynchronous nodes*". Flip flops are synchronous nodes. On the active edge of the clock, the output of the flip flop changes to the state of the input and holds that state throughout the next clock cycle. Synchronous nodes are connected to the clock signal.

Simple gates like ANDs or ORs are asynchronous nodes. Their output changes - with a short delay - whenever one of their inputs changes. During that transition phase, the output can even go into some undefined or intermediate state.

For simplicity, we assume that all inputs of the circuits are directly connected to the output of a synchronous node outside the circuit and that all outputs of the circuit are directly connected to the input of a synchronous node outside the circuit.

For an IC to have a synchronous design, mainly two requirements must be met:

The "*signal delay*" introduced between two synchronous nodes must be smaller or equal than the clock period so there is enough time for nodes to become stable. In figure 1, the round-ended boxes are asynchronous nodes whereas the square boxes are synchronous nodes. The delay introduced on the dashed path is 43ns and exceeds the given clock period of 30ns.

There may be "*n o cycles*" composed exclusively of asynchronous nodes. In the real circuit such cycles could oscillate. In figure 2, the dashed path constitutes a cycle of asynchronous nodes.

Figure 3 shows a circuit with a synchronous design. It does not contain cycles composed of asynchronous nodes and the longest path between two synchronous nodes is shorter than the clock period of 30ns.

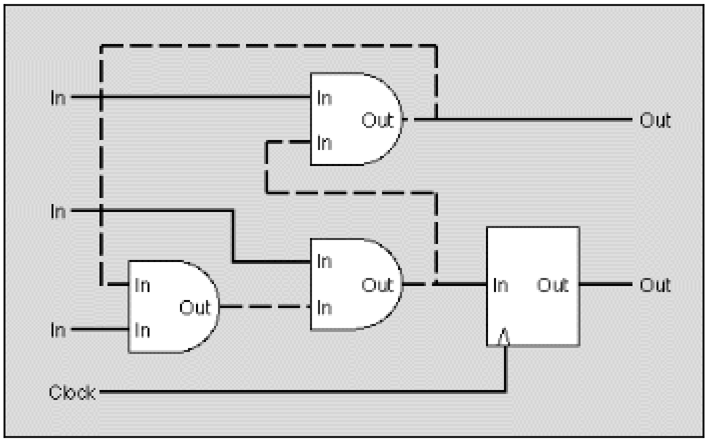


Figure: The design contains a cycle (dashed line)

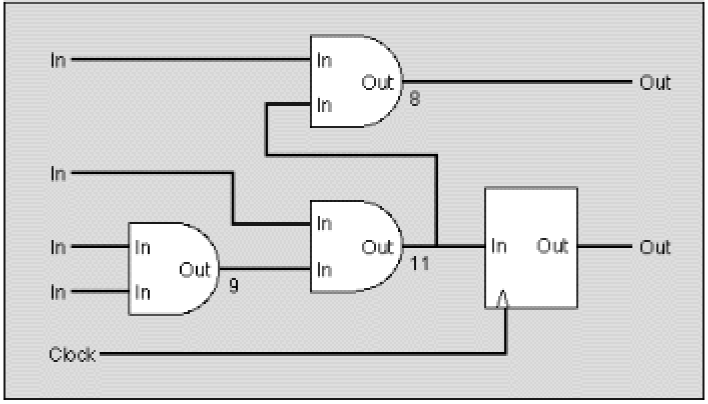


Figure: A synchronous design

Your are to write a program that decides for a given IC whether it has a synchronous design or not. You are given a network of synchronous and asynchronous nodes, a delay for each node, some inputs and outputs and the clock period.

You may safely assume that

the delays introduced between any input and any output of the same node are equal, i.e. equal to the delay given for that node,

synchronous nodes have no delay at all,

all connections between two nodes connect an output to an input.

Input

The input file contains several circuits. The first line gives the number of circuits in the file.

For each circuit in the file, the first line contains the clock period for the circuit, given as an integer number in nanoseconds. The next line gives the number of nodes. The following lines each contain a node, described by a letter and a integer number. The letter is 'i' for an input, 'o' for an output, 'a' for an asynchronous node and 's' for a synchronous node. The number gives the delay introduced by the node as an integer number in nanoseconds (only meaningful for an asynchronous node). Nodes are implicitly numbered, starting at zero.

After the nodes, the number of connections for the circuit follows. Each following line contains a pair of integer numbers denoting a connection between the output and the input of two nodes. The connection links an output of the node given by the first number and an input of the node given by the second number.

The clock signal is not given in the input file. We assume that all synchronous nodes are properly connected to the clock signal.

Output

For each circuit in the input file, your output file should contain a line with one of the following messages:

"Synchronous design. Maximum delay: <ss>." if the circuit has a synchronous design.

<*ss*> should be replaced by the longest delay found on any path between two synchronous nodes.

"Circuit contains cycle." if the circuit contains a cycle composed exclusively of asynchronous nodes.

"Clock period exceeded." if there is a path between two synchronous nodes that is longer than the given clock period and there are no cycles composed of asynchronous nodes.

Sample Input

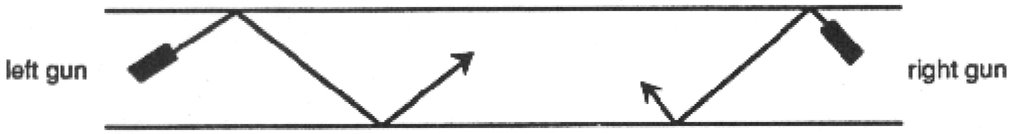
```
1
30
10
i 0
i 0
i 0
i 0
o 0
o 0
a 9
a 11
a 8
s 0
9
0 8
1 7
2 6
3 6
6 7
7 8
8 4
7 9
9 5
```

Sample Output

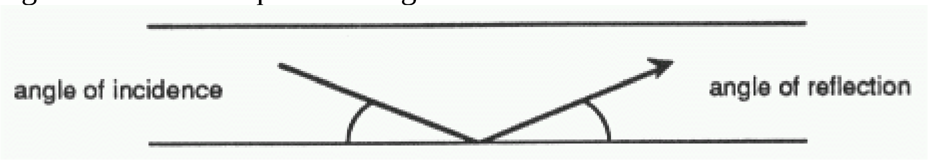
```
Synchronous design. Maximum delay: 28.
```

Problem No. 20: Robot Crash

The DoD company has contracted to determine under what conditions a pair of scanner robots can collide. The robots are fired simultaneously from ``guns" that are mounted near opposite ends of a horizontal strip. They travel in straight lines until they hit a wall of the strip or until they are in the same spot at the approximately the same time.



Whenever a robot hits a wall, it bounces off without loss of speed and in a straight line so that the angle of incidence equals the angle of reflection.



If the robots are in the same spot at approximately the same time, then they ``collide." Write a program to determine whether robots collide and if so where. To simplify the computer model of the physical problem, assume the following.

1) The horizontal strip is 2-dimensional, and it runs left-to-right. Its walls are straight lines.

2) Each robot is a point mass. That is, the circumference of each robot is 0.

3) A robot maintains the speed with which it was originally fired until it collides with the other robot or until it passes the gun from which the other robot was fired.

4) There are 2 guns, one mounted to the left of the other on a horizontal strip. The initial angle of the left gun is between -85 °and 85 °. The initial angle of the right gun is between 95 °and 180 °or -95 °and -180 °. (All angles are measured counterclockwise from the positive *x*-axis.)

5) Robots collide when they pass through the same place within 0.5 second of each other.

6) The horizontal strip is 10 units high. For any point (*x*,*y*) in the strip, $0 \leq y \leq 10$.

7) Robots speeds will be positive.

Input

Input for your program is a text file which contains data for several different pairs of robots. The lines of the text file come in pairs. The first line of a pair gives initial firing information about the robot fired from the leftmost gun. The second line of the pair gives initial firing information about the robot fired from the rightmost gun. Each line contains 4 data items as follows:

x-coordinate   *y*-coordinate   angle in degrees   speed (reals)

The end of input is indicated by end-of-file. Assume that the input is error-free.

Output (To help maintain floating point accuracy when converting degrees to radians, use the predeclared constant pi).

For each robot problem, output from your program should consist of the number of the problem (ex: Robot Problem #1, Robot Problem #2), and a statement indicating whether or not the robots do collide. If they do collide, your program should also print the coordinates of the point of collision. If there are multiple collisions, only print the first one. If there are multiple collisions at the same time, only print the one with smallest x-coordinate. All real output should be printed with 2 digits to the right of the decimal. Print a blank line after each robot problem. NOTE: Assume that a=b only if $|a-b| < \varepsilon=10^{-7}$

Sample Input

```
0  4  0  3.3
40  5  125  5
1  6  -5  10
```

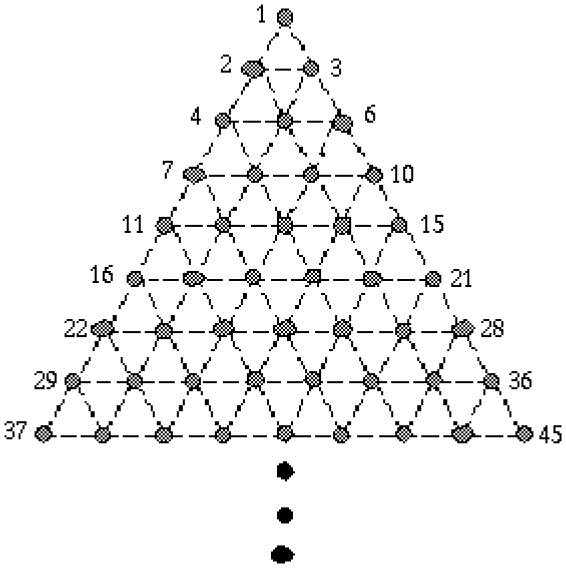
```
5  2  95  20
2  5  45  5
42 5  -135  5
0 6 20 3
0 5 180 4
```

Sample Output

```
Robot Problem #1: Robots do not collide.
Robot Problem #2: Robots collide at (4.68,5.68)
Robot Problem #3: Robots collide at (22.00,5.00)
Robot Problem #4: Robots do not collide.
```

Problem No. 21: Triangular Vertices

Consider the points on an infinite grid of equilateral triangles as shown below:



Note that if we number the points from left to right and top to bottom, then groups of these points form the vertices of certain geometric shapes. For example, the sets of points {1,2,3} and {7,9,18} are the vertices of triangles, the sets {11,13,26,24} and {2,7,9,18} are the vertices of parallelograms, and the sets {4,5,9,13,12,7} and {8,10,17,21,32,34} are the vertices of hexagons.

Write a program which will repeatedly accept a set of points on this triangular grid, analyze it, and determine whether the points are the vertices of one of the following ``acceptable" figures: triangle, parallelogram, or hexagon. In order for a figure to be acceptable, it must meet the following two conditions:

1) Each side of the figure must coincide with an edge in the grid.

and 2) All sides of the figure must be of the same length.

Input

The input will consist of an unknown number of point sets. Each point set will appear on a separate line in the file. There are at most six points in a set and the points are limited to the range 1..32767.

Output

For each point set in the input file, your program should deduce from the number of points in the set which geometric figure the set potentially represents; e.g., six points can only represent a hexagon, etc. The output must be a series of lines listing each point set followed by the results of your analysis.

