2003 Mid-Atlantic Regional Programming Contest

Welcome to the 2003 Programming Contest. Before you start the contest, please be aware of the following notes:

1. There are eight (8) problems in the packet, using letters A-H. These problems are NOT sorted by difficulty. As a team's solution is judged correct, the team will be awarded a balloon. The balloon colors are as follows:

Problem	Problem Name	Balloon Color
A	A Contesting Decision	Blue
B	Which Way Do I Go?	Orange
C	Choose Your Words Carefully	Silver
D	A Round Peg in a Ground Hole	Red
E	Helping Florida	Gold
F	Three Sides Make a Triangle	Purple
G	High and Dry	Pink
H	Reverse Roman Notation	Green

2. All solutions must read from standard input and write to standard output. In C this is scanf/printf, in C++ this is cin/cout, and in Java this is System.in/System.out. The judges will ignore all output sent to standard error. (You may wish to use standard error to output debugging information.) From your workstation you may test your program with an input file by redirecting input from a file:

```
program < file.in
```

3. Solutions for problems submitted for judging are called runs. Each run will be judged. Runs for each particular problem will be judged in the order they are received. However, it is possible that runs for different problems may be judged out of order. For example, you may submit a run for B followed by a run for C, but receive the response for C first. DO NOT request clarifications on when a response will be returned. If you have not received a response for a run within 60 minutes of submitting it, **you may have a runner ask the site judge to determine the cause of the delay. Under no circumstances should you ever submit a clarification request about a submission for which you have not received a judgment.**

   The judges will respond to your submission with one of the following responses. In the event that more than one response is applicable, the judges may respond with any of the applicable responses.

Response	Explanation
Correct	Your submission has been judged correct.
Incorrect Output	Your submission generated output that is not correct.
Output Format Error	Your submission's output is not in the correct format, is misspelled, or did not produce all of the required output.
Excessive Output	Your submission generated output in addition to or instead of what is required.
Compilation Error	Your submission failed to compile.
Run-Time Error	Your submission experienced a run-time error.
Time-Limit Exceeded	Your submission did not solve the judges' test data within 30 seconds.

4. A team's score is based on the number of problems they solve and penalty points, which reflect the amount of time and incorrect submissions made before the problem is solved. For each problem solved correctly, penalty points are charged equal to the time at which the problem was solved plus 20 minutes for each incorrect submission. No penalty points are added for problems that are never solved. Teams are ranked first by the number of problems solved and second by the fewest penalty points.

5. This problem set contains sample input and output for each problem. However, you may be assured that the judges will test your submission against several other more complex datasets, which will not be revealed until after the contest. Your major challenge is designing other input sets for yourself so that you may fully test your program before submitting your run. Should you receive an incorrect judgment, you are advised to consider what other datasets you could design to further evaluate your program.

6. In the event that you feel a problem statement is ambiguous, you may request a clarification. Read the problem carefully before requesting a clarification. If the judges believe that the problem statement is sufficiently clear, you will receive the response, "The problem statement is sufficient, no clarification is necessary." If you receive this response, you should read the problem description more carefully. If you still feel there is an ambiguity, you will have to be more specific or descriptive of the ambiguity you have found. If the problem statement is ambiguous in specifying the correct output for particular input, please include that input data in the clarification request.

   Additionally, you may submit a clarification request asking for the correct output for input you provide. The judges will seek to respond to these requests with the correct output. These clarification requests will be answered only when no clarifications regarding ambiguity are pending. The judges reserve the right to suspend responding to these requests during the contest.

   If a clarification, including output for a given input, is issued during the contest, it will be broadcast to all teams.

7. The submission of abusive programs or clarification requests to the judges will be considered grounds for immediate disqualification.

8. Good luck, and HAVE FUN!!!

Problem A: A Contesting Decision

Judging a programming contest is hard work, with demanding contestants, tedious decisions, and monotonous work. Not to mention the nutritional problems of spending 12 hours with only donuts, pizza, and soda for food. Still, it can be a lot of fun.

Software that automates the judging process is a great help, but the notorious unreliability of some contest software makes people wish that something better were available. You are part of a group trying to develop better, open source, contest management software, based on the principle of modular design.

Your component is to be used for calculating the scores of programming contest teams and determining a winner. You will be given the results from several teams and must determine the winner.

