ACM International Collegiate Programming Contest 1997/98

Sponsored by IBM

Supported by WICONA International, Wilken GmbH and GPS

Southwestern European Regional Contest

University of Ulm, Germany

November 23rd, 1997

This problem set should contain nine (9) problems on nineteen (19) numbered pages. Please inform a runner immediately if something is missing from your problem set.

ACM International Collegiate Programming Contest 1997/98
Southwestern European Regional Contest

Problem A

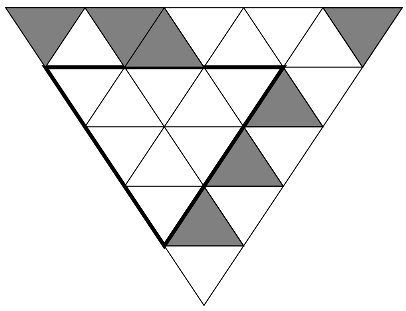
Triangles

Source: `triangle.(c|C|pas)`
Input: `triangle.in`

It is always very nice to have little brothers or sisters. You can tease them, lock them in the bathroom or put red hot chili in their sandwiches. But there is also a time when all meanness comes back!

As you know, in one month it is Christmas and this year you are honored to make the big star that will be stuck on the top of the Christmas tree. But when you get the triangle-patterned silver paper you realize that there are many holes in it. Your little sister has already cut out smaller triangles for the normal Christmas stars. Your only chance is to find an algorithm that tells you for each piece of silver paper the size of the largest remaining triangle.

Given a triangle structure with white and black fields inside you must find the largest triangle area of white fields, as shown in the following figure.



Input

The input file contains several triangle descriptions. The first line of each description contains an integer n ($1 \leq n \leq 100$), which gives the height of the triangle. The next n lines contain characters of the set { *space*, `#`, − } representing the rows of the triangle, where '`#`' is a black and '−' a white field. The spaces are used only to keep the triangle shape in the input by padding at the left end of the lines. (Compare with the sample input. The first test case corresponds to the figure.)

For each triangle, the number of the characters '`#`' and '−' per line is odd and decreases from $2n-1$ down to 1.

The input is terminated by a description starting with $n = 0$.

Output

For each triangle in the input, first output the number of the triangle, as shown in the sample output. Then print the line "`The largest triangle area is` *a*`.`", where a is the number of fields inside the largest triangle that consists only of white fields. Note that the largest triangle can have its point at the top, as in the second case of the sample input.

Output a blank line after each test case.

Sample Input

```
5
#-##----#
 -----#-
  ---#-
   -#-
    -
4
#-#-#--
 #---#
  ##-
   -
0
```

Sample Output

```
Triangle #1
The largest triangle area is 9.

Triangle #2
The largest triangle area is 4.
```

ACM International Collegiate Programming Contest 1997/98
Southwestern European Regional Contest

Problem B

Instant Complexity

Source: `complex.(c|C|pas)`
Input: `complex.in`

Analyzing the run-time complexity of algorithms is an important tool for designing efficient programs that solve a problem. An algorithm that runs in linear time is usually much faster than an algorithm that takes quadratic time for the same task, and thus should be preferred.

Generally, one determines the run-time of an algorithm in relation to the 'size' n of the input, which could be the number of objects to be sorted, the number of points in a given polygon, and so on. Since determining a formula dependent on n for the run-time of an algorithm is no easy task, it would be great if this could be automated. Unfortunately, this is not possible in general, but in this problem we will consider programs of a very simple nature, for which it is possible. Our programs are built according to the following rules (given in BNF), where $< number >$ can be any non-negative integer:

- $< Program > ::=$ `"BEGIN"` $< Statementlist >$ `"END"`
- $< Statementlist > ::= < Statement > \mid < Statement > < Statementlist >$
- $< Statement > ::= < LOOP-Statement > \mid < OP-Statement >$
- $< LOOP-Statement > ::= < LOOP-Header > < Statementlist >$ `"END"`
- $< LOOP-Header > ::=$ `"LOOP"` $< number > \mid$ `"LOOP n"`
- $< OP-Statement > ::=$ `"OP"` $< number >$

The run-time of such a program can be computed as follows: the execution of an `OP`-statement costs as many time-units as its parameter specifies. The statement list enclosed by a `LOOP`-statement is executed as many times as the parameter of the statement indicates, i.e., the given constant number of times, if a number is given, and n times, if `n` is given. The run-time of a statement list is the sum of the times of its constituent parts. The total run-time therefore generally depends on n.

