The Twenty-first Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

PROBLEM A

System Dependencies

Input file: depend.in

Components of computer systems often have dependencies—other components that must be installed before they will function properly. These dependencies are frequently shared by multiple components. For example, both the TELNET client program and the FTP client program require that the TCP/IP networking software be installed before they can operate. If you install TCP/IP and the TELNET client program, and later decide to add the FTP client program, you do not need to reinstall TCP/IP.

For some components it would not be a problem if the components on which they depended were reinstalled; it would just waste some resources. But for others, like TCP/IP, some component configuration may be destroyed if the component was reinstalled.

It is useful to be able to remove components that are no longer needed. When this is done, components that only support the removed component may also be removed, freeing up disk space, memory, and other resources. But a supporting component, not explicitly installed, may be removed only if all components which depend on it are also removed. For example, removing the FTP client program and TCP/IP would mean the TELNET client program, which was not removed, would no longer operate. Likewise, removing TCP/IP by itself would cause the failure of both the TELNET and the FTP client programs. Also if we installed TCP/IP to support our own development, then installed the TELNET client (which depends on TCP/IP) and then still later removed the TELNET client, we would not want TCP/IP to be removed.

We want a program to automate the process of adding and removing components. To do this we will maintain a record of installed components and component dependencies. A component can be installed explicitly in response to a command (unless it is already installed), or implicitly if it is needed for some other component being installed. Likewise, a component, not explicitly installed, can be explicitly removed in response to a command (if it is not needed to support other components) or implicitly removed if it is no longer needed to support another component.

Input

The input will contain a sequence of commands (as described below), each on a separate line containing no more than eighty characters. Item names are case sensitive, and each is no longer than ten characters. The command names (**DEPEND**, **INSTALL**, **REMOVE** and **LIST**) always appear in uppercase starting in column one, and item names are separated from the command name and each other by one or more spaces. All appropriate **DEPEND** commands will appear before the occurrence of any **INSTALL** command that uses them. There will be no circular dependencies. The end of the input is marked by a line containing only the word **END**.

Command Syntax	*Interpretation/Response*
`DEPEND item1 item2 [item3 …]`	**item1** depends on **item2** (and **item3** …)
`INSTALL item1`	install **item1** and those on which it depends
`REMOVE item1`	remove **item1**, and those on which it depends, if possible
`LIST`	list the names of all currently-installed components

Output

Echo each line of input. Follow each echoed **INSTALL** or **REMOVE** line with the actions taken in response, making certain that the actions are given in the proper order. Also identify exceptional conditions (see *Expected Output,* below, for examples of all cases). For the **LIST** command, display the names of the currently installed components. No output, except the echo, is produced for a **DEPEND** command or the line containing **END**. There will be at most one dependency list per item.

Sample Input

```
DEPEND   TELNET TCPIP NETCARD
DEPEND TCPIP NETCARD
DEPEND DNS TCPIP NETCARD
DEPEND  BROWSER   TCPIP  HTML
INSTALL NETCARD
INSTALL TELNET
INSTALL foo
REMOVE NETCARD
INSTALL BROWSER
INSTALL DNS
LIST
REMOVE TELNET
REMOVE NETCARD
REMOVE DNS
REMOVE NETCARD
INSTALL NETCARD
REMOVE TCPIP
REMOVE BROWSER
REMOVE TCPIP
LIST
END
```