Sample Input

```
1 2 3
11 13 29 31
26 11 13 24
4 5 9 13 12 7
1 2 3 4 5
47
11 13 23 25
```

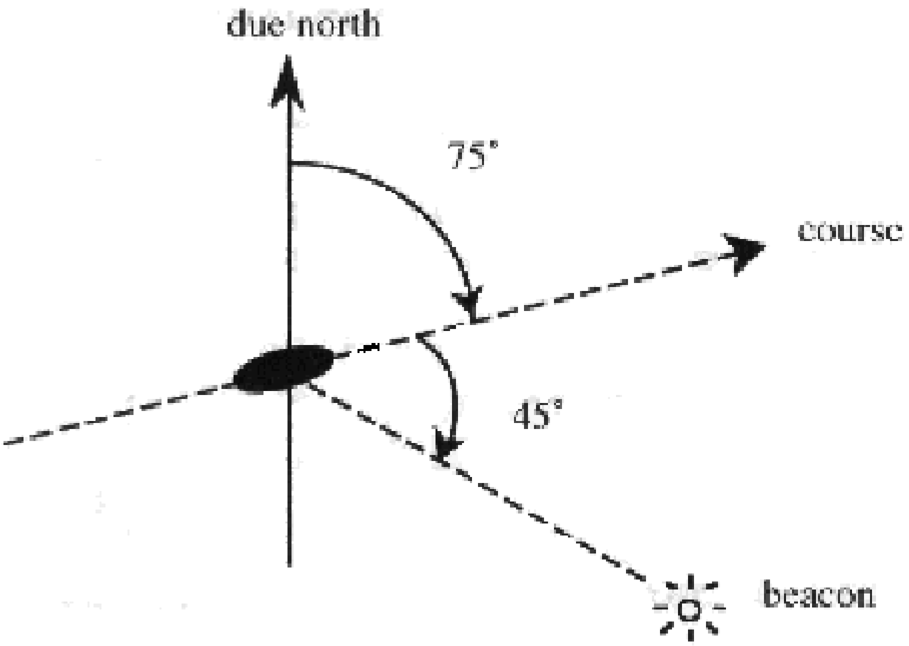
Sample Output

```
1 2 3 are the vertices of a triangle
11 13 29 31 are not the vertices of an acceptable figure
26 11 13 24 are the vertices of a parallelogram
4 5 9 13 12 7 are the vertices of a hexagon
1 2 3 4 5 are not the vertices of an acceptable figure
47 are not the vertices of an acceptable figure
11 13 23 25 are not the vertices of an acceptable figure
```

Problem No. 22: Radio Direction Finder

A boat with a directional antenna can determine its present position with the help of readings from local beacons. Each beacon is located at a known position and emits a unique signal. When a boat detects a signal, it rotates its antenna until the signal is at maximal strength. This gives a relative bearing to the position of the beacon. Given a previous beacon reading (the time, the relative bearing, and the position of the beacon), a new beacon reading is usually sufficient to determine the boat's present position. You are to write a program to determine, when possible, boat positions from pairs of beacon readings.

For this problem, the positions of beacons and boats are relative to a rectangular coordinate system. The positive *x*-axis points east; the positive *y*-axis points north. The course is the direction of travel of the boat and is measured in degrees clockwise from north. That is, north is 0 □, east is 90 □, south is 180 □, and west is 270 □. The relative bearing of a beacon is given in degrees clockwise relative to the course of the boat. A boat's antenna cannot indicate on which side the beacon is located. A relative bearing of 90 □means that the beacon is toward 90 □or 270 □.



The boat's course is 75°.
The beacon has a relative bearing
of 45° from the boat's course.

Input

The input consists of several datasets. The first line of each dataset is an integer specifying the number of beacons (at most 30). Following that is a line for each beacon. Each of those lines begins with the beacon's name (a string of 20 or fewer alphabetic characters), the *x*-coordinate of its position, and the *y*-coordinate of its position. These fields are single-space separated.

Coming after the lines of beacon information is an integer specifying a number of boat scenarios to follow. A boat scenario consists of three lines, one for velocity and two for beacon readings.

Data on input line	Meaning of data
course speed	the boat's course, the speed at which it is traveling
time#1 name#1 angle#1	time of first beacon reading, name of first beacon, relative bearing of first beacon
time#2 name#2 angle#2	time of second reading, name of second beacon, relative bearing of second beacon

All times are given in minutes since midnight measured over a single 24-hour period. The speed is the distance (in units matching those on the rectangular coordinate system) over time. The second line of a scenario gives the first beacon reading as the time of the reading (an integer), the name of the beacon, and the angle of the reading as measured from the boat's course. These 3 fields have single space separators. The third line gives the second beacon reading. The time for that reading will always be at least as large as the time for the first reading.

Output

For each scenario, your program should print the scenario number (Scenario 1, Scenario 2, etc.) and a message indicating the position (rounded to 2 decimal places) of the boat as of the time of the *second* beacon reading. If it is impossible to determine the position of the boat, the message should say ``Position cannot be determined." Sample input and corresponding correct output are shown below.

Sample Input

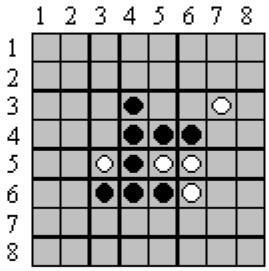
```
4
First 2.0 4.0
Second 6.0 2.0
Third 6.0 7.0
Fourth 10.0 5.0
2
0.0 1.0
1 First 270.0
2 Fourth 90.0
116.5651 2.2361
4 Third 126.8699
5 First 319.3987
4
First 2.0 4.0
Second 6.0 2.0
Third 6.0 7.0
Fourth 10.0 5.0
1
0.0 1.0
1 First 270.0
2 Fourth 90.0
```

Sample Output

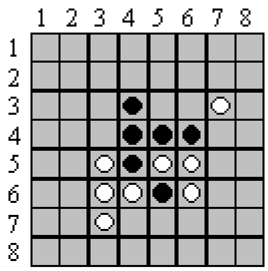
```
Scenario 1: Position cannot be determined
Scenario 2: Position is (6.00, 5.00)
Scenario 3: Position cannot be determined
```

Problem No. 23: Othello

Othello is a game played by two people on an 8 x 8 board, using disks that are white on one side and black on the other. One player places disks with the white side up and the other player places disks with the black side up. The players alternate placing one disk on an unoccupied space on the board. In placing a disk, the player must bracket at least one of the other colour disks. Disks are bracketed if they are in a straight line horizontally, vertically, or diagonally, with a disk of the current player's colour at each end of the line. When a move is made, all the disks that were bracketed are changed to the colour of the player making the move. (It is possible that disks will be bracketed across more than one line in a single move.)



Legal Moves for White
(2,3), (3,3), (3,5),
(6,2), (7,3), (7,4),



Board Configuration after
White Moves to (7,3)

On the left
Legal Moves for White
(2,3),(3,3),(3,5),(3,6)
(6,2),(7,3),(7,4),(7,5)

On the right
Board Configuration after
White Moves to (7,3)

Write a program to read a series of Othello games. The first line of the input is the number of games to be processed. Each game consists of a board configuration followed by a list of commands. The board configuration consists of 9 lines. The first 8 specify the current state of the board. Each of these 8 lines contains 8 characters, and each of these characters will be one of the following:

`-'	indicating an unoccupied square
`B'	indicating a square occupied by a black disk
`W'	indicating a square occupied by a white disk

The ninth line is either a `B' or a `W' to indicate which is the current player. You may assume that the data is legally formatted.

The commands are to list all possible moves for the current player, make a move, or quit the current game. There is one command per line with no blanks in the input. Commands are formatted as follows:

List all possible moves for the current player.

The command is an `L' in the first column of the line. The program should go through the board and print all legal moves for the current player in the format (*x*,*y*) where *x* represents the row of the legal move and *y* represents its column. These moves should be printed in row major order which means:

1) all legal moves in row number *i* will be printed before any legal move in row number *j* if *j* is greater than *i*

and 2) if there is more than one legal move in row number *i*, the moves will be printed in ascending order based on column number.

All legal moves should be put on one line. If there is no legal move because it is impossible for the current player to bracket any pieces, the program should print the message ``No legal move."

Make a move.

The command is an `M' in the first column of the line, followed by 2 digits in the second and third column of the line. The digits are the row and the column of the space to place the piece of the current player's colour, *unless the current player has no legal move*. If the current player has no legal move, the current player is first changed to the other player and the move will be the move of the new current player. You may assume that the move is then legal. You should record the changes to the board, including adding the new piece and changing the colour of all bracketed pieces. At the end of the move, print the number of pieces of each colour on the board in the format ``Black - xx White - yy" where xx is the number of black pieces on the board and yy is the number of white pieces on the board. After a move, the current player will be changed to the player that did not move.

Quit the current game.

The command will be a `Q' in the first column of the line. At this point, print the final board configuration using the same format as was used in the input. This terminates input for the current game.

You may assume that the commands will be syntactically correct. Put one blank line between output from separate games and no blank lines anywhere else in the output.

Sample Input

```
2
--------
--------
--------
---WB---
---BW---
--------
--------
--------
W
L
M35
L
Q
WWWWB---
WWWB----
WWB-----
WB------
--------
--------
--------
--------
B
L
M25
L
Q
```

Sample Output

```
(3,5) (4,6) (5,3) (6,4)
Black -  1 White -  4
(3,4) (3,6) (5,6)
--------
--------
----W---
---WW---
---BW---
--------
--------
--------
```

```
No legal move.
Black -  3 White - 12
(3,5)
WWWWB---
WWWWW---
WWB-----
WB------
--------
--------
--------
--------
```

Problem No. 24: Golygons

Imagine a country whose cities have all their streets laid out in a regular grid. Now suppose that a tourist with an obsession for geometry is planning expeditions to several such cities.

Starting each expedition from the central cross-roads of a city, the intersection labelled (0,0), our mathematical visitor wants to set off north, south, east or west, travel one block, and view the sights at the intersection (0,1) after going north, (0,-1) after going south, (1,0) after going east or (-1,0) after going west. Feeling ever more enthused by the regularity of the city, our mathematician would like to walk a longer segment before stopping next, going two blocks.

What's more, our visitor doesn't want to carry on in the same direction as before, nor visit the same point twice, nor wishes to double back, so will make a 90 □turn either left or right. The next segment should be three blocks, again followed by a right-angle turn, then four, five, and so on with ever-increasing lengths until finally, at the end of the day, our weary traveller returns to the starting point, (0,0).

The possibly self-intersecting figure described by these geometrical travels is called a golygon.

Unfortunately, our traveller will be making these visits in the height of summer when road works will disrupt the stark regularity of the cities' grids. At some intersections there will be impassable obstructions. Luckily, however, the country's limited budget means there will never be more than 50 road works blocking the streets of any particular city. In an attempt to gain accountability to its citizens, the city publishes the plans of road works in advance. Our mathematician has obtained a copy of these plans and will ensure that no golygonal trips get mired in molten tar.

Write a program that constructs all possible golygons for a city.

Input

Since our tourist wants to visit several cities, the input file will begin with a line containing an integer specifying the number of cities to be visited.