Input

The input file starts with a line containing the number k of programs in the input. Following this are k programs which are constructed according to the grammar given above. Whitespace and newlines can appear anywhere in a program, but not within the keywords `BEGIN`, `END`, `LOOP` and `OP` or in an integer value. The nesting depth of the `LOOP`-operators will be at most 10.

Output

For each program in the input, first output the number of the program, as shown in the sample output. Then output the run-time of the program in terms of n; this will be a polynomial of degree ≤ 10. Print the polynomial in the usual way, i.e., collect all terms, and print it in the form "`Runtime = `a`*n^10+`b`*n^9+...+`i`*n^2+`j`*n+`k", where terms with zero coefficients are left out, and factors of 1 are not written. If the runtime is zero, just print "`Runtime = 0`".

Output a blank line after each test case.

Sample Input

```
2
BEGIN
  LOOP n
    OP 4
    LOOP 3
      LOOP n
        OP 1
      END
      OP 2
    END
    OP 1
  END
  OP 17
END

BEGIN
  OP 1997 LOOP n LOOP n OP 1 END END
END
```

Sample Output

```
Program #1
Runtime = 3*n^2+11*n+17

Program #2
Runtime = n^2+1997
```

ACM International Collegiate Programming Contest 1997/98
Southwestern European Regional Contest

Problem C

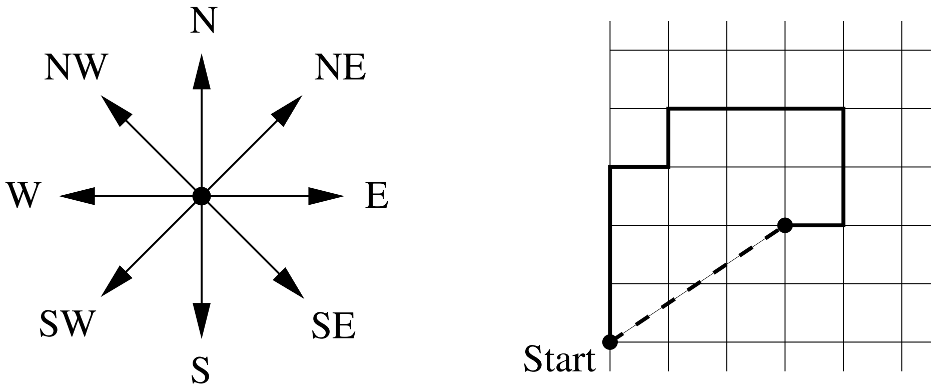
There's treasure everywhere!

Source: `treasure.(c|C|pas)`
Input: `treasure.in`

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.
```

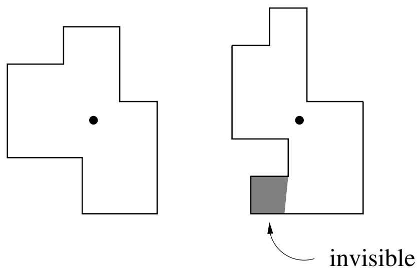
Problem D

Video Surveillance

Source: `video.(c|C|pas)`
Input: `video.in`

A friend of yours has taken the job of security officer at the Star-Buy Company, a famous department store. One of his tasks is to install a video surveillance system to guarantee the security of the customers (and the security of the merchandise of course) on all of the store's countless floors. As the company has only a limited budget, there will be only one camera on every floor. But these cameras may turn around to look in every direction.

The first problem is to choose where to install the camera for every floor. The only requirement is that every part of the room must be visible from there. In the following figure the left floor can be completely surveyed from the position indicated by a dot, while for the right floor, there is no such position, the given position failing to see the lower left part of the floor.



Before trying to install the cameras, your friend first wants to know whether there is indeed a suitable position for them. He therefore asks you to write a program that, given a ground plan, determines whether there is a position from which the whole floor is visible. All floor ground plans form rectangular polygons, whose edges do not intersect each other and touch each other only at the corners.

Input

The input file contains several floor descriptions. Every description starts with the number n of vertices that bound the floor ($4 \leq n \leq 100$). The next n lines contain two integers each, the x and y coordinates for the n vertices, given in clockwise order. All vertices will be distinct and at corners of the polygon. Thus the edges alternate between horizontal and vertical.

A zero value for n indicates the end of the input.

Output

For every test case first output a line with the number of the floor, as shown in the sample output. Then print a line stating "`Surveillance is possible.`" if there exists a position from which the entire floor can be observed, or print "`Surveillance is impossible.`" if there is no such position.

Print a blank line after each test case.

Sample Input

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

Sample Output

```
Floor #1
Surveillance is possible.