Scoring

There are two components to a team's score. The first is the number of problems solved. The second is penalty points, which reflects the amount of time and incorrect submissions made before the problem is solved. For each problem solved correctly, penalty points are charged equal to the time at which the problem was solved plus 20 minutes for each incorrect submission. No penalty points are added for problems that are never solved.

So if a team solved problem one on their second submission at twenty minutes, they are charged 40 penalty points. If they submit problem 2 three times, but do not solve it, they are charged no penalty points. If they submit problem 3 once and solve it at 120 minutes, they are charged 120 penalty points. Their total score is two problems solved with 160 penalty points.

The winner is the team that solves the most problems. If teams tie for solving the most problems, then the winner is the team with the fewest penalty points.

Input

For the programming contest your program is judging, there are four problems. You are guaranteed that the input will not result in a tie between teams after counting penalty points.

Line 1	`<nTeams>`
Line 2–$n+1$	`<Name> <p1Sub> <p1Time> <p2Sub> <p2Time> ... <p4Time>` The first element on the line is the team name, which contains no whitespace. Following that, for each of the four problems, is the number of times the team submitted a run for that problem and the time at which it was solved correctly (both integers). If a team did not solve a problem, the time will be zero. The number of submissions will be at least one if the problem was solved.

Output

The output consists of a single line listing the name of the team that won, the number of problems they solved, and their penalty points.

Example

Input:

```
4
Stars 2 20 5 0 4 190 3 220
Rockets 5 180 1 0 2 0 3 100
Penguins 1 15 3 120 1 300 4 0
Marsupials 9 0 3 100 2 220 3 80
```

Output:

```
Penguins 3 475
```

Problem B: Which Way Do I Go?

You've decided that the time is right for the next Internet map startup. Your inspiration for this venture is your directionally impaired spouse, who doesn't care for the shortest or quickest routes generated by current online map systems—instead, easier is better.

The backbone of your operation is, of course, your algorithm for calculating directions. Your algorithm must accept the map from your massive database of the entire country and produce the best route that meets the customer's query. Queries consist of an origin, destination, and the optimization goal, which can be for the shortest route, fastest route, or route with the fewest turns. You are guaranteed that there will be a path between source and destination for any query asked.

Input

There are three sections in the input file, the first lists the cities, the second lists the roads, and the third lists the queries.

The first line of input is the number of cities, c, followed by c lines each containing the name of a city. There is no whitespace in a city name.

The next line is the number of roads, r, followed by r lines describing a road. Each road description has the following form:

```
<RoadName> <CityA> <ABDistance> <ABTime> <CityB> [<BCDist> <BCTime> <CityC> [...]]
```

There is no whitespace in a road's name. Roads may pass through any number of cities. The cities appear in the order the road passes through them. No road passes through the same city multiple times. Roads are bidirectional. The distance and time (both real numbers) it takes to follow a road between each pair of cities is the distance and time listed between those two names. When following a road between multiple cities (A to C, for example), the distance and time is cumulative for all steps along the path.

The remainder of the lines in the file each list one query. Queries are of the form:

```
<querytype> <origin> <destination>
```

Querytype is one of `time`, `distance`, or `turns` and the origin and destination are the names of cities.

The queries end at end-of-file.

Output

For each query, your output will begin with a line:

```
from <origin>
```

Each following line will be of the form:

```
<roadName> to <cityname>
```

which lists the road to turn onto from the previous city, and the city to take that road to. If a single road is followed between multiple cities, only the final city reached on that road before turning onto another road is listed. The final city listed will be the destination city.

Example

Input:

```
5
Chicago
DesMoines
OklahomaCity
Dallas
LosAngeles
4
I80 Chicago 300 4 DesMoines
I35 DesMoines 550 7.3 OklahomaCity 205 3 Dallas
I40 OklahomaCity 1330 20.5 LosAngeles
Rt66 Chicago 2448 46 LosAngeles
time Chicago LosAngeles
turns Chicago LosAngeles
```

Output:

```
from Chicago
I80 to DesMoines
I35 to OklahomaCity
I40 to LosAngeles
from Chicago
Rt66 to LosAngeles
```

Problem C: Choose Your Words Carefully

Let’s face it, most students who do a good job on programming contest problems aren’t the best writers. However, there is no artistic process that a good programmer can’t improve with a little program to automate the process.