For each city there will follow a line containing a positive integer not greater than 20 indicating the length of the longest edge of the golygon. That will be the length of the last edge which returns the traveler to (0,0). Following this on a new line will be an integer from 0 to 50 inclusive which indicates how many intersections are blocked. Then there will be this many pairs of integers, one pair per line, each pair indicating the *x* and *y* coordinates of one blockage.

Output

For each city in the input, construct all possible golygons. Each golygon must be represented by a sequence of characters from the set {n,s,e,w} on a line of its own, and they should be output in lexicographics order. Following the list of golygons should be a line indicating how many solutions were found. This line should be formatted as shown in the example output. A blank line should appear following the output for each city.

Sample Input

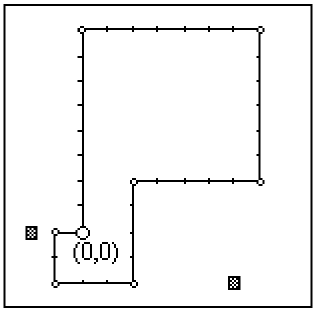
```
2
8
2
-2 0
6 -2
8
2
2 1
-2 0
```

Sample Output

```
wsenenws
Found 1 golygon(s).

Found 0 golygon(s).
```

Diagram of the 1st City



Problem No. 25: Crossword Answers

A crossword puzzle consists of a rectangular grid of black and white squares and two lists of definitions (or descriptions).

One list of definitions is for ``words" to be written left to right across white squares in the rows and the other list is for words to be written down white squares in the columns. (A word is a sequence of alphabetic characters.)

To solve a crossword puzzle, one writes the words corresponding to the definitions on the white squares of the grid.

The definitions correspond to the rectangular grid by means of sequential integers on ``eligible" white squares. White squares with black squares immediately to the left or above them are ``eligible." White squares with no squares either immediately to the left or above are also ``eligible." No other squares are numbered. All of the squares on the first row are numbered.

The numbering starts with 1 and continues consecutively across white squares of the first row, then across the eligible white squares of the second row, then across the eligible white squares of the third row and so on across all of the rest of the rows of the puzzle. The picture below illustrates a rectangular crossword puzzle grid with appropriate numbering.

1	2	3	■	4	5	6
■	7		■	8		
9			10	■	11	
12		■	13	14		
■	15	16	■	17		■
18			■	19		20

An ``across" word for a definition is written on a sequence of white squares in a row starting on a numbered square that does not follow another white square in the same row.

The sequence of white squares for that word goes across the row of the numbered square, ending immediately before the next black square in the row or in the rightmost square of the row.

A ``down" word for a definition is written on a sequence of white squares in a column starting on a numbered square that does not follow another white square in the same column.

The sequence of white squares for that word goes down the column of the numbered square, ending immediately before the next black square in the column or in the bottom square of the column.

Every white square in a correctly solved puzzle contains a letter.

You must write a program that takes several solved crossword puzzles as input and outputs the lists of across and down words which constitute the solutions.

Input

Each puzzle solution in the input starts with a line containing two integers *r* and *c* ($1 \le r \le 10$ and $1 \le c \le 10$), where *r* (the first number) is the number of rows in the puzzle and *c* (the second number) is the number of columns.

The *r* rows of input which follow each contain *c* characters (excluding the end-of-line) which describe the solution. Each of those *c* characters is an alphabetic character which is part of a word or the character ``*", which indicates a black square.

The end of input is indicated by a line consisting of the single number 0.

Output

Output for each puzzle consists of an identifier for the puzzle (puzzle #1:, puzzle #2:, etc.) and the list of across words followed by the list of down words. Words in each list must be output one-per-line in increasing order of the number of their corresponding definitions.

The heading for the list of across words is ``Across". The heading for the list of down words is ``Down".

In the case where the lists are empty (all squares in the grid are black), the Across and Down headings should still appear.

Separate output for successive input puzzles by a blank line.

Sample Input

```
2 2
AT
*O
6 7
AIM*DEN
*ME*ONE
UPON*TO
SO*ERIN
*SA*OR*
IES*DEA
0
```

Sample Output

```
puzzle #1:
Across
  1.AT
  3.O
Down
  1.A
  2.TO

puzzle #2:
Across
  1.AIM
  4.DEN
  7.ME
  8.ONE
  9.UPON
 11.TO
 12.SO
 13.ERIN
 15.SA
 17.OR
 18.IES
 19.DEA
Down
  1.A
  2.IMPOSE
  3.MEO
  4.DO
  5.ENTIRE
  6.NEON
  9.US
 10.NE
 14.ROD
```

16.AS
18.I
20.A

Problem No. 26: VTAS - Vessel Traffic Advisory Service

In order to promote safety and efficient use of port facilities, the Association of Coastal Merchants (ACM) has developed a concept for a Vessel Traffic Advisory Service (VTAS) that will provide traffic advisories for vessels transiting participating ports.

The concept is built on a computer program that maintains information about the traffic patterns and reported movements of vessels within the port over multiple days. For each port, the traffic lanes are defined between waypoints. The traffic lanes have been designated as directional to provide traffic separation and flow controls. Each port is represented by a square matrix containing the distances (in nautical miles) along each valid traffic lane. The distances are defined from each row waypoint to each column waypoint. A distance of 0 indicates that no valid traffic lane exists between the two waypoints.

Vessel traffic enters the port at a waypoint and transits the traffic lanes. A vessel may begin its transit at any of the waypoints and must follow a valid connected route via the valid traffic lanes. A vessel may end its transit at any valid waypoint.

The service provided by the VTAS to transiting vessels includes:

Projection of arrival times at waypoints

Notification of invalid routes

Projected encounters with other vessels on each leg of the transit. An ``encounter" occurs when two vessels are between common waypoints (either traffic lane) at a common time

Warning of close passing with another vessel in the vicinity of a waypoint (within 3 minutes of projected waypoint arrival)

Reported times will be rounded to the nearest whole minute. Time is maintained based on a 24 hour clock (i.e. 9 am is 0900, 9 PM is 2100, midnight is 0000). Speed is measured in knots which is equal to 1 nautical mile per hour.

Input

The input file for the computer program includes several datasets, each containing a Port Specification to provide the description of the traffic patterns within the port and a Traffic List which contains the sequence of vessels entering the port and their intended tracks. The end of each dataset is indicated by a Vessel Name beginning with an ``*".

Port Specification:	Number of Waypoints in Port (an integer *N*)
	Waypoint ID List (*N* single-character designators)
	Waypoint Connection Matrix (*N* rows of *N* real values specifying the distances between waypoints in nautical miles)
Traffic List:	Vessel Name (alphabetic characters)
	Time at first waypoint (on 24-hour clock) & Planned Transit Speed (in knots)
	Planned Route (ordered list of waypoints)

Output

The output for each dataset shall provide for each vessel as it enters the port a listing indicating the arrival of the vessel and its planned speed followed by a table containing the waypoints in its route and projected arrival at each waypoint. Following this table will be appropriate messages indicating:

Notification of Invalid Routes

Projected Encounters on each leg

Warning of close passing at waypoints

All times are to be printed as four-digit integers with leading zeros when necessary. Print a blank line after each dataset.

Assumptions & Limitations:

1. Vessel names are at most 20 characters long.

2. There are at most 20 waypoints in a port and at most 20 waypoints in any route.

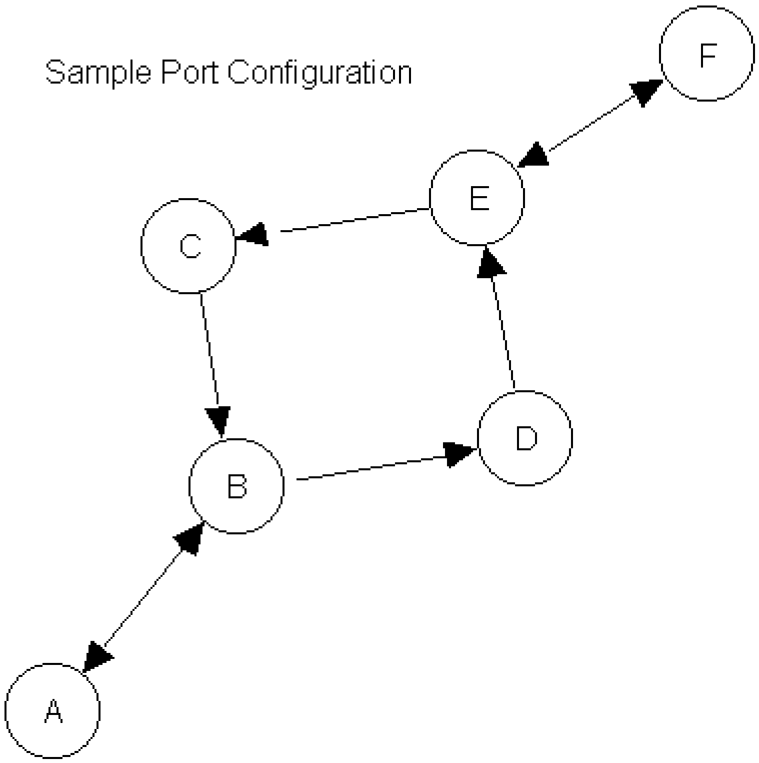
3. There will be at most 20 vessels in port at any time.

4. A vessel will complete its transit in at most 12 hours.

5. No more than 24 hours will elapse between vessel entries.

Sample Input

```
6 ABCDEF 0 3 0 0 0 0 3 0 0 2 0 0 0 3 0 0 0 0 0 0 0 0 3 0 0 0 2 0 0 4 0 0 0 0 4 0 Tug 2330
12 ABDEF WhiteSailboat 2345 6 ECBDE TugWBarge 2355 5 DECBA PowerCruiser 0
15 FECBA LiberianFreighter 7 18 ABDXF ChineseJunk
45 8 ACEF *****
```



Sample Output

```
Tug entering system at 2330 with a planned speed of 12.0 knots
        Waypoint:    A    B    D    E    F
        Arrival:   2330 2345 2355 0010 0030
```

```
WhiteSailboat entering system at 2345 with a planned speed of 6.0 knots
        Waypoint:   E   C   B   D   E
        Arrival:  2345 0005 0035 0055 0125

TugWBarge entering system at 2355 with a planned speed of 5.0 knots
        Waypoint:   D   E   C   B   A
        Arrival:  2355 0031 0055 0131 0207
Projected encounter with Tug on leg between Waypoints D & E
** Warning ** Close passing with Tug at Waypoint D

PowerCruiser entering system at 0000 with a planned speed of 15.0 knots
        Waypoint:   F   E   C   B   A
        Arrival:  0000 0016 0024 0036 0048
Projected encounter with Tug on leg between Waypoints F & E
Projected encounter with WhiteSailboat on leg between Waypoints C & B
** Warning ** Close passing with WhiteSailboat at Waypoint B

LiberianFreighter entering system at 0007 with a planned speed of 18.0 knots
**> Invalid Route Plan for Vessel: LiberianFreighter

ChineseJunk entering system at 0045 with a planned speed of 8.0 knots
**> Invalid Route Plan for Vessel: ChineseJunk
```

Problem No. 27: Cutting Corners

Bicycle messengers who deliver documents and small items to businesses have long been part of the guerrilla transportation services in several major U.S. cities. The cyclists of Boston are a rare breed of riders. They are notorious for their speed, their disrespect for one-way streets and traffic signals, and their brazen disregard for cars, taxis, buses, and pedestrians.