Floor #2
Surveillance is impossible.
```

Problem E

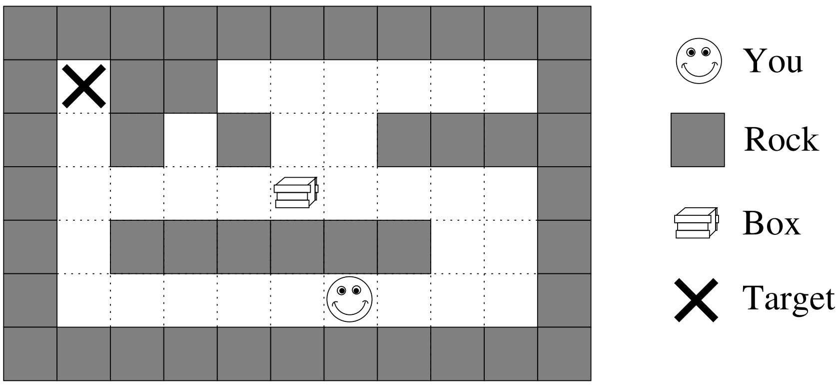
Pushing Boxes

Source: `pushing.(c|C|pas)`
Input: `pushing.in`

Imagine you are standing inside a two-dimensional maze composed of square cells which may or may not be filled with rock. You can move north, south, east or west one cell at a step. These moves are called *walks*.

One of the empty cells contains a box which can be moved to an adjacent free cell by standing next to the box and then moving in the direction of the box. Such a move is called a *push*. The box cannot be moved in any other way than by pushing, which means that if you push it into a corner you can never get it out of the corner again.

One of the empty cells is marked as the target cell. Your job is to bring the box to the target cell by a sequence of walks and pushes. As the box is very heavy, you would like to minimize the number of pushes. Can you write a program that will work out the best such sequence?



Input

The input file contains the descriptions of several mazes. Each maze description starts with a line containing two integers r and c (both ≤ 20) representing the number of rows and columns of the maze.

Following this are r lines each containing c characters. Each character describes one cell of the maze. A cell full of rock is indicated by a '`#`' and an empty cell is represented by a '`.`'. Your starting position is symbolized by '`S`', the starting position of the box by '`B`' and the target cell by '`T`'.

Input is terminated by two zeroes for r and c.

Output

For each maze in the input, first print the number of the maze, as shown in the sample output. Then, if it is impossible to bring the box to the target cell, print "`Impossible.`".

Otherwise, output a sequence that minimizes the number of pushes. If there is more than one such sequence, choose the one that minimizes the number of total moves (walks and pushes). If there is still more than one such sequence, any one is acceptable.

Print the sequence as a string of the characters `N`, `S`, `E`, `W`, `n`, `s`, `e` and `w` where uppercase letters stand for pushes, lowercase letters stand for walks and the different letters stand for the directions north, south, east and west.

Output a single blank line after each test case.

Sample Input

```
1 7
SB....T
1 7
SB..#.T
7 11
###########
#T##......#
#.#.#..####
#....B....#
#.######..#
#.....S...#
###########
8 4
....
.##.
.#..
.#..
.#.B
.##S
....
###T
0 0
```

Sample Output

```
Maze #1
EEEEE

Maze #2
Impossible.