Output for the Sample Input

```
DEPEND   TELNET TCPIP NETCARD
DEPEND TCPIP NETCARD
DEPEND DNS TCPIP NETCARD
DEPEND  BROWSER   TCPIP  HTML
INSTALL NETCARD
   Installing NETCARD
INSTALL TELNET
   Installing TCPIP
   Installing TELNET
INSTALL foo
   Installing foo
REMOVE NETCARD
   NETCARD is still needed.
INSTALL BROWSER
   Installing HTML
   Installing BROWSER
INSTALL DNS
   Installing DNS
LIST
   HTML
   BROWSER
   DNS
   NETCARD
   foo
   TCPIP
   TELNET
REMOVE TELNET
   Removing TELNET
REMOVE NETCARD
   NETCARD is still needed.
REMOVE DNS
   Removing DNS
REMOVE NETCARD
   NETCARD is still needed.
INSTALL NETCARD
   NETCARD is already installed.
REMOVE TCPIP
   TCPIP is still needed.
REMOVE BROWSER
   Removing BROWSER
   Removing HTML
   Removing TCPIP
REMOVE TCPIP
   TCPIP is not installed.
LIST
   NETCARD
   foo
END
```

PROBLEM B
Jill Rides Again
Input file: jill.in

Jill likes to ride her bicycle, but since the pretty city of Greenhills where she lives has grown, Jill often uses the excellent public bus system for part of her journey. She has a folding bicycle which she carries with her when she uses the bus for the first part of her trip. When the bus reaches some pleasant part of the city, Jill gets off and rides her bicycle. She follows the bus route until she reaches her destination or she comes to a part of the city she does not like. In the latter event she will board the bus to finish her trip.

Through years of experience, Jill has rated each road on an integer scale of "niceness." Positive niceness values indicate roads Jill likes; negative values are used for roads she does not like. Jill plans where to leave the bus and start bicycling, as well as where to stop bicycling and re-join the bus, so that the sum of niceness values of the roads she bicycles on is maximized. This means that she will sometimes cycle along a road she does not like, provided that it joins up two other parts of her journey involving roads she likes enough to compensate. It may be that no part of the route is suitable for cycling so that Jill takes the bus for its entire route. Conversely, it may be that the whole route is so nice Jill will not use the bus at all.

Since there are many different bus routes, each with several stops at which Jill could leave or enter the bus, she feels that a computer program could help her identify the best part to cycle for each bus route.

Input
The input file contains information on several bus routes. The first line of the file is a single integer b representing the number of route descriptions in the file. The identifier for each route (r) is the sequence number within the data file, $1 \le r \le b$. Each route description begins with the number of stops on the route: an integer s, $2 \le s \le 20{,}000$ on a line by itself. The number of stops is followed by $s-1$ lines, each line i ($1 \le i < s$) is an integer n_i representing Jill's assessment of the niceness of the road between the two stops i and $i+1$.

Output
For each route r in the input file, your program should identify the beginning bus stop i and the ending bus stop j that identify the segment of the route which yields the maximal sum of niceness, $m = n_i+n_{i+1}+\ldots+n_{j-1}$. If more than one segment is maximally nice, choose the one with the longest cycle ride (largest $j-i$). To break ties in longest maximal segments, choose the segment that begins with the earliest stop (lowest i). For each route r in the input file, print a line in the form:

```
The nicest part of route r is between stops i and j.
```

However, if the maximal sum is not positive, your program should print:

```
Route r has no nice parts.
```

Sample Input

```
3
3
  -1
   6
10
   4
  -5
   4
  -3
   4
   4
  -4
   4
  -5
4
  -2
  -3
  -4
```

Output for the Sample Input

```
The nicest part of route 1 is between stops 2 and 3
The nicest part of route 2 is between stops 3 and 9
Route 3 has no nice parts
```

PROBLEM C
Morse Mismatches
Input file: morse.in

Samuel F. B. Morse is best known for the coding scheme that carries his name. Morse code is still used in international radio communication. The coding of text using Morse code is straightforward. Each character (case is insignificant) is translated to a predefined sequence of *dits* and *dahs* (the elements of Morse code). Dits are represented as periods (".") and dahs are represented as hyphens or minus signs ("–"). Each element is transmitted by sending a signal for some period of time. A dit is rather short, and a dah is, in perfectly formed code, three times as long as a dit. A short silent space appears between elements, with a longer space between characters. A still longer space separates words. This dependence on the spacing and timing of elements means that Morse code operators sometimes do not send perfect code. This results in difficulties for the receiving operator, but frequently the message can be decoded depending on context.