Bicycle messenger services are very competitive. Billy's Bicycle Messenger Service is no exception. To boost its competitive edge and to determine its actual expenses, BBMS is developing a new scheme for pricing deliveries that depends on the shortest route messengers can travel. You are to write a program to help BBMS determine the distances for these routes.

The following assumptions help simplify your task:

Messengers can ride their bicycles anywhere at ground level except inside buildings.

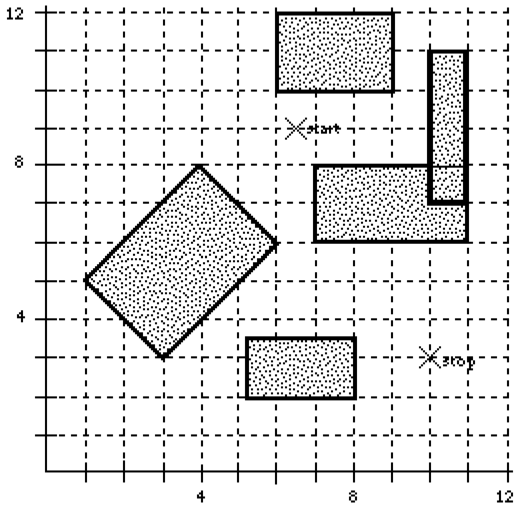
Ground floors of irregularly shaped buildings are modeled by the union of the interiors of rectangles. By agreement any intersecting rectangles share interior space and are part of the same building.

The defining rectangles for two separate buildings never touch, although they can be quite close. (Bicycle messengers- skinny to a fault-can travel between any two buildings. They can cut the sharpest corners and run their skinny tires right down the perimeters of the buildings.)

The starting and stopping points are never inside buildings.

There is always some route from the starting point to the stopping point.

Your program must be able to process several scenarios. Each scenario defines the buildings and the starting and stopping points for a delivery route. The picture below shows a bird's-eye view of a typical scenario.



Input

The input file represents several scenarios. Input for each scenario consists of lines as follows:

First line: *n*

The number of rectangles describing the buildings in the scenario. 0≤n≤20

Second line: x_1 y_1 x_2 y_2

The *x*- and *y*-coordinates of the starting and stopping points of the route.

Remaining *n* lines: x_1 y_1 x_2 y_2 x_3 y_3

The *x*- and *y*-coordinates of three vertices of a rectangle.

The *x*- and *y*-coordinates of all input data are real numbers between 0 and 1000 inclusive. Successive coordinates on a line are separated by one or more blanks. The integer -1 follows the data of the last scenario.

Output

Output should number each scenario (Scenario #1, Scenario #2, etc.) and give the distance of the shortest route from starting to stopping point as illustrated in the Sample Output below. The distance should be written with two digits to the right of the decimal point. Output for successive scenarios should be separated by a blank line.

Sample Input

```
5
6.5   9     10  3
1     5     3   3     6  6
5.25  2     8   2     8  3.5
6     10    6   12    9  12
7     6     11  6     11 8
10    7     11  7     11 11
-1
```

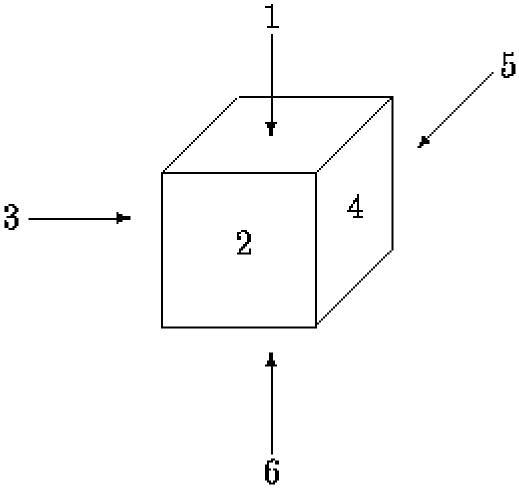
Sample Output

```
Scenario #1
```

route distance: 7.28

Problem No. 28: Cube Painting

We have a machine for painting cubes. It is supplied with three different colours: blue, red and green. Each face of the cube gets one of these colours. The cube's faces are numbered as in Figure.



Figure

Since a cube has 6 faces, our machine can paint a face-numbered cube in 3^6 = 729 different ways. When ignoring the face-numbers, the number of different paintings is much less, because a cube can be rotated. See example below. We denote a painted cube by a string of 6 characters, where each character is a b, r, or g. The i^{th}character (*1≤i≤6*) from the left gives the color of face *i*. For example, Figure 2 is a picture of rbgggr and Figure 3 corresponds to rggbgr. Notice that both cubes are painted in the same way: by rotating it around the vertical axis by 90 °, the one changes into the other.

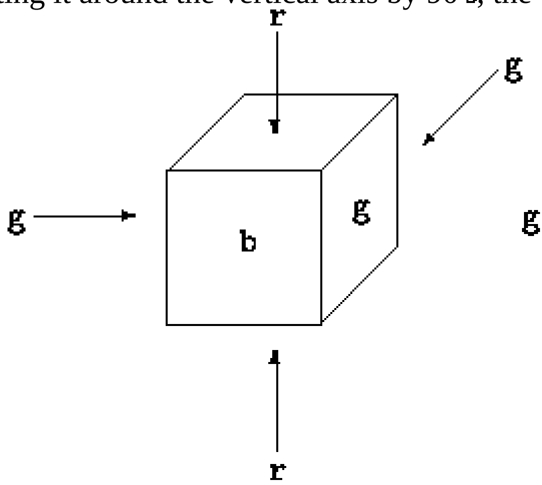


Figure 2.

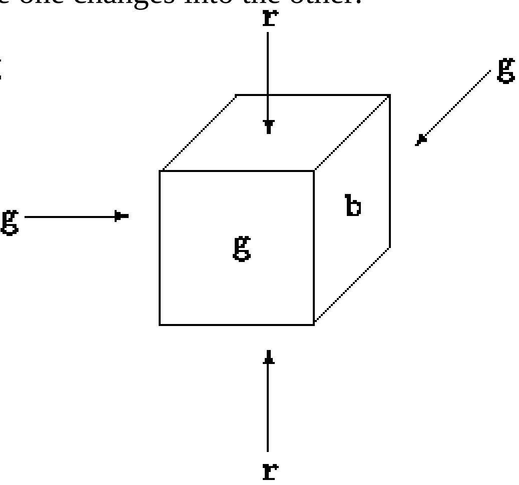


Figure 3.

Input

The input of your program is a textfile that ends with the standard end-of-file marker. Each line is a string of 12 characters. The first 6 characters of this string are the representation of a painted cube, the remaining 6 characters give you the representation of another cube. Your program determines whether these two cubes are painted in the same way, that is, whether by any combination of rotations one can be turned into the other. (Reflections are not allowed.)

Output

The output is a file of boolean. For each line of input, output contains TRUE if the second half can be obtained from the first half by rotation as describes above, FALSE otherwise.

Sample Input

```
rbgggrrggbgr
rrrbbbrrbbbr
rbgrbgrrrrrg
```

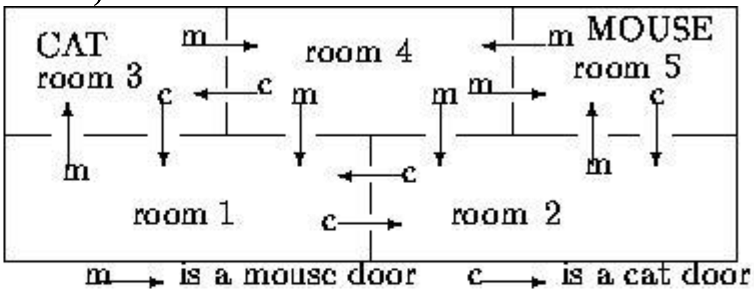
Sample Output

```
TRUE
FALSE
FALSE
```

Problem No. 29: Cat and Mouse

In a house with many rooms live a cat and a mouse. The cat and the mouse each have chosen one room as their ``home". From their ``home" they regularly walk through the house. A cat can go from room A to room B if and only if there is a cat door from room A to room B. Cat doors can only be used in one direction. Similarly a mouse can go from room A to room B if and only if there is a mouse door from room A to room B . Also mouse doors can be used in only one direction. Furthermore, cat doors cannot be used by a mouse, and mouse doors cannot be used by a cat.

Given a map of the house you are asked to write a program that finds out
if there exist walks for the cat and mouse where they meet each other in some room, and
if the mouse can make a walk through at least two rooms, end in its ``home" room again, and along the way cannot ever meet the cat. (Here, the mouse may not ever meet the cat, whatever the cat does.)



For example, in the map, the cat can meet the mouse in rooms 1, 2, and 3. Also, the mouse can make a walk through two rooms without ever meeting the cat, viz., a round trip from room 5 to 4 and back.

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 integers and defines the configuration of the house. The first line has three integers separated by blanks: the first integer defines the number of rooms, the second the initial room of the cat (the cat's ``home"), and the third integer defines the initial room of the mouse (the mouse's ``home"). Next there are zero or more lines, each with two positive integers separated by a blank. These lines are followed by a line with two -1's separated by a blank. The pairs of positive integers define the cat doors. The pair A B represents the presence of a cat door from room A to room B . Finally there are zero or more lines, each with two positive integers separated by a blank. These pairs of integers define the mouse doors. Here, the pair A B represents the presence of a mouse door from room A to room B .

The number of rooms is at least one and at most 100. All rooms are numbered consecutively starting at 1. You may assume that all positive integers in the input are legal room numbers.

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.

The output consists of two characters separated by a blank and ended by a new-line character. The first character is Y if there exist walks for the cat and mouse where they meet each other in some room. Otherwise, it is N. The second character is Y if the mouse can make a walk through at least two rooms, end in its ``home" room again, and along the way cannot ever meet the cat. Otherwise, it is N.

Sample Input (From example above)

```
1

5 3 5
1 2
2 1
3 1
4 3
5 2
-1 -1
1 3
2 5
3 4
4 1
4 2
4 5
5 4
```

Sample Output

```
Y Y
```

Problem No. 30: Egyptian Multiplication

In 1858, A. Henry Rhind, a Scottish antiquary, came into possession of a document which is now called the Rhind Papyrus. Titled ``Directions for Attaining Knowledge into All Obscure Secrets", the document provides important clues as to how the ancient Egyptians performed arithmetic.

There is no zero in the number system. There are separate characters denoting ones, tens, hundreds, thousands, ten-thousands, hundred-thousands, millions and ten-millions. For the purposes of this problem, we use near ASCII equivalents for the symbols:

| for one (careful, it's a vertical line, not 1)
n for ten
9 for hundred
8 for thousand
r for ten-thousand

(The actual Egyptian hieroglyphs were more picturesque but followed the general shape of these modern symbols. For the purpose of this problem, we will not consider numbers greater than 99,999.)

Numbers were written as a group of ones preceded in turn by groups of tens, hundreds, thousands and ten-thousands. Thus our number 4,023 would be rendered: ||| nn 8888. Notice that a zero digit is indicated by a group consisting of none of the corresponding symbol. The number 40,230 would thus be rendered: nnn 99 rrrr. (In the Rhind Papyrus, the groups are drawn more picturesquely, often spread across more than one horizontal line; but for the purposes of this problem, you should write numbers all on a single line.)