Maze #3
eennwwWWWWeeeeeeesswwwwwwwnNN

Maze #4
swwwnnnnnneeeessssSSS
```

Problem F

Always on the run

Source: `ontherun.(c|C|pas)`
Input: `ontherun.in`

Screeching tires. Searching lights. Wailing sirens. Police cars everywhere. Trisha Quickfinger did it again! Stealing the 'Mona Lisa' had been more difficult than planned, but being the world's best art thief means expecting the unexpected. So here she is, the wrapped frame tucked firmly under her arm, running to catch the northbound metro to Charles-de-Gaulle airport.

But even more important than actually stealing the painting is to shake off the police that will soon be following her. Trisha's plan is simple: for several days she will be flying from one city to another, making one flight per day. When she is reasonably sure that the police has lost her trail, she will fly to Atlanta and meet her 'customer' (known only as Mr. P.) to deliver the painting.

Her plan is complicated by the fact that nowadays, even when you are stealing expensive art, you have to watch your spending budget. Trisha therefore wants to spend the least money possible on her escape flights. This is not easy, since airlines prices and flight availability vary from day to day. The price and availability of an airline connection depends on the two cities involved and the day of travel. Every pair of cities has a 'flight schedule' which repeats every few days. The length of the period may be different for each pair of cities and for each direction.

Although Trisha is a good at stealing paintings, she easily gets confused when booking airline flights. This is where you come in.

Input

The input file contains the descriptions of several scenarios in which Trisha tries to escape. Every description starts with a line containing two integers n and k. n is the number of cities through which Trisha's escape may take her, and k is the number of flights she will take. The cities are numbered 1, 2, ..., n, where 1 is Paris, her starting point, and n is Atlanta, her final destination. The numbers will satisfy $2 \leq n \leq 10$ and $1 \leq k \leq 1000$.

Next you are given $n(n-1)$ flight schedules, one per line, describing the connection between every possible pair of cities. The first $n-1$ flight schedules correspond to the flights from city 1 to all other cities (2, 3, ..., n), the next $n-1$ lines to those from city 2 to all others (1, 3, 4, ..., n), and so on.

The description of the flight schedule itself starts with an integer d, the length of the period in days, with $1 \leq d \leq 30$. Following this are d non-negative integers, representing the cost of the flight between the two cities on days 1, 2, ...d. A cost of 0 means that there is no flight between the two cities on that day.

So, for example, the flight schedule "`3 75 0 80`" means that on the first day the flight costs 75, on the second day there is no flight, on the third day it costs 80, and then the cycle repeats: on the fourth day the flight costs 75, there is no flight on the fifth day, etc.

The input is terminated by a scenario having $n = k = 0$.

Output

For each scenario in the input, first output the number of the scenario, as shown in the sample output. If it is possible for Trisha to travel k days, starting in city 1, each day flying to a different city than the day before, and finally (after k days) arriving in city n, then print "`The best flight costs` *x*`.`", where x is the least amount that the k flights can cost.

If it is not possible to travel in such a way, print "`No flight possible.`".

Print a blank line after each scenario.

Sample Input

```
3 6
2 130 150
3 75 0 80
7 120 110 0 100 110 120 0
4 60 70 60 50
3 0 135 140
2 70 80
2 3
2 0 70
1 80
0 0
```

Sample Output

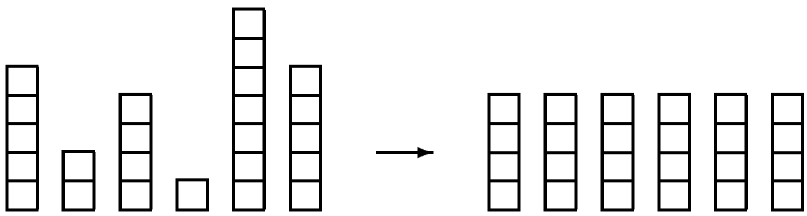
```
Scenario #1
The best flight costs 460.

Scenario #2
No flight possible.
```

Problem G

Box of Bricks

Source: `bricks.(c|C|pas)`
Input: `bricks.in`

Little Bob likes playing with his box of bricks. He puts the bricks one upon another and builds stacks of different height. "Look, I've built a wall!", he tells his older sister Alice. "Nah, you should make all stacks the same height. Then you would have a real wall.", she retorts. After a little consideration, Bob sees that she is right. So he sets out to rearrange the bricks, one by one, such that all stacks are the same height afterwards. But since Bob is lazy he wants to do this with the minimum number of bricks moved. Can you help?



Input

The input consists of several data sets. Each set begins with a line containing the number n of stacks Bob has built. The next line contains n numbers, the heights h_i of the n stacks. You may assume $1 \leq n \leq 50$ and $1 \leq h_i \leq 100$.

The total number of bricks will be divisible by the number of stacks. Thus, it is always possible to rearrange the bricks such that all stacks have the same height.

The input is terminated by a set starting with $n = 0$. This set should not be processed.

Output

For each set, first print the number of the set, as shown in the sample output. Then print the line "`The minimum number of moves is` k`.`", where k is the minimum number of bricks that have to be moved in order to make all the stacks the same height.

Output a blank line after each set.

Sample Input