In this problem we consider reception of words in Morse code without spacing between letters. Without the spacing, it is possible for multiple words to be coded the same. For example, if the message "dit dit dit" were received, it could be interpreted as "EEE", "EI", "IE" or "S" based on the coding scheme shown in the sample input. To decide between these multiple interpretations, we assume a particular context by expecting each received word to appear in a dictionary.

For this problem your program will read a table giving the encoding of letters and digits into Morse code, a list of expected words (*context*), and a sequence of words encoded in Morse code (*morse*). These *morse* words may be flawed. For each *morse* word, your program is to determine the matching word from *context*, if any. If multiple words from *context* match *morse*, or if no word matches perfectly, your program will display the best matching word and a mismatch indicator.

If a single word from *context* matches *morse* perfectly, it will be displayed on a single line, by itself. If multiple *context* words match *morse* perfectly, then select the matching word with the fewest characters. If this still results in an ambiguous match, any of these matches may be displayed. If multiple *context* words exist for a given *morse*, the matching word will be displayed followed by an exclamation point ("!").

We assume only a simple case of errors in transmission in which elements may be either truncated from the end of a *morse* word or added to the end of a *morse* word. When no perfect matches for *morse* are found, display the word from *context* that matches the longest prefix of *morse*, or has the fewest extra elements beyond those in *morse*. If multiple words in *context* match using these rules, any of these matches may be displayed. Words that do not match perfectly are displayed with a question mark ("?") suffixed.

The input data will only contain cases that fall within the preceding rules.

Input

The Morse code table will appear first and consists of lines each containing an uppercase letter or a digit *C*, zero or more blanks, and a sequence of no more than six periods and hyphens giving the Morse code for *C*. Blanks may precede or follow the items on the line. A line containing a single asterisk ("*"), possibly preceded or followed by blanks, terminates the Morse code table. You may assume that there will be Morse code given for every character that appears in the *context* section.

The *context* section appears next, with one word per line, possibly preceded and followed by blanks. Each word in *context* will contain no more than ten characters. No characters other than upper case letters and digits will appear. Thered will be at most 100 *context* words. A line containing only a single asterisk ("*"), possibly preceded or followed by blanks, terminates the *context* section.

The remainder of the input contains *morse* words separated by blanks or end-of-line characters. A line containing only a single asterisk ("*"), possibly preceded or followed by blanks, terminates the input. No *morse* word will have more than eighty (80) elements.

Output

For each input *morse* word, display the appropriate matching word from *context* followed by an exclamation mark ("!") or question mark ("?") if appropriate. Each word is to appear on a separate line starting in column one.

Sample Input

```
A       .-
B       -...
C       -.-.
D       -..
E       .
F       ..-.
G       --.
H       ....
I       ..
J       .---
K       -.-
L       .-..
M       --
N       -.
O       ---
P       .--.
Q       --.-
R       .-.
S       ...
T       -
U       ..-
V       ...-
W       .--
X       -..-
Y       -.--
Z       --..
0       ------
1       .-----
2       ..---
3       ...--
4       ....-
5       .....
6       -....
7       --...
8       ---..
9       ----.
*
AN
EARTHQUAKE
EAT
GOD
HATH
IM
READY
TO
WHAT
WROTH
*
.--.....--   .....--....
--.----..   .--.-.----..
.--.....--   .--.
..-.-.-....--.-..-.--.-.
..--   .-...--..-.--
----        ..--
*
```

Output for the Sample Input

```
WHAT
HATH
GOD
WROTH?
WHAT
AN
EARTHQUAKE
IM!
READY
TO
IM!
```

The Twenty-first Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

PROBLEM D

RAID!

Input file: raid.in

RAID (Redundant Array of Inexpensive Disks) is a technique which uses multiple disks to store data. By storing the data on more than one disk, RAID is more fault tolerant than storing data on a single disk. If there is a problem with one of the disks, the system can still recover the original data provided that the remaining disks do not have corresponding problems.