In this case, your writing problem is a tendency to use words too often. To help check for this, you are going to write a program that will search your papers for the word or words you use most often.

Input

The input consists entirely of your paper. Each line will be under 80 characters and will contain words, punctuation, and numbers. Words consist of the characters {a–z,A–Z,0–9}. Words are separated by whitespace, end-of-line, and punctuation. The punctuation that may be found includes the characters `,.;\’‘\"()/:-`. No other characters will be found in the input.

The input ends at end-of-file.

Word comparisons are case-insensitive.

Output

Your output begins with the line:

```
<n> occurrences
```

where n is the number of times the most frequently appearing word occurs.

Following that line will be the words that occurred n times, one per line. The words may be listed in any order.

Example

Input:

```
Fourscore and seven years ago our fathers brought forth on this
continent a new nation, conceived in liberty and dedicated to the
proposition that all men are created equal. Now we are engaged in
a great civil war, testing whether that nation or any nation so
conceived and so dedicated can long endure.
```

Output:

```
3 occurrences
nation
and
```

Problem D: A Round Peg in a Ground Hole

The DIY Furniture company specializes in assemble-it-yourself furniture kits. Typically, the pieces of wood are attached to one another using a wooden peg that fits into pre-cut holes in each piece to be attached. The pegs have a circular cross-section and so are intended to fit inside a round hole.

A recent factory run of computer desks were flawed when an automatic grinding machine was mis-programmed. The result is an irregularly shaped hole in one piece that, instead of the expected circular shape, is actually an irregular polygon. You need to figure out whether the desks need to be scrapped or if they can be salvaged by filling a part of the hole with a mixture of wood shavings and glue.

There are two concerns. First, if the hole contains any protrusions (i.e., if there exist any two interior points in the hole that, if connected by a line segment, that segment would cross one or more edges of the hole), then the filled-in-hole would not be structurally sound enough to support the peg under normal stress as the furniture is used. Second, assuming the hole is appropriately shaped, it must be big enough to allow insertion of the peg. Since the hole in this piece of wood must match up with a corresponding hole in other pieces, the precise location where the peg must fit is known.

Write a program to accept descriptions of pegs and polygonal holes and determine if the hole is ill-formed and, if not, whether the peg will fit at the desired location. Each hole is described as a polygon with vertices $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. The edges of the polygon are (x_i, y_i) to (x_{i+1}, y_{i+1}) for $i = 1 \ldots n-1$ and (x_n, y_n) to (x_1, y_1).

Input:

Input consists of a series of piece descriptions. Each piece description consists of the following data:

Line 1	`<nVertices> <pegRadius> <pegX> <pegY>`
	number of vertices in polygon, n (integer)
	radius of peg (real)
	X and Y position of peg (real)
n **Lines**	`<vertexX> <vertexY>`
	On a line for each vertex, listed in order, the X and Y position of vertex

The end of input is indicated by a number of polygon vertices less than 3.

Output:

For each piece description, print a single line containing the string:

`HOLE IS ILL-FORMED`	if the hole contains protrusions
`PEG WILL FIT`	if the hole contains no protrusions and the peg fits in the hole at the indicated position
`PEG WILL NOT FIT`	if the hole contains no protrusions but the peg will not fit in the hole at the indicated position

Example

Input:

```
5 1.5 1.5 2.0
1.0 1.0
2.0 2.0
1.75 2.0
1.0 3.0
0.0 2.0
5 1.5 1.5 2.0
1.0 1.0
2.0 2.0
1.75 2.5
1.0 3.0
0.0 2.0
1
```

Output:

```
HOLE IS ILL-FORMED
PEG WILL NOT FIT
```

Problem E: Helping Florida

While Florida's difficulties electing presidents are well known, a lesser known problem is electing committee chairs within the state's House of Delegates. The process used is a runoff election, where each committee member submits a ballot with a ranked list of other members for the position of chair. Unfortunately, those responsible for tabulating the votes keep losing track of which ballots still have countable votes, and so no one trusts the results.

Each committee member submits a ballot. Each ballot contains a ranked list of votes. Tabulating the votes proceeds in rounds. For the first round, the first vote on each ballot is counted. If any candidate has more than half of the votes, they win.