To multiply two numbers *a* and *b*, the Egyptians would work with two columns of numbers. They would begin by writing the number | in the left column beside the number *a* in the right column. They would proceed to form new rows by doubling the numbers in both columns. Notice that doubling can be effected by copying symbols and normalizing by a carrying process if any group of symbols is larger than 9 in size. Doubling would continue as long as the number in the left column does not exceed the other multiplicand *b*. The numbers in the first column that summed to the multiplicand *b* were marked with an asterisk. The numbers in the right column alongside the asterisks were then added to produce the result.

Below, we show the steps corresponding to the multiplication of 483 by 27:

```
| *                              ||| nnnnnnnn 9999
|| *                             |||||| nnnnnn 999999999
||||                             || nnn 999999999 8
|||||||| *                       |||| nnnnnn 99999999 888
|||||| n *                       |||||||| nn 9999999 8888888
```

The solution is: | nnnn 888 r

(The solution came from adding together:

```
||| nnnnnnnn 9999
|||||| nnnnnn 999999999
|||| nnnnnn 99999999 888
|||||||| nn 9999999 8888888.)
```

You are to write a program to perform this Egyptian multiplication.

Input

Input will consist of several pairs of nonzero numbers written in the Egyptian system described above. There will be one number per line; each number will consist of groups of symbols, and each group is terminated by a single space (including the last group). Input will be terminated by a blank line.

Output

For each pair of numbers, your program should print the steps described above used in Egyptian multiplication. Numbers in the left column should be flush with the left margin. Each number in the left and right column will be represented by groups of symbols, and each group is terminated by a single space (including the last group). If there is an asterisk in the left column, it should be separated from the end of the left number by a single space. Up to the 34th character position should then be filled with spaces. Numbers in the right column should begin at the 35th character position on the line and end with a newline character.

Test data will be chosen to ensure that no overlap can occur. After showing each of the doubling steps, your program should print the string: ``The solution is: '' followed by the product of the two numbers in Egyptian notation (modulus 100000).

Sample Input

```
||
||
|||
||||
nnnnnn 9
||| n
n
9
|||
8

```

Sample Output

```
|                                 ||
|| *                              ||||
The solution is: ||||
|                                 |||
||                                ||||||
|||| *                            || n
The solution is: || n
| *                               nnnnnn 9
||                                nn 999
|||| *                            nnnn 999999
|||||||| *                        nnnnnnnn 99 8
The solution is: nnnnnnnn 88
|                                 n
||                                nn
|||| *                            nnnn
||||||||                          nnnnnnnn
|||||| n                          nnnnnn 9
|| nnn *                          nn 999
```

```
|||| nnnnnn *
The solution is: 8
|
||
||||
|||||||| *
|||||| n
|| nnn *
|||| nnnnnn *
|||||||| nn 9 *
|||||| nnnnn 99 *
|| n 99999 *
The solution is: 888
```

```
nnnn 999999

|||
||||||
|| n
|||| nn
|||||||| nnnn
|||||| nnnnnnnnn
|| nnnnnnnnn 9
|||| nnnnnnnn 999
|||||||| nnnnnn 9999999
|||||| nnn 99999 8
```

Problem No. 31: Logic

Consider a 10*10 grid. Cells in this grid can contain one of five logic operations (AND, OR, NOT, Input, Output). These can be joined together to form a logic circuit. Given a description of a circuit and a set of boolean values, build the logic circuit and execute the input stream against it.

Input

The first line of the input contains a single integer *n*, which specifies the number of circuits to be processed. There will then be *n* groups of circuit descriptions and test values.

A circuit is made up of a number of operations. Each line describing an operation begins with three characters: the co-ordinates for a cell, 0-9 on the *X*-axis then 0-9 on the *Y*-axis, followed by a single character to represent the operation of that cell (`&' for AND, `|' for OR, `!' for NOT, `i' for Input and `o' for Output). Optionally following each triple is a set of co-ordinate pairs which represent the *x* and *y* co-ordinates of cells that take the output of this cells operation as an input for theirs. This (possibly empty) output list is terminated by `..'. The list of operations is terminated by a line containing the word `end'.

Next, for each circuit, comes the set of test values. The first line contains an integer *t* which gives the number of test cases your program must run. Next, there are *t* lines, each line containing a sequence of `0' and `1' characters symbolising the input values for one test case. The number of inputs will always correspond to the number of inputs defined by the circuit description. The input values are to be applied to the inputs in the order in which the input operations were defined in the circuit description.

The next circuit description, if any, will then follow.

Output

For each circuit, your program should output one line for each test case given in the input. The line should contain one `0' or `1' character for each output defined by the circuit description in the order in which the outputs were defined.

Your program should output a blank line after each set of test cases.

Sample Input

```
1
00i 11 13 ..
```

```
02i 11 13 ..
11& 21 ..
21o ..
13| 23 ..
23o ..
end
4
00
01
10
11
```

Sample Output

```
00
01
01
11
```

Notes:

i, o and ! operations will always have exactly one input.

& and | operations will always have exactly two inputs.

Even if an operation can feed others, it does not have to.

No recursive circuits.

Hint: Sample input specifies a circuit consisting of an `AND' and an `OR' operation in parallel both fed from the same two inputs:

```
               +---------\
3              |          |OR #----------OUT(2)
               |     +---/
               |     |
2     IN(2)----------+
               |     |
               +---------\
1                    |    |AND#----------OUT(1)
                     +---/
                     |
0      IN(1)---------+

         0                 1              2
```

In grid terms this is two inputs at 0,0 and 1,0. The first input feeds the AND operation at 1,1 and the OR operation at 1,3. The second input operation feeds the second input for the same AND and OR operations. The AND operation then feeds an output operation at 2,1. The OR operation also feeds an output operation, this one at 2,4.

Problem No. 32: Recognising Good ISBNs

Most books now published are assigned a code which uniquely identifies the book. The International Standard Book Number, or ISBN, is normally a sequence of 10 decimal digits, but in some cases, the capital letter X may also appear as the tenth digit. Hyphens are included at various places in the ISBN to make them easier to read, but have no other significance. The sample input and expected output shown below illustrate many valid, and a few invalid, forms for ISBNs.

Actually, only the first nine digits in an ISBN are used to identify a book. The tenth character serves as a check digit to verify that the preceding 9 digits are correctly formed. This check digit is selected so that the value computed as shown in the following algorithm is evenly divisible by 11. Since the check digit may sometimes need to be as large as 10 to guarantee divisibility by 11, a special symbol was selected by the ISBN designers to represent 10, and that is the role played by X.

The algorithm used to check an ISBN is relatively simple. Two sums, *s*1 and *s*2, are computed over the digits of the ISBN, with *s*2 being the sum of the partial sums in *s*1 after each digit of the ISBN is added to it. The ISBN is correct if the final value of *s*2 is evenly divisible by 11.

An example will clarify the procedure. Consider the (correct) ISBN 0-13-162959-X (for Tanenbaum's Computer Networks). First look at the calculation of *s*1:

```
----------------------------------------------------------------
digits in the ISBN    0   1   3   1   6   2   9   5   9   10(X)
partial sums        0   1   4   5  11  13  22  27  36   46
----------------------------------------------------------------
```

The calculation of *s*2 is done by computing the total of the partial sums in the calculation of *s*1:

```
----------------------------------------------------------------
s2 (running totals)   0   1   5  10  21  34  56  83  119  165
----------------------------------------------------------------
```

We now verify the correctness of the ISBN by noting that 165 is, indeed, evenly divisible by 11.

Input

The input data for this problem will contain one candidate ISBN per line of input, perhaps preceded and/or followed by additional spaces. No line will contain more than 80 characters, but the candidate ISBN may contain illegal characters, and more or fewer than the required 10 digits. Valid ISBNs may include hyphens at arbitrary locations. The end of file marks the end of the input data.

Output

The output should include a display of the candidate ISBN and a statement of whether it is legal or illegal. The expected output shown below illustrates the expected form.

Sample Input

```
0-89237-010-6
0-8306-3637-4
  0-06-017758-6
    This_is_garbage
1-56884-030-6
    0-8230-2571-3
    0-345-31386-0
    0-671-88858-7
    0-8104-5687-7
    0-671-74119-5
```

```
   0-812-52030-0
   0-345-24865-1-150
0-452-26740-4
   0-13-139072-4
   0-1315-2447-X
```

Sample Output

```
0-89237-010-6 is correct.
0-8306-3637-4 is correct.
0-06-017758-6 is correct.
This_is_garbage is incorrect.
1-56884-030-6 is correct.
0-8230-2571-3 is correct.
0-345-31386-0 is correct.
0-671-88858-7 is correct.
0-8104-5687-7 is correct.
0-671-74119-5 is correct.
0-812-52030-0 is correct.
0-345-24865-1-150 is incorrect.
0-452-26740-4 is correct.
0-13-139072-4 is correct.
0-1315-2447-X is correct.
```

Problem No. 33: What Base is This?

In positional notation we know the position of a digit indicates the weight of that digit toward the value of a number. For example, in the base 10 number 362 we know that 2 has the weight 10^0, 6 has the weight 10^1, and 3 has the weight 10^2, yielding the value $3*10^2+6*10^1+2*10^0$, or just 300 + 60 + 2. The same mechanism is used for numbers expressed in other bases. While most people assume the numbers they encounter everyday are expressed using base 10, we know that other bases are possible. In particular, the number 362 in base 9 or base 14 represents a totally different value than 362 in base 10.

For this problem your program will presented with a sequence of pairs of integers. Let's call the members of a pair *X* and *Y*. What your program is to do is determine the smallest base for *X* and the smallest base for *Y* (likely different from that for *X*) so that *X* and *Y* represent the same value.

Consider, for example, the integers 12 and 5. Certainly these are not equal if base 10 is used for each. But suppose 12 was a base 3 number and 5 was a base 6 number? 12 base 3 = $1*3^1+2*3^0$, or 5 base 10, and certainly 5 in any base is equal to 5 base 10. So 12 and 5 *can* be equal, if you select the right bases for each of them!

Input

On each line of the input data there will be a pair of integers, *X* and *Y*, separated by one or more blanks; leading and trailing blanks may also appear on each line, are to be ignored. The bases associated with *X* and *Y* will be between 1 and 36 (inclusive), and as noted above, need not be the same for *X* and *Y*. In representing these numbers the digits 0 through 9 have their usual decimal interpretations. The uppercase alphabetic characters A through Z represent digits with values 10 through 35, respectively.

Output

For each pair of integers in the input display a message similar to those shown in the examples shown below. Of course if the two integers cannot be equal regardless of the assumed base for each, then print an appropriate message; a suitable illustration is given in the examples.

Sample Input

```
12   5
   10     A
12 34
  123   456
  1    2
  10   2
```

Sample Output

```
12 (base 3) = 5 (base 6)
10 (base 10) = A (base 11)
12 (base 17) = 34 (base 5)
123 is not equal to 456 in any base 2..36
1 is not equal to 2 in any base 2..36
10 (base 2) = 2 (base 3)
```

Problem No. 34: Don't Have A Cow, Dude

Old MacDonald has a farm and on this farm he has a cow - and a fenced circular pasture with a radius of 100 yards. He plans to tie the cow to a post on the circumference of the pasture. He wants the cow to be able to eat one third of the grass in the pasture. How long should the rope be?