One approach to RAID breaks data into blocks and stores these blocks on all but one of the disks. The remaining disk is used to store the parity information for the data blocks. This scheme uses *vertical parity* in which bits in a given position in data blocks are exclusive ORed to form the corresponding parity bit. The parity block moves between the disks, starting at the first disk, and moving to the next one in order. For instance, if there were five disks and 28 data blocks were stored on them, they would be arranged as follows:

Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
Parity for 1-4	Data block 1	Data block 2	Data block 3	Data block 4
Data block 5	Parity for 5-8	Data block 6	Data block 7	Data block 8
Data block 9	Data block 10	Parity for 9-12	Data block 11	Data block 12
Data block 13	Data block 14	Data block 15	Parity for 13-16	Data block 16
Data block 17	Data block 18	Data block 19	Data block 20	Parity for 17-20
Parity for 21-24	Data block 21	Data block 22	Data block 23	Data block 24
Data block 25	Parity for 25-28	Data block 26	Data block 27	Data block 28

With this arrangement of disks, a block size of two bits and even parity, the hexadecimal sample data 6C7A79EDFC (01101100 01111010 01111001 11101101 11111100 in binary) would be stored as:

Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
00	01	10	11	00
01	10	11	10	10
01	11	01	10	01
11	10	11	11	01
11	11	11	00	11

If a block becomes unavailable, its information can still be retrieved using the information on the other disks. For example, if the first bit of the first block of disk 3 becomes unavailable, it can be reconstructed using the corresponding parity and data bits from the other four disks. We know that our sample system uses even parity:

$$0 \oplus 0 \oplus ? \oplus 1 \oplus 0 = 0$$

So the missing bit must be 1.

An arrangement of disks is invalid if a parity error is detected, or if any data block cannot be reconstructed because two or more disks are unavailable for that block.

Write a program to report errors and recover information from RAID disks.

Input

The input consists of several disk sets.

Each disk set has 3 parts. The first part of the disk set contains three integers on one line: the first integer d, $2 \le d \le 6$, is the number of disks, the second integer s, $1 \le s \le 64$, is the size of each block in bits, and the third integer b, $1 \le b \le 100$, is the total number of data and parity blocks on each disk. The second part of the disk set is a single letter on a line, either "E" signifying even parity or "O" signifying odd parity. The third part of the disk set contains d lines, one for each disk, each holding $s \times b$ characters representing the bits on the disk, with the most significant bits first. Each bit will be specified as "0" or "1" if it holds valid data, or "x" if that bit is unavailable. The end of input will be a disk set with $d = 0$. There will be no other data for this set which should not be processed.

Output

For each disk set in the input, display the number of the set and whether the set is valid or invalid. If the set is valid, display the recovered data bits in hexadecimal. If necessary, add extra "0" bits at the end of the recovered data so the number of bits is always a multiple of 4. All output shall be appropriately labeled.

Sample Input

```
5 2 5
E
0001011111
0110111011
1011011111
1110101100
0010010111
3 2 5
E
0001111111
0111111011
xx11011111
3 5 1
O
11111
11xxx
x1111
0
```

Output for the Sample Input

```
Disk set 1 is valid, contents are: 6C7A79EDFC
Disk set 2 is invalid.
Disk set 3 is valid, contents are: FFC
```

The Twenty-first Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

PROBLEM E

Optimal Routing

Input file: routing.in

Acme Courier Message, Inc. (ACM) is planning to add a service for delivery of documents and small parcels. ACM will group parcels and documents in bags, which will be transported by car among different stations for intermediate handling and routing prior to final delivery. ACM is in the initial stages of determining the workload requirements for transporting bags among stations.