After each round, the candidate receiving the fewest votes (or candidates, in the case of a tie for fewest votes) is eliminated. Votes are then re-tabulated with each ballot's highest vote for a remaining candidate being counted. If all of the candidates voted for on a ballot are eliminated, then that ballot is considered "non-viable," and it is no longer counted toward the total number of ballots when calculating majority.

The process is repeated until you have a single winner, or a tie between the remaining candidates. The rules of the election guarantee you will have a tie or a winner.

Input

For each dataset:

Line 1	`<candidates> <ballots>` The number of candidates receiving votes The number of ballots, b
b **Lines**	One line per ballot. For each ballot, the names of the candidates are listed in order of preference. A candidate name is a string with no whitespace in it. A ballot may not contain votes for all candidates. No candidate will be repeated on a single ballot.

After the last dataset, a line of

```
0 0
```

will indicate the end of the input.

Output

For each dataset a single line is output:

For a winner:

```
<candidate> won
```

For a tie:

```
it is a tie between <candidate> and <candidate> [ and <candidate> [...]]
```

Each candidate name is separated by "and". They may be printed in any order.

Example

Input:

```
3 9
Buchanan Bush
Buchanan Bush
Buchanan Gore
Gore Bush
Gore Bush
Gore Bush
Gore Bush
Bush Buchanan
Bush Buchanan
0 0
```

Output:

```
Buchanan won
```

Problem F: Three Sides Make a Triangle

You work for an art store that has decided to carry every style and size of drafting triangle in existence. Unfortunately, sorting these has become a problem. The manager has given you the task of organizing them. You have decided to classify them by edge length and angles. To measure each triangle, you set it on a large sheet of very accurate graph paper and record the coordinate of each point. You then run these three points through a computer program to classify the triangles according to:

Scalene no equal sides

Isosceles two equal sides

Equilateral three equal sides

and

Acute all angles under 90

Right one angle equal 90

Obtuse one angle over 90

Of course, sometimes you make mistakes entering the data, so if you input points that do not form a valid triangle, your program should tell you.

Input

One triangle is described per line. Each line has six measurements taken to the nearest 0.001 in the order:

```
x1 y1 x2 y2 x3 y3
```

The final line of input will contain only a `-1`.

None of the test sets contain non-right angles in the range 88-92 degrees, nor do any of the test sets include any non-equal side lengths for one triangle within 0.01 of one another.

Output

You will output one line for each triangle, which will contain two words:

```
<length classification> <angle classification>
```

or

```
Not a Triangle
```

The final line of your output file will be:

```
End of Output
```

Example

Input:

```
10.000 10.000 10.000 20.000 20.000 10.000
0.000 0.000 4.000 0.000 2.000 3.464
-1
```

Output:

```
Isosceles Right
Equilateral Acute
End of Output
```

Problem G: High and Dry

You are planning a canoe trip through a tidal estuary (a network of waterways subject to tides). You can paddle only by day (sunrise to sunset). A particular difficulty is that many of the places you can consider stopping will only have enough water to float your canoe when the tide is sufficiently high, and getting stuck offshore in the mud for several hours is not your idea of a good time!

You have a list of the available docks, in the order you will encounter them. For each dock you know how many miles it is from your starting point and for how many hours before and after low tide it cannot be reached. You need not stop at each dock, and the main channel of the waterway will always have enough water for you to make progress if you wish to bypass one or more docks. You want to be sure, however, that you put in to an accessible dock by sunset on each day. You also cannot depart from a dock in the morning until enough water is available.

The time of sunrise, sunset, and low tides for each day will change slightly. So successive sunrises will not be exactly 24 hours apart nor sunsets, nor will successive low tides occur exactly 12 hours apart. Over the limited number of days for your trip, we can approximate these changes as a fixed number of minutes per day away from the ideal.

Write a program to prepare itineraries for such trips, indicating at which docks you will stop at end of each day's paddling. Each itinerary should require as few days as possible. Given a choice of itineraries with the same number of days, choose the one in which you make the most progress in the earlier days.