You must solve a generalization of this problem.

Input and Output

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. Write a program which will input two numbers, *R* (*raduis* - an integer between 1 and 1000 inclusive) and *P* (*part of* - a real number between 0.0 and 0.5 inclusive) and solve old MacDonald's problem. How long a rope should old MacDonald use to allow the cow to eat *P* of the grass in the circular pasture of radius *R*. Express your answer correct to two decimal places. Use π=3.14159.

For each test case, output a statement in the format shown in the sample output below. The outputs of two consecutive cases will be separated by a blank line.

Sample input

```
1

100 0.33
```

Sample output

R = 100, P = 0.33, Rope = 13.24

Note

The value 13.24 in the sample output is purposely not correct. It is included only to show you the correct format

---

Problem No. 35: Ackermann Functions

An Ackermann function has the characteristic that the length of the sequence of numbers generated by the function cannot be computed directly from the input value. One particular integer Ackermann function is the following:

$$X_{n+1} = \begin{cases} \dfrac{X_n}{2} & \textit{if } X_n \textit{ is even} \\ 3X_n + 1 & \textit{if } X_n \textit{ is odd} \end{cases}$$

This Ackermann has the characteristic that it eventually converges on 1. A few examples follow in which the starting value is shown in square brackets followed by the sequence of values that are generated, followed by the length of the sequence in curly braces:

[10] 5 16 8 4 2 1 {6}
[13] 40 20 10 5 16 8 4 2 1 {9}
[14] 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 {17}
[19] 58 29 88 44 22 ... 2 1 {20}
[32] 16 8 4 2 1 {5}
[1] 4 2 1 {3}

Input and Output

Your program is to read in a series of pairs of values that represent the first and last numbers in a closed sequence. For each closed sequence pair determine which value generates the longest series of values before it converges to 1. The largest value in the sequence will not be larger than can be accommodated in a 32-bit Pascal LongInt or C long. The last pair of values will be 0, 0. The output from your program should be as follows:

Between *L* and *H*, *V* generates the longest sequence of *S* values.

Where:

L = the lower boundary value in the sequence

H = the upper boundary value in the sequence

V = the first value that generates the longest sequence, (if two or more values generate the longest sequence then only show the lower value) *S* = the length of the generated sequence.

In the event that two numbers in the interval should both produce equally long sequences, report the first.

Sample Input

```
 1 20
35 55
 0 0
```

Sample Output

```
Between 1 and 20, 18 generates the longest sequence of 20 values.
Between 35 and 55, 54 generates the longest sequence of 112 values.
```

Problem No. 36: Making the Grade

Mr. Chips has a simple grading scheme that lends itself to automated computation. You will write a program that will read in his students' grades, bonus points, and attendance record, compute the student's grades, and output the average grade point of the class.

Mr. Chips grades as follows. All tests are based on 100 points and all test grades are between 0 and 100 points. If he has given more than 2 tests then he will drop the lowest test grade for each student before computing student averages. After computing student averages he computes the overall class average (*mean*) and standard deviation (*sd*). The cutoff points for grades are: an average ≥one *sd* above the *mean* is an A, an average ≥the *mean* but < one *sd* above the *mean* is a B, an average ≥one *sd* below the *mean* but < the *mean* is a C, and an average < one *sd* below the *mean* is a D.

For every two bonus points accrued by a student Mr. Chips increases their computed average by 3 percentage points. Thus, if students have one bonus point, their averages are not bumped at all. If they have 4 or 5 bonus points, their averages are bumped by 6 percentage points, and so on. Bumping of averages based on bonus points takes place after the grade cut-off points have been determined.

Finally, for every 4 absences, students lose one letter grade (from A to B, B to C, C to D, and D to F). For example, if they have 9 absences they will lose two letter grades. Students cannot get a grade lower than F. If students have perfect attendance, they gain one letter grade; although they cannot get a grade higher than an A. During his computations, Mr. Chips always rounds his results to the nearest tenth. In summary, Mr. Chips drops a student's lowest test grade if more than 2 tests have been administered, computes each student's average, computes the class *mean* and *sd*, adjusts the students' averages based on bonus points, determines the student's unadjusted grades, and then adjusts the grades based on attendance.

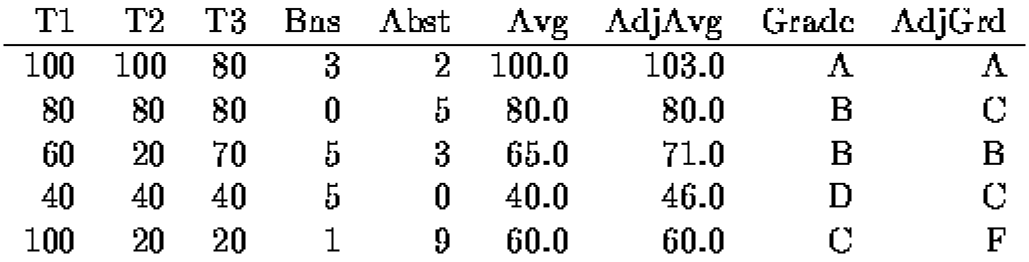
The average grade point (*avg grd pnt*) of a class is determined by using 4 points for each A, 3 points for each B, 2 points for each C, 1 point for each D, and 0 points for each F. The total points for the class are added together and divided by the number of students in the class (which is always at least 2).

The standard deviation *sd* of a list of numbers $x_1, \ldots, x_n$ is:

$$sd = \sqrt{\frac{\sum(x_i - mean)^2}{n}}$$

If the calculated standard deviation is less than 1 then Mr. Chips uses 1 in place of the standard deviation for grade calculation.

Suppose Mr. Chips has 5 students and has given 3 tests. The following table shows the grades, number of bonuses and days absent, plus the computed average (with lowest test dropped), the adjusted average (with bonus), the unadjusted grade and the adjusted grade (with attendance). The *mean* and *sd* used to determine letter grade cutoffs are 69.0 and 20.1. For example, for an unadjusted B, one's average must be greater than or equal to 69.0 and less than 89.1.The *avg grd pnt* is 2.2



T1	T2	T3	Bns	Abst	Avg	AdjAvg	Grade	AdjGrd
100	100	80	3	2	100.0	103.0	A	A
80	80	80	0	5	80.0	80.0	B	C
60	20	70	5	3	65.0	71.0	B	B
40	40	40	5	0	40.0	46.0	D	C
100	20	20	1	9	60.0	60.0	C	F

Input

The first line contains an integer *N* between 1 and 10 describing how many of Mr. Chip's classes are represented in the input. The first line for each class contains two integers *S* and *T*. *S* is the number of students in the class (1 < *S* < 31) and *T* is the number of tests the students took (1 < *T* < 11). The next *S* lines will each represent one student in the class. A student line first lists each of their *T* test scores as integers between 0 and 100 inclusive, and then lists their bonus points and their number of absences.

Output

There should be *N*+2 lines of output. The first line of output should read MAKING THE GRADE OUTPUT. There will then be one line of output for each of Mr. Chip's classes showing that class's average grade point. The final line of output should read END OF OUTPUT.

Sample Input

```
3
3 2
100 50 2 5
60 60 17 1
20 10 0 0
5 5
100  80  90  80  90   0   0
80 80 80 80 80   0 0
50 50 50 50 50 0 0
100 100 20 20 20 0 0
30 30 30 30 30 0 0
10  4
79   56   59   89    4    5
100   89   96   79    6    2
80   80   80   76    1    3
76   76   76   76    5    4
58   78   67   75    4    0
100   96  100   95    1    2
47   49   46   45    0    5
67   98   59   87    5    5
23   45   52   54    2    7
78   75   79   79    3    3
```

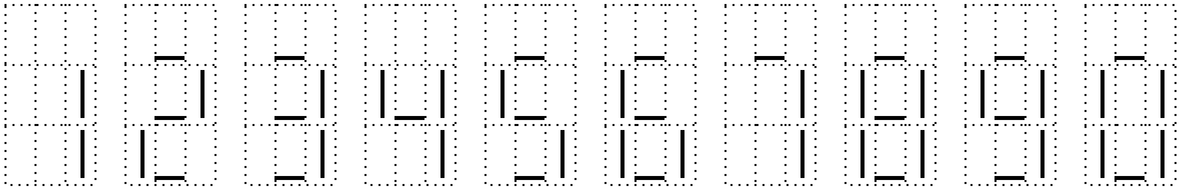
Sample Output

```
MAKING THE GRADE OUTPUT
3.0
3.2
2.4
```

END OF OUTPUT

Problem No. 37: Bank (Not Quite O.C.R.)

Banks, always trying to increase their profit, asked their computer experts to come up with a system that can read bank cheques; this would make the processing of cheques cheaper. One of their ideas was to use optical character recognition (OCR) to recognize bank accounts printed using 7 line-segments.
Once a cheque has been scanned, some image processing software would convert the horizontal and vertical bars to ASCII bars `|' and underscores `_'.
The ASCII 7-segment versions of the ten digits look like this:



A bank account has a 9-digit account number with a checksum. For a valid account number, the following equation holds: *$(d_1+2 * d_2 + 3 * d_3 + ... + 9 * d_9) \bmod 11 = 0$.*
Digits are numbered from right to left like this: $d_9d_8d_7d_6d_5d_4d_3d_2d_1$.
Unfortunately, the scanner sometimes makes mistakes: some line-segments may be missing. Your task is to write a program that deduces the original number, assuming that:
when the input represents a valid account number, it is the original number;
at most one digit is garbled;
the scanned image contains no extra segments.
For example, the following input

```
    _  _     _  _  _  _  _ 
  | _| _||_||_ |_   ||_||_|
  | _  _|  | _||_|  ||_| _|
```

used to be "123456789".

Input Specification

The input file starts with a line with one integer specifying the number of account numbers that have to be processed. Each account number occupies 3 lines of 27 characters.

Output Specification

For each test case, the output contains one line with 9 digits if the correct account number can be determined, the string ``failure" if no solutions were found and ``ambiguous" if more than one solution was found.

Sample Input

```
4

    _  _     _  _  _  _  _ 
  | _| _||_||_ |_   ||_||_|
  | _  _|  | _||_|  ||_| _|

    _  _  _  _  _  _     _ 
|_||_||  ||  ||_   |  |  ||_ 
  | _||_||_||_|  |  |  | _|
 _  _  _  _  _  _  _  _  _ 
|_||_||_||_||_||_||_||_||_|
|_||_||_||_||_||_||_||_||_|

 _     _  _  _  _  _  _  _ 
|_|  ||_||_||_||_||_||_||_|
|_|  ||_||_||_||_||_||_||_|
```

Sample Output

```
123456789
ambiguous
failure
878888888
```

Problem No. 38: Moscow Time

In e-mail the following format for date and time setting is used:
EDATE::=Day_of_week, Day_of_month Month Year Time Time_zone

Here *EDATE* is the name of date and time format, the text to the right from ``::=" defines how date and time are written in this format. Below the descriptions of *EDATE* fields are presented:

Day-of-week

The name of a day of the week. Possible values: MON, TUE, WED, THU, FRI, SAT, SUN. The name is followed by ``," character (a comma).

Day-of-month

A day of the month. Set by two decimal digits.

Month

The name of the month. Possible values: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC.