When a driver delivers a bag, she will (if possible) locate and pick up another bag for delivery to another station, continuing in this manner until there are no more deliverable bags. A deliverable bag is one that can be picked up and delivered to its destination by a driver prior to the end of her workday. The total time for a driver's workday begins with the time of pickup of her first bag and includes the time she spends delivering bags, the time in transit, and the time waiting at stations for deliverable bags. ACM would like its drivers to spend the maximum amount of time possible delivering the bags between stations within a normal workday. In addition, ACM wants drivers' final destinations to be the same as the stations where they started whenever possible.

You must write a program to determine optimal driver routes for several ACM scenarios. Each scenario describes bags and stations for a single workday. In this simple version, routes for all drivers will originate from the same station, which we call station A. Optimal routes are subject to the following restrictions.

1. A driver's normal workday will not exceed 10 hours.
2. Drivers will travel from one station to another with one bag, if one is available for pickup. If there are no deliverable bags at a station, the driver will proceed to another station that has a scheduled deliverable bag to continue her route.
3. If several different routes with a final destination of station A are possible, the one requiring the longest cumulative delivery time is optimal. If there are more than one with the longest cumulative delivery time, the one with the shortest total workday time is optimal.
4. Whenever possible, the final destination of a driver is station A. However, if it is impossible to schedule a final destination of station A, then the route requiring the longest cumulative delivery time is optimal. If there are more than one with the longest cumulative delivery time, the one with the shortest total workday time is optimal.
5. Every bag that originates from station A will be delivered. (Some bags originating at other stations will not necessarily be delivered.) No bag will be delivered more than once.

The optimal route for the driver who picks up the first available deliverable bag at station A is completely determined before any consideration of subsequent drivers. The optimal route of the second driver, who picks up the next available deliverable bag at station A that has not already been scheduled for delivery by the first driver, is completely determined next. The optimal route determination continues in this manner until all the bags that can be delivered have been scheduled for delivery. Undeliverable bags will be identified and reported. Throughout the entire process, each driver will be routed according to the bags not already scheduled earlier for delivery. In all scenarios the time to travel from station A to any other station is 10 hours or less.

Input

Input for each scenario comes in two parts: a list of the bags and a table of times required to drive between stations. The first line in each scenario consists of an integer *n* representing the number of bags to be delivered. The next *n* lines describe each bag in the following format:

id origin destination time

where *id* is the bag identification number (integer), *origin* and *destination* are the station labels for the bag's origin and destination (uppercase letters), and time is when the bag is available for transport. The format for *time* is *hhmm*, where *hh* and *mm* are integers representing time on a 24-hour clock varying from 0001 to 2400. Data on a line are

separated by single blanks. Each station is labeled with a unique uppercase letter. Bags may appear in any time order in the list. The end of input is signified by a scenario for which the number of bags is 0.
Input data for the table of driving times consist of lines of the form:

station1 station2 time

where *station1* and *station2* are uppercase letters and *time* is as described earlier. Transit times between stations are listed for all stations which are included in the list of bags. Transit times are bidirectional. Different scenarios are completely unrelated.

Output

Output for each scenario begins by identifying the scenario by number (Scenario 1, Scenario 2, etc.). Following that is a listing of each driver's optimal route. Each route begins with the number of the driver (Driver 1, Driver 2, etc.) and then a summary of the driver's route including all transits between stations in the order in which the stations were visited. For transits which deliver a bag, display the bag identification number and its origin and destination stations For transits which do not deliver a bag, display the origin and destinations stations. Output for each driver is summarized by the total delivery time and the total workday time in the form *hhmm*, following the time format specified in the input of time values. If two different routes for a driver are optimal, then output may show either one. The final section of output for a scenario will include a listing of all undeliverable bags or a statement indicating successful delivery of all bags. Each section of a scenario and each scenario should be separated by a blank line.