Input

Input consists of a series of trip descriptions. All time inputs will be presented in the form `HH:MM:SS`, where `HH` is a two digit integer indicating hours from 00–23, `MM` is a two-digit integer indicating minutes from 00–59, and `SS` is a two-digit integer indicating seconds from 00–59. For each trip description:

Line 1 Maximum number of days for the trip (integer 1–10)

Line 2 Average paddling speed in miles per hour (positive real number)

Line 3 `<sunrise> <interval>`
Time of sunrise on day 1 of the trip (05:00:00–08:00:00)
Time between successive sunrises (23:45:00–24:15:00)

Line 4 `<sunset> <interval>`
Time of sunset on day 1 (17:00:00–20:00:00)
Time between successive sunsets (23:45:00–24:15:00)

Line 5 `<lowtide> <interval>`
Time of 1st low tide on day 1 (any time)
Time between successive low tides (11:00:00–13:00:00)

Line 6 Number of docks available along the way, not counting your trip starting point but including your final destination (positive integer)

7–end `<distance> <inaccessible>`
One line for each dock (including the starting dock at distance 0.0), containing:
Distance of that dock, in miles, from your trip starting point (real)
Number of hours before and after low tide when dock is inaccessible(integer 0–12)

The final dataset is followed by a `0`.

Output

For each trip description, print a single line:

```
NO ITINERARY POSSIBLE
```

if the trip cannot be completed in the indicated number of days. If the trip can be completed, print a single line containing the numbers of the docks at which you would stop on each day, each number separated from the others by a single space.

Example

Input:

```
4
5.0
07:22:00  23:54:00
18:16:00  23:58:30
05:21:00  12:24:00
9
0.0 0
15.0 1
30.0 2
45.0 3
60.0 2
75.0 1
90.0 1
105.0 2
125.0 3
140.0 2
0
```

Output:

```
2 5 7 9
```

Problem H: Reverse Roman Notation

Hermes Poseidon (HP) has produced a new calculator, the HP CXX, using the very latest in modern technology. It supports the four basic arithmetic operations on integer values from I to MMMMCMXCIX.

In this problem, you are simulating The HP CXX. Each line of input will be either a roman numeral representation of a positive integer (between I(1) and MMMMCMXCIX(4999)), which will then be pushed to the top of the virtual stack, or it will be an arithmetic operation ($+ - */$) to be performed on the top two values of the stack. In addition, there is the $=$ operation, which is a request to print the value of the top of the stack (in roman numerals, of course).

For the $-$ operation, subtract the first number on the stack from the second. For $/$, divide the second number on the stack by the first. An attempt to divide by 0 should result in the error message "division by zero exception". When that happens, push the dividend (non-zero number) back onto the stack, but not the divisor (zero).

If an operation is requested, and there are insufficient numbers on the stack, print the error "stack underflow" and leave the stack unchanged. This applies to both the binary operations $+-*/$ and the print operation $=$.

If an attempt is made to print a number whose value is 0 or less, or whose value is greater than MMMMCMXCIX(4999), display the error message "out of range exception" and go on to the next line of input.

Roman Numerals

For those who are unfamiliar with Roman Numerals, here is a quick summary:

Each letter used in Roman numerals stands for a different number:

Roman Numeral	Number
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

A string of letters means that their values should be added together. For example, $\mathrm{XXX} = 10 + 10 + 10 = 30$, and $\mathrm{LXI} = 50 + 10 + 1 = 61$. If a smaller value is placed before a larger one, we subtract instead of adding. For instance, $\mathrm{IV} = 5 - 1 = 4$ and $\mathrm{XC} = 100 - 10 = 90$.

- Except for M, do not add more than three of the same letters together.
- Subtract only powers of ten, such as I, X, or C. Writing VL for 45 is not allowed: write XLV instead.
- Subtract only a single letter from a single numeral. Write VIII for 8, not IIX; 19 is XIX, not IXX.
- Don't subtract a letter from another letter more than ten times greater. This means that you can only subtract I from V or X, and X from L or C, so MIM is illegal.

Input

Each input line consists of either:

- A Roman numeral between I and MMMMCMXCIX, or
- An arithmetic operation $+$, $-$, $/$, or $*$, or the print operator, $=$

The input ends at the end-of-file.

Output

A line will be output:

- For every print operation, print the value at the top of the stack, or
- One of the error messages, on a line by itself:
  - `division by zero exception`
  - `stack underflow`
  - `out of range exception`

  No other output should be produced

Example

Input:

```
I
I
+
=
II
*
=
VIII
-
=
```

Output:

```
II
IV
out of range exception
```