```
6
5 2 4 1 7 5
0
```

Sample Output

```
Set #1
The minimum number of moves is 5.
```

Problem H

Island of Logic

Source: `island.(c|C|pas)`
Input: `island.in`

The Island of Logic has three kinds of inhabitants: divine beings that always tell the truth, evil beings that always lie, and human beings that are truthful during the day and lie at night. Every inhabitant recognizes the type of every other inhabitant.

A social scientist wants to visit the island. Because he is not able to distinguish the three kinds of beings only from their looks, he asks you to provide a communication analyzer that deduces facts from conversations among inhabitants. The interesting facts are whether it is day or night and what kind of beings the speakers are.

Input

The input file contains several descriptions of conversations. Each description starts with an integer n, the number of statements in the conversation. The following n lines each contain one statement by an inhabitant. Every statement line begins with the speaker's name, one of the capital letters A, B, C, D, E, followed by a colon '`:`'. Next is one of the following kinds of statements:

- `I am [not] ( divine | human | evil | lying ).`
- *X* `is [not] ( divine | human | evil | lying ).`
- `It is ( day | night ).`

Square brackets `[]` mean that the word in the brackets may or may not appear, round brackets `()` mean that exactly one of the alternatives separated by | must appear. *X* stands for some name from A, B, C, D, E. There will be no two consecutive spaces in any statement line, and at most 50 statements in a conversation.

The input is terminated by a test case starting with $n = 0$.

Output

For each conversation, first output the number of the conversation in the format shown in the sample output. Then print "`This is impossible.`", if the conversation cannot happen according to the rules or "`No facts are deducible.`", if no facts can be deduced. Otherwise print all the facts that can be deduced. Deduced facts should be printed using the following formats:

- *X* `is ( divine | human | evil ).`
- `It is ( day | night ).`

X is to be replaced by a capital letter speaker name. Facts about inhabitants must be given first (in alphabetical order), then it may be stated whether it is day or night.

The output for each conversation must be followed by a single blank line.

Sample Input

```
1
A: I am divine.
1
A: I am lying.
1
A: I am evil.
3
A: B is human.
B: A is evil.
A: B is evil.
0
```

Sample Output

```
Conversation #1
No facts are deducible.

Conversation #2
This is impossible.

Conversation #3
A is human.
It is night.