Sample Input

```
7
1 A B 0800
3 A C 0850
2 B C 0700
6 B D 1250
5 B C 1400
7 C A 1600
8 D C 1130
A B 0400
A C 0135
A D 0320
B C 0345
B D 0120
C D 0200
0
```

Output for the Sample Input

```
Scenario 1

Driver 1
  Bag #1 from station A to station B
  Bag #2 from station B to station C
  Bag #7 from station C to station A
  Total delivery time: 0920
  Total workday time: 0935

Driver 2
  Bag #3 from station A to station C
   -->Transit without delivery from station C to station B
  Bag #5 from station B to station C
  Total delivery time: 0520
  Total workday time: 0905

Undelivered Bags:
  Bag #8 remains at station D
  Bag #6 remains at station B
```

The Twenty-first Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

PROBLEM F

Do You Know the Way to San Jose?

Input file sanjose.in

The Internet now offers a variety of interactive map facilities, so that users can see either an overview map of a large geographic region or can "zoom in" to a specific street, sometimes even a specific building, on a much more detailed map. For instance, downtown San Jose might appear in a map of California, a map of Santa Clara county, and a detailed street map.

Suppose you have a large collection of rectangular maps and you wish to design a browsing facility that will process a sequence of map requests for locations at various levels of map detail. Locations are expressed using *location names*. Each location name has a unique pair of real coordinates (x,y). Maps are unique, labeled with identifying *map names*, and defined by two pairs of real coordinates—(x_1,y_1) (x_2,y_2)—representing opposite corners of the map. All map edges are parallel to the standard Cartesian x and y axes. A map and a location can have the same name. The *aspect ratio* of a map is the ratio of its height to its width (where width is measured in the x direction and height is measured in the y direction).

The level of detail of a map can be approximated by using the rectangular area represented by the map; i.e., assume that a map covering a smaller area contains more detailed information. Maps can overlap one another. If a location (x,y) lies within two or more maps having equal areas, the preferred map (at that level of detail) is the one in which the location is nearest the center of the map. If the location is equidistant from the centers of two overlapping maps of the same area, then the preferred map (at that level of detail) is the one whose aspect ratio is nearest to the aspect ratio of the browser window, which is 0.75. If this still does not break the tie, then the preferred map is the one in which the location is furthest from the lower right corner of the map (this heuristic is intended to minimize the need for scrolling in the user's browser window). Finally, if there is still a tie, then the preferred map is the one containing the smallest x-coordinate.

The *maximum detail level* available for a given location is the maximal number of maps of different areas that contain the location. Clearly, different locations can have different maximum detail levels. The map at detail i for the location is the map with the ith largest area among a maximal set of maps of the distinct area containing the location. Thus, the map at detail level 1 for the location will be the least detailed (largest area) map containing the location and the map at the maximum detail level will be the most detailed (smallest area) map containing the location.

Input

The input file consists of a set of maps, locations, and requests; it is organized as follows:

- The word "MAPS", in all uppercase letters and on a line by itself, introduces a set of one or more maps. Following the set heading, each map is described by a single line consisting of a map name (an alphabetic string with no leading, trailing, or embedded blanks) and two real coordinate pairs—x_1 y_1 x_2 y_2—representing opposite corners of the map.
- The word "LOCATIONS", in all uppercase letters and on a line by itself, introduces a set of one or more locations. Following this heading, each location is described by a line consisting of a location name (an alphabetic string with no leading, trailing, or embedded blanks) and a real coordinate pair—x y—representing the center of the location.
- The word "REQUESTS", in all uppercase letters and on a line by itself, introduces a set of zero or more requests. Following this heading, each request is described by a line consisting of a location name (an alphabetic string with no leading, trailing, or embedded blanks) followed by a positive integer representing the desired detail level for that location.
- The word "END", in all uppercase and on a line by itself, terminates the file.

All map and location data preceding the requests are valid. There will be no duplicate maps. The result of processing a valid request is the name of the map containing the given location at the given detail level (using the tie-breaking rules described above). Invalid requests can result from requesting unknown location names, locations that do not appear in any map, or detail levels that exceed the number of maps of different areas containing the location.