Year

Set by two or four decimal digits. If a year is set by two decimals it is assumed that this is a number of the year of the XX century. For instance, 74 and 1974 set a year of 1974.

Time

Local time in format *hours:minutes:seconds*, where hours, minutes and seconds are made up of two decimal digits. The time keeps within the limits from 00:00:00 to 23:59:59.

Time-zone

Offset of local time from Greenwich mean time. It is set by the difference sign ``+" or ``-"and by sequence of four digits. First two digits set the hours and the last two the minutes of offset value. The absolute value of the difference does not exceed 24 hours. Time zone can also be presented by one of the following names:

Name	Digital value
UT	-0000
GMT	-0000
EDT	-0400

CDT	-0500
MDT	-0600
PDT	-0700

Each two adjacent fields of *EDATA* are separated with exactly one space. Names of day of the week, month and time zone are written in capitals. For instance, 10 a.m. of the Contest day in St.Petersburg can be presented as

TUE, 03 DEC 96 10:00:00 +0300

Write a program which transforms the given date and time in *EDATE* format to the corresponding date and time in Moscow time zone. So called ``summer time" is not taken into consideration. Your program should rely on the predefined correctness of the given *Day_of_week* and *Time_zone*.

A note

Moscow time is 3 hours later than Greenwich mean time (time zone +0300)

Months: January, March, May, July, August, October and December have 31 days. Months: April, June, September and November have 30 days. February, as a rule, has 28 days, save for the case of the leap year (29 days).

A year is a leap year if valid one out of two following conditions:
its number is divisible by 4 and is not divisible by 100;
its number is divisible by 400.
For instance, 1996 and 2000 are the leap years, while 1900 and 1997 are not.

Input

Input data file contains date and time in *EDATE* format in each line. Minimum permissible year in the input data is 0001, maximum 9998. Input *EDATA* string does not contain leading and trailing spaces.

Output

Output must contain a single line for each one in the input file with date and time of Moscow time zone in *EDATE* format. In output *EDATE* string a Year must be presented with four decimal digits. The output string should not include leading and trailing spaces.

Sample Input

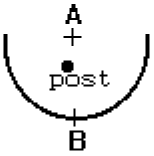
```
SUN, 03 DEC 1996 09:10:35 GMT
WED, 28 FEB 35 23:59:00 +0259
```

Sample Output

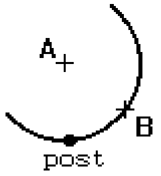
```
SUN, 03 DEC 1996 12:10:35 +0300
THU, 01 MAR 1935 00:00:00 +0300
```

Problem No. 39: Horse Shoe Scoring

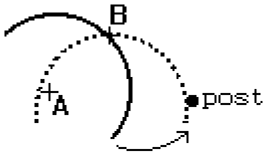
The game of horseshoes is played by tossing horseshoes at a post that is driven into the ground. Four tosses generally make up a game. The scoring of a toss depends on where the horseshoe lands with respect to the post. If the center of the post is within the region bounded by the interior of the horseshoe and the imaginary line connecting the two legs of the horseshoe, and the post is not touching the horseshoe, it is a ``ringer" and worth five points. If any part of the horseshoe is touching the post, it is a ``toucher" and worth 2 points. If the toss is neither a ringer nor a toucher and some part of the horseshoe will touch the post when it is pivoted around its point B, it is a ``swinger'' and worth 1 point. Any horseshoe which does not fit any of the scoring definitions scores zero points. See the figures below for examples of each of the scoring possibilities.



Ringer



Toucher



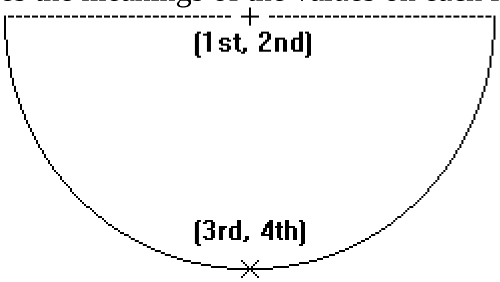
Swinger

This program uses mathematical horseshoes that are semicircles with radius 10 centimetres. The location of the horseshoe is the given by two points: the centre point of the semicircle, measured in centimetres relative to x and y axes, and the point that exactly bisects the semicircle. The post is at location (0,0) and is 2 centimetres in diameter. The top of the post is level with the ground allowing the horseshoe to lay on top of the post; therefore, a ``toucher'' would mean that any part of the horseshoe lies within the circle with a radius of 1 centimetre centred at (0, 0).

Each ``turn'' consists of four tosses. The purpose of your program is to determine the score of the ``turn'' by computing the sum of the point values for each of the four tosses.

Input

Input to your program is a series of turns, and a turn consists of four horseshoe positions. Each line of input consists of two coordinate pairs representing the position of a toss. Each coordinate consists of a floating point (X,Y) coordinate pair $(-100.0 \le X,Y \le 100.0)$ with up to 3 digits of precision following the decimal point; the first and second numbers are the X and Y coordinates of the centre point of the horseshoe semicircle (Point A) and the third and fourth numbers are the X and Y coordinates of the point (B) which bisects the horseshoe semicircle. All coordinates are in centimetres. You can be assured that the distance between points A and B for each horseshoe will be 10 +- 0.000001 centimetres. The figure below illustrates the meanings of the values on each line.



The first four lines of input define the horseshoe positions for the first turn; lines 5 through 8 define the second turn, etc. There are at most 9999 turns in the input file, and every turn contains four horseshoe positions. Your program should continue reading input to the end of file.

Output

The first line of output for your program should be the string ``Turn Score'' in columns 1 through 10. For each ``turn'', your program should print the number of the turn right-justified in columns 1-4 (turns are numbered starting with 1), a single space (ASCII character 32 decimal) in columns 5 through 8 , and the score for the turn right-justified in columns 9 and 10 with a single leading blank for scores 0 to 9. Numbers that are right-justified should be preceded by blanks, not zeroes, as the fill character.

Sample Input

```
76.5 53.3 76.5 43.3
-5.1 1.0 4.9 1.0
5.1 0.7 5.1 -9.3
7.3 14.61 7.3 4.61
23.1 17.311 23.1 27.311
-23.1 17.311 -23.1 27.311
-23.1 -17.311 -23.1 -27.311
23.1 -17.311 23.1 -27.311
76.5 53.3 76.5 43.3
76.5 53.3 76.5 43.3
-1.0 -2.0 9.0 -2.0
1.0 -2.0 9.0 4.0
```

Sample Output

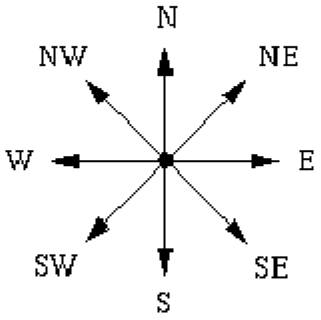
```
Turn Score
   1    11
   2     0
   3    10
```

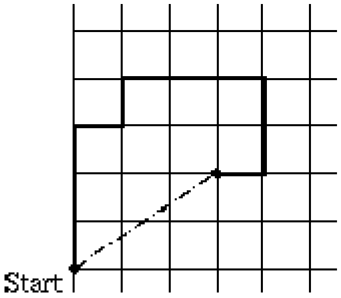
Problem No. 40: There's treasure everywhere!

Finding buried treasures is simple: all you need is a map! The pirates in the Caribbean were famous for their enormous buried treasures and their elaborate maps. The maps usually read like ``Start at the lone palm tree. Take three steps towards the forest, then seventeen step towards the small spring, . . . blahblah . . . , finally six steps toward the giant rock. Dig right here, and you will find my treasure!" Most of these directions just boil down to taking the mentioned number of steps in one of the eight principal compass directions (depicted in the left of the figure).

Obviously, following the paths given by these maps may lead to an interesting tour of the local scenery, but if one is in a hurry, there is usually a much faster way: just march directly from your starting point to the place where the treasure is buried. Instead of taking three steps north, one step east, one step north, three steps east, two steps south and one step west (see figure), following the direct route (dashed line in figure) will result in a path of about 3.6 steps.

You are to write a program that computes the location of and distance to a buried treasure, given a `traditional' map.





Input

The input file contains several strings, each one on a line by itself, and each one consisting of at most 200 characters. The last string will be END, signaling the end of the input. All other strings describe one treasure map each, according to the following format:

The description is a comma-separated list of pairs of lengths (positive integers less than 1000) and directions (N (north), NE (northeast), E (east), SE (southeast), S (south), SW (southwest), W (west) or NW (northwest)). For example, 3W means 3 steps to the west, and 17NE means 17 steps to the northeast. A full stop (.) terminates the description, which contains no blanks.

Output

For every map description in the input, first print the number of the map, as shown in the sample output. Then print the absolute coordinates of the treasure, in the format ``The treasure is located at (*x*,*y*).''. The coordinate system is oriented such that the *x*-axis points east, and the *y*-axis points north. The path always starts at the origin (0,0).

On the next line print the distance to that position from the point (0,0), in the format ``The distance to the treasure is *d*.''. The fractional values *x*, *y*, *d* must be printed exact to three digits to the right of the decimal point.

Print a blank line after each test case.

Sample Input

```
3N,1E,1N,3E,2S,1W.
10NW.
END
```

Sample Output

```
Map #1
The treasure is located at (3.000,2.000).
The distance to the treasure is 3.606.

Map #2
The treasure is located at (-7.071,7.071).
The distance to the treasure is 10.000.
```

Problem No. 41: Scheduling Lectures

You are teaching a course and must cover n ($1 \le n \le 1000$) topics. The length of each lecture is L ($1 \le L \le 500$) minutes. The topics require t_1, t_2, ..., t_n ($1 \le t_i \le L$) minutes each. For each topic, you must decide in which lecture it should be covered. There are two scheduling restrictions:

1.

Each topic must be covered in a single lecture. It cannot be divided into two lectures. This reduces discontinuity between lectures.

2.

Topic i must be covered before topic i + 1 for all $1 \leq i < n$. Otherwise, students may not have the prerequisites to understand topic i + 1.

With the above restrictions, it is sometimes necessary to have free time at the end of a lecture. If the amount of free time is at most 10 minutes, the students will be happy to leave early. However, if the amount of free time is more, they would feel that their tuition fees are wasted. Therefore, we will model the dissatisfaction index (DI) of a lecture by the formula:

$$DI = \begin{cases} 0 & \text{if } t = 0 \\ -C & \text{if } 1 \leq t \leq 10 \\ (t - 10)^2 & \text{otherwise} \end{cases}$$

where C is a positive integer, and t is the amount of free time at the end of a lecture. The total dissatisfaction index is the sum of the DI for each lecture.

For this problem, you must find the minimum number of lectures that is needed to satisfy the above constraints. If there are multiple lecture schedules with the minimum number of lectures, also minimize the total dissatisfaction index.

Input

The input consists of a number of cases. The first line of each case contains the integer n, or 0 if there are no more cases. The next line contains the integers L and C. These are followed by n integers $t_1, t_2, \ldots, t_n$.

Output

For each case, print the case number, the minimum number of lectures used, and the total dissatisfaction index for the corresponding lecture schedule on three separate lines. Output a blank line between cases.