Conversation #4
A is evil.
B is divine.
```

Reasoning made easy

To make things clearer, we will show the reasoning behind the third input example, where A says "`I am evil.`". What can be deduced from this? Obviously A cannot be divine, since she would be lying, similarly A cannot be evil, since she would tell the truth. Therefore, A must be human, moreover, since she is lying, it must be night. So the correct output is as shown.

In the fourth input example, it is obvious that A is lying since her two statements are contradictory. So, B can be neither human nor evil, and consequently must be divine. B always tells the truth, thus A must be evil. Voilà!

Problem I

MBone

Source: `mbone.(c|C|pas)`
Input: `mbone.in`

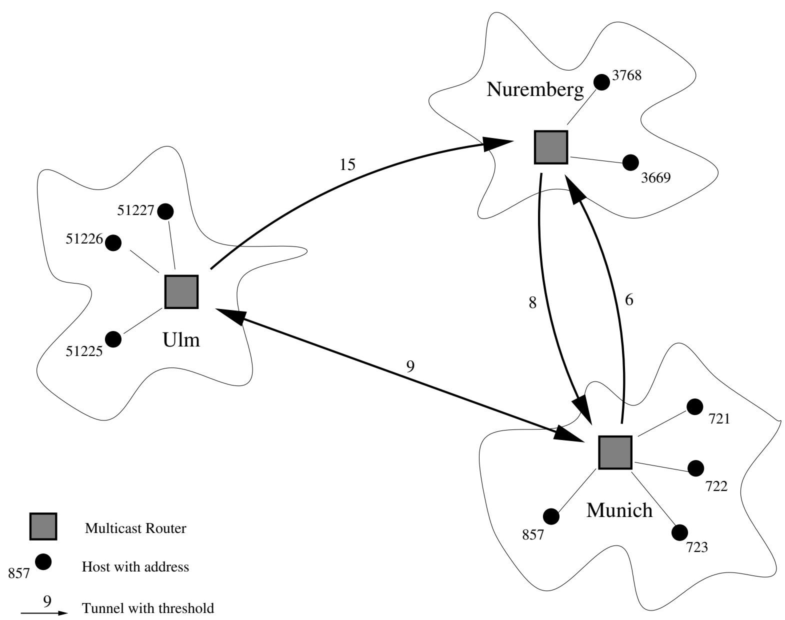
MBone is an abbreviation for 'Multicast Backbone'. It is the realization of a virtual network built on top of the Internet protocol. In contrast to connection-oriented transmission of data (unicast) and the transmission from a sender to all destinations in a network (broadcast) it provides the multicast facility, a facility to send data to all hosts that have joined a so-called 'multicast group'. All members of a group are able to send data to and receive data from the group.

Your program is to simulate a simplified version of the MBone. In our setting MBone is a combination of multicast *routers* and *hosts*, each host belonging to one of the routers. A router and the hosts that belong to it are called an *island*. Routers are connected via *tunnels* which are simple communication channels: data packets sent from one side through the tunnel are received on the other side.

In order to become a member of a multicast group, a host must send a protocol message to its corresponding multicast router specifying the address of the group it wants to join. As a consequence the host will receive all data packets sent to this group.

In order to send a data packet to a multicast group, a host sends the packet to the multicast router within its island. Every multicast router duplicates all received packets and sends them through each of its outgoing tunnels. After that it sends copies of the packet to all hosts on its island that have joined the group specified in the packet.

The distribution range of a packet within MBone is restricted through an integer value called *TTL* (Time To Live) which is assigned to every packet. If a packet is sent through a tunnel its TTL is decremented by the *threshold* (an integer value) specified for each tunnel. A packet will not be sent over a tunnel if the TTL of the packet is lower than the threshold of the tunnel.



Input

The input to your program will consist of several descriptions of MBone networks. The first part of each description defines the network topology, and the second part describes the activities on this network. The first part starts with a line containing a single integer m ($1 \leq m \leq 10$), the number of islands in the network. A value of $m = 0$ indicates the end of input. The following lines contain the descriptions for the m islands.

Each island description starts with a line containing the name of the multicast router (given as a string of at most 20 non-blank characters) followed by an integer for the number of remaining lines in the island description. These lines can be of two kinds:

Host belonging to island:	`H <Host Address>`
Tunnel:	`T <Threshold> <Dest. Name>`

$< Host\ Address >$ and $< Threshold >$ are positive integer values specifying the address of the host and the threshold of the tunnel, respectively. $< Dest.\ Name >$ is the name of the destination router at the other end of the tunnel, which is always different from the current router.

The first line of the second part contains a single integer of at most 1000 indicating the number of lines in the following activity description. Each one of these lines describes the activity of a host: join a group, leave a group or send a packet to a group.

Join a group:	`J <Host Address> <Group Address>`
Leave a group:	`L <Host Address> <Group Address>`
Send a packet to a group:	`S <Host Address> <Group Address> <Packet ID> <TTL>`

The $< Group\ Address >$,$< Packet\ ID >$ and $< TTL >$ are positive integer values with the obvious meaning. All names used for the routers and all host addresses used in a scenario, as well as all packet IDs are unique. TTLs of packets will be at most 1000. There will be at most 50 hosts and 100 tunnels in the network and at most 20 active groups (i.e, groups for which there is at least one member host) at any time. No host will try to leave a group that it is not in, nor try to join a group it is in.

Output

In the output you have to print the packets received by the hosts in the network for each scenario. If hosts receive multiple copies of a packet (routed via different paths), they keep only the copy with the highest TTL (reaching them via the 'shortest' path).

For each network description, first output the number of the network, as shown in the sample output. Each one of the subsequent lines is of the format `<Host Address> <Packet ID> <TTL>`, meaning that host `<Host Address>` received the packet having the ID `<Packet ID>` with the remaining TTL `<TTL>`. The three entries of the line should be separated by single blank characters. The output must be sorted in ascending order: first by the host address and second by the packet ID.

Output a blank line after each test case.

Sample Input

```
3
Nuremberg 3
T 8 Munich
H 3768
H 3669
Munich 6
```

```
H 721
H 722
H 723
T 6 Nuremberg
H 857
T 9 Ulm
Ulm 5
H 51225
H 51226
H 51227
T 15 Nuremberg
T 9 Munich
14
J 51227 26
J 3768 27
J 723 26
J 3768 26
S 3768 26 1000 17
J 857 26
S 3768 26 320 16
J 722 26
L 857 26
S 51227 26 1001 37
S 723 26 533 5
L 51227 26
L 3768 27
L 723 27
0
```

Sample Output

```
Network #1
722 533 5
722 1001 28
723 320 8
723 533 5
723 1000 9
723 1001 28
857 320 8
3768 320 16
3768 1000 17
3768 1001 22
51227 1000 0
51227 1001 37
```