The following example should illustrate all these definitions:

Output

Each request must be echoed to the output. If the request is valid, display the name of the map satisfying the request. If the location is not on a map, display a message to that effect. If the location is on the map but the detail level is too large, display the name of the map of the smallest available area (largest possible detail level).

Sample Input

```
MAPS
BayArea -6.0 12.0 -11.0 5.0
SantaClara 4.0 9.0 -3.5 2.5
SanJoseRegion -3.0 10.0 11.0 3.0
CenterCoast -5.0 11.0 1.0 -8.0
SanMateo -5.5 4.0 -12.5 9.0
NCalif -13.0 -7.0 13.0 15.0
LOCATIONS
Monterey -4.0 2.0
SanJose -1.0 7.5
Fresno 7.0 0.1
SanFrancisco -10.0 8.6
SantaCruz -4.0 2.0
SanDiego 13.8 -19.3
REQUESTS
SanJose 3
SanFrancisco 2
Fresno 2
Stockton 1
SanDiego 2
SanJose 4
SantaCruz 3
END
```

Output for the Sample Input

```
SanJose at detail level 3 using SanJoseRegion
SanFrancisco at detail level 2 using BayArea
Fresno at detail level 2 no map at that detail level; using NCalif
Stockton at detail level 1 unknown location
SanDiego at detail level 2 no map contains that location
SanJose at detail level 4 using SantaClara
SantaCruz at detail level 3 no map at that detail level; using CenterCoast
```

The Twenty-first Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

PROBLEM G

Spreadsheet Tracking

Input file tracking.in

Data in spreadsheets are stored in cells, which are organized in rows (*r*) and columns (*c*). Some operations on spreadsheets can be applied to single cells (*r*,*c*), while others can be applied to entire rows or columns. Typical cell operations include inserting and deleting rows or columns and exchanging cell contents.

Some spreadsheets allow users to mark collections of rows or columns for deletion, so the entire collection can be deleted at once. Some (unusual) spreadsheets allow users to mark collections of rows or columns for insertions too. Issuing an insertion command results in new rows or columns being inserted before each of the marked rows or columns. Suppose, for example, the user marks rows 1 and 5 of the spreadsheet on the left for deletion. The spreadsheet then shrinks to the one on the right.

	1	2	3	4	5	6	7	8	9
1	22	55	66	77	88	99	10	12	14
2	2	24	6	8	22	12	14	16	18
3	18	19	20	21	22	23	24	25	26
4	24	25	26	67	22	69	70	71	77
5	68	78	79	80	22	25	28	29	30
6	16	12	11	10	22	56	57	58	59
7	33	34	35	36	22	38	39	40	41

	1	2	3	4	5	6	7	8	9
1	2	24	6	8	22	12	14	16	18
2	18	19	20	21	22	23	24	25	26
3	24	25	26	67	22	69	70	71	77
4	16	12	11	10	22	56	57	58	59
5	33	34	35	36	22	38	39	40	41

If the user subsequently marks columns 3, 6, 7, and 9 for deletion, the spreadsheet shrinks to this.

	1	2	3	4	5
1	2	24	8	22	16
2	18	19	21	22	25
3	24	25	67	22	71
4	16	12	10	22	58
5	33	34	36	22	40

If the user marks rows 2, 3 and 5 for insertion, the spreadsheet grows to the one on the left. If the user then marks column 3 for insertion, the spreadsheet grows to the one in the middle. Finally, if the user exchanges the contents of cell (1,2) and cell (6,5), the spreadsheet looks like the one on the right.

	1	2	3	4	5
1	2	24	8	22	16
2					
3	18	19	21	22	25
4					
5	24	25	67	22	71
6	16	12	10	22	58
7					
8	33	34	36	22	40

	1	2	3	4	5	6
1	2	24		8	22	16
2						
3	18	19		21	22	25
4						
5	24	25		67	22	71
6	16	12		10	22	58
7						
8	33	34		36	22	40

	1	2	3	4	5	6
1	2	22		8	22	16
2						
3	18	19		21	22	25
4						
5	24	25		67	22	71
6	16	12		10	24	58
7						
8	33	34		36	22	40

You must write tracking software that determines the final location of data in spreadsheets that result from row, column, and exchange operations similar to the ones illustrated here.