Sample Input

```
6
30 15
10
10
10
10
10
10
120 10
80
80
10
50
```

```
30
20
40
30
120
100
0
```

Sample Output

```
Case 1:
Minimum number of lectures: 2
Total dissatisfaction index: 0

Case 2:
Minimum number of lectures: 6
Total dissatisfaction index: 2700
```

Group B
Additional Problems

1. Write a C-program to investigate one of the possible placements of eight chess queens on a normal chessboard. The program should ask the user to enter the placement of the first queen and then continue from there for the other seven ones. None of the eight queens must have a kill on any of the other ones. Note that a kill situation is present between two queens if they are either on the same horizontal, vertical or diagonal lines. Different possibilities will arise and the program is only required to print the first possibility it will get to.

2. It is required to simulate the operation of the electric oven. The set temperature of the oven is first entered through the keyboard. The entrance of the current oven temperature (ambient temperature if the oven was not working for a while) then follows. At this instant the computer should first make sure that the oven door is closed. When it is so, the temperature will start to rise (use an incremental scaling factor of 0.001 for the temperature increment loop). The oven temperature needs to be displayed on the screen while it is incrementing, in addition to an indication of the "OVEN ON". A 10% range over the temperature in both directions is to be implemented so that when the higher temperature limit is reached, the electrical supply is disconnected. The screen should display "OVEN OFF". The oven then starts to cool down with a decremental scaling factor of –0.0001 for the temperature decrement loop. The temperature decrease is also displayed on the screen. For each new temperature. When the temperature reaches the lower temperature limit set by the tolerance, the electric heater of the oven will then restart. In this case, the temperature starts to increase again until it reaches the upper limit and then the whole cycle is repeated continuously.

   Note that if the temperature setting is 230°C

   The higher limit is 230+0.1*230 = 253°C

   The lower limit is 230-0.1*230 = 207°C

3. A water tank filling system is to be controlled using a computer. The simulation of this process is required to be written in C. The system consists of a large tank and a water pump. The tank is fitted with two sensors inside it at different heights from the bottom of the tank. These sensors are digital level indicators signalling a High logic (1) when the water level is higher than their position. A logic Low (0) is transmitted from the sensors when the water level is below their position. The two levels are referred to as L1 (High Water Level Indicator), L2 (Low Water Level Indicator). The height settings of these two sensors are entered from the keyboard (as a percentage of the highest possible water level, e.g., 10% and 90%) at the beginning of the program run. In addition, the initial water level is also entered from the keyboard during the initialisation process (e.g., 53%). Moreover, the initialisation of the process will also introduce the scaling factor for the water filling using the motor or the water consumption from the tank. Note that the water consumption rate (e.g., 0.001) has to be smaller than the filling rate (e.g., 0.01) multiplying factors. The pump motor will only operate if the water level reaches the minimum level. The screen should read the current water level of the tank, in addition to the status of the motor (whether it is "Motor ON" or "Motor OFF"). This message should appear during each simulation step. The motor will remain in the ON position until the water level goes just beyond the high water level of L1 of the tank filling capabilities. At this point, the motor will stop and will remain in the OFF position until the water being consumed all the time at the same entered rate goes below the minimum level indicated by L2. At this point, the motor will be turned ON and will start filling the tank as before until the water level reaches L1. Note that during the period when the motor is ON, a beep from the computer is to be generated at a reduced rate from the computations rate.

4. The simulation of the simultaneous operation of two lifts is required to be performed in C. The two lifts will be in two different levels, which will be entered by the user (**P1** and **P2**). The building contains 10 floors. The lower floor is number 0 and the upper floor is number 9. A position switch generates a signal represents the elevator's position. The C-program reads the position as a variable **P1** for the first lift and **P2** for the second lift. The program also reads-in the request coming from the passengers

waiting for the lift at a certain level. The program stores the request as the variable **R**. The program determines the new direction of each elevator. The direction is either up or down. The up direction is activated through variables **UP1** and **UP2** and the down direction is activated through variables **DOWN1** and **DOWN2**. The two variable-pairs (**UP1** and **DOWN1**) and (**UP2** and **DOWN2**) can be zeros at the same time but can never be one at the same time (Think about the logic: each elevator cannot move up and down in the same time). If the lift request is generated, the lift nearer to it will go to the level, provided only that it is in its same direction of motion (up or down).

Think of this problem as in the following example when the two lifts (1 and 2) are in levels 4 and 7 respectively at the same time, and both lifts are going down. If the request comes from level 5, it is not logic to make the lift going down at level 4 stop and go up to take the passengers but it is the other one which is in level 7 going down that will stop in level 5 and take the passengers. As mentioned earlier, the program will initialise levels **P1** and **P2** of the two lifts and their destinations **D1** and **D2** as decided by the passengers in the lift. If a request **R** is entered, then the program must take the decision, operate the two lifts and stop the appropriate lift at the Requested level. The new passengers do not have a choice on a new level until the destination (**D1** or **D2**) is reached. The program will then stop and ask the user to enter a new request destination (**P1** or **P2**).

5. A power transformer was tested to identify its parameters. The following test results were obtained:

   No Load Lest: V_{nl} = 220V, I_{nl} =0.8A and P_{nl} = 100W

   Short Circuit Test: V_{sc} = 22V, I_{sc} =4.5A and P_{sc} = 100W

   It is required to write a C program which enters the above test results from the keyboard of a transformer and then enter the VA rating of the transformer, its primary rated voltage and its secondary rated voltage. The program will then calculate the parameters of the approximate equivalent circuit referred to the primary once and referred to the secondary another time. If the transformer parameters are not valid a screen message should be generated to tell the user. The next step is to calculate the percentage voltage drop at different loading conditions from zero to 100% of the rated transformer loading for R-load, L-Load, C-load. These three curves are to be displayed on the screen sideways and all appropriate curve scaling is to be performed properly. This means that the y-axis of the curve is placed horizontally across the screen and the x-axis is placed vertically on the left hand side of the screen. Note that the screen has 80 columns by 25 rows.

6. A grid is formed up of N by M points. The user is at the upper left corner of the grid and wants to traverse the grid to the opposite corner point. The only problem is that there is a different sum of money placed at each point from the grid. The user is requested to traverse the grid in a way so as not to turn back (if the user is at a certain node, he/she can only move right, down or diagonally down to the right). The main aim of the user is to gather as much money as possible from the path to the opposing corner. It is required to write a C program, which performs this task. The program should read the dimensions N and M as well as the weights of each node. Then the program prints the path for maximum gain (enumerating the nodes with numbers in the horizontal direction and letters in the vertical direction: each node has a number and a letter to designate its position, such as C5).

7. A grid is formed up of M by N points. The user is at the top left corner of the grid and wants to traverse the grid to a pre-specified point on the grid. The only problem is that the user has to pay road tax at each node he reaches. The user is requested to traverse the grid in a way so as not to turn back (if the user is at a certain node, he/she can only move right, down or diagonally down to the right). The main aim of the user is to pay the minimum possible amount of money through the path to the opposing corner. It is required to write a C program, which performs this task. The program should read the dimensions M and N as well as the taxes at each node. Then the program prints the path (enumerating the nodes with numbers in the horizontal direction and letters in the vertical direction. Each node has a number and a letter to designate its position (such as D3).

8. An electrical solar/battery-powered vehicle is to be operated on a sunny day. Consider that the rate of energy consumption (in *Joules*) is defined for a certain journey by the following piece-wise linear equation dependant on the acceleration rate of the vehicle (in *m/sec*^{2}) speed (in *m/sec*) of the vehicle:

$$E(t) = \begin{cases} 10\,(a-5)^2 + 10\,v & for \quad 0 < a \le 20 \\ 10\,v & for \quad v = Const \\ -3\,a^2 + 10\,v & for \quad -20 \le a < 0 \end{cases}$$

The amount of energy gained by the solar panels placed on top and around the vehicle, is proportional to the amount of sunrays incident on their surface. If any cloud covers the sun, the amount of energy dips to half its original value. The clouds appear during the first 5 minutes of every 15 minutes.

If the trajectory of the vehicle consists of an acceleration period of 10*m/sec*^{2} constant acceleration to reach the maximum speed of 60 km/hr. The vehicle then remains at this speed for the longest period of the trajectory. The deceleration period returns the kinetic energy of the vehicle to the batteries. This is called regenerative breaking and lasts for only 5 seconds whatever the speed is.

Determine the maximum distance that the vehicle will run for if the initial energy stored in the battery and the energy stored by the sunrays are entered from the keyboard.

9. Write a C program that interactively plays the game of dots and crosses with a single user. The game consists of three rows and three columns where the symbols “X” and “O” are placed, corresponding to each of the two players taking turns. The winner of the game is the one who can manage to place either 3 “Xs” or 3 “Os” in a diagonal. If no possibility is there to reach a diagonal for either players, the result is a draw between the two players. The winner of the game gets 3 points, each party of the draws gets 1 point only and the loser does not get any. The game is repeated a predefined number of times and the winner of the game is the one with maximum score. The computer will keep track of the moves of the player and itself and print the result of the game in 3 consecutive rows for each iteration of the game.

10. Write a C program that prompts the user to enter the number of inputs of a certain digital circuit. The program will generate all the input combinations and the user has to enter the logic value of the outputs for each of the input combinations (the program assumes one output only). The truth table is then displayed on the screen and once the user accepts the values displayed; the Karnaugh map is drawn on the screen. The program then simplifies the map and generates the output equation which should be written on the screen assuming that the variables of the map are A_0 to A_{n-1}, where n is the number of inputs, A_0 being the least significant bit of the variables and A_{n-1} is the most significant bit.

11. It is required to write a C program to simulate the game of "Hang Man". Ten words are predefined inside the program (hard coded). The user is prompted to enter a number from 1 to 10. The corresponding word is then used as the candidate for the game. The same number of characters in this word is displayed on the screen as dots. For instance for the word dog, the following will be displayed "…". The user is then prompted to guess a letter. If the letter suggested by the user is in the word, the letter appears on the screen (in all its positions in the word). If the user enters a wrong letter, the same word is reprinted on the screen (showing only the guessed letters). The list of wrong letters appears in the bottom in a list. When this number reaches 10 letters, the user has lost his guess. He/she might be prompted to play again or not.

12. For a function $y=f(x)$, finding a root for the function means finding a value x that yields $y=0$. For many functions (the continuous functions), the following is true: if $f(a)$ and $f(b)$ are of different signs, then there must be at least one root between a and b. One way to approximate that root is by the bisection method, described as follows: let m be the midpoint of the interval from a to b. If $f(m)=0$, then you have found a root, so quit. If $f(a)$ and $f(m)$ have opposite signs, there must be a root between a and m; if not, there must be a root between m and b. In either case you can repeat this process with one or the other sub-interval. Since the sub-intervals will get smaller as you proceed, you will be getting closer and closer to the root. Quit after a predetermined number of passes.

Write a program which uses this method to approximate the root of $y=x^3+4.5x^2-0.19x-0.14$ that lies between 0 and 1. Stop after 30 iterations and use m as the approximation, unless a root is found before 30 iterations have occurred.

13. Write a program that prompts the user to enter a text (a certain sentence). The user is prompted to either choose encryption or decryption. The program performs one of the following operations:

    a. XOR each character with a specified number from the user.

    b. Invert each character bit by bit.

    These operations are presented to the user in a choice menu being printed on the screen. The user can select only one of them to be applied to the specified string and then choose whether he/she wants to perform the inverse operation again or not.