Input and Output

The input consists of a sequence of spreadsheets, operations on those spreadsheets, and queries about them. Each spreadsheet definition begins with a pair of integers specifying its initial number of rows (r) and columns (c), followed by an integer specifying the number (n) of spreadsheet operations. Row and column labeling begins with 1. The maximum number of rows or columns of each spreadsheet is limited to 50. The following n lines specify the desired operations.

An operation to exchange the contents of cell (r_1, c_1) with the contents of cell (r_2, c_2) is given by:

EX r_1 c_1 r_2 c_2

The four insert and delete commands—DC (delete columns), DR (delete rows), IC (insert columns), and IR (insert rows)Ñare given by:

<command> A x_1 x_2 … x_a

where *<command>* is one of the four commands; A is a positive integer less than 10, and x_1, … x_A are the labels of the columns or rows to be deleted or inserted before. For each insert and delete command, the order of the rows or columns in the command has no significance. Within a single delete or insert command, labels will be unique.

The operations are followed by an integer which is the number of queries for the spreadsheet. Each query consists of positive integers r and c, representing the row and column number of a cell in the original spreadsheet. For each query, your program must determine the current location of the data that was originally in cell (r, c). The end of input is indicated by a row consisting of a pair of zeros for the spreadsheet dimensions.

For each spreadsheet, your program must output its sequence number (starting at 1). For each query, your program must output the original cell location followed by the final location of the data or the word GONE if the contents of the original cell location were destroyed as a result of the operations. Separate output from different spreadsheets with a blank line.

The data file will not contain a sequence of commands that will cause the spreadsheet to exceed the maximum size.

Sample Input

```
7 9
5
DR   2  1 5
DC  4  3 6 7 9
IC  1  3
IR  2  2 4
EX 1 2 6 5
4
4 8
5 5
7 8
6 5
0 0
```

Output for the Sample Input

```
Spreadsheet #1
Cell data in (4,8) moved to (4,6)
Cell data in (5,5) GONE
Cell data in (7,8) moved to (7,6)
Cell data in (6,5) moved to (1,2)
```

The Twenty-first Annual ACM International Collegiate

Programming Contest Finals

sponsored by

Microsoft®

Problem H

Window Frames

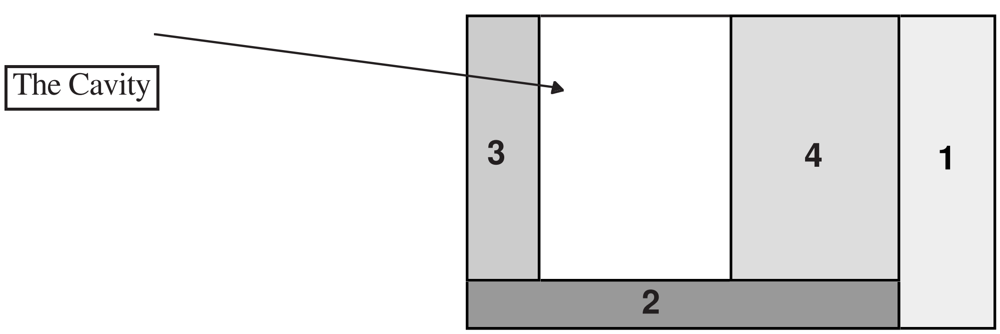
Input file window.in

Elements of graphical user interfaces include such things as buttons, text boxes, scroll bars, drop-down menus and scrollable list boxes. Each is considered to be a special kind of object called a widget. Where these widgets are placed, how much space they are allocated, and how they change size constitutes the geometry of a window.

One geometry management scheme uses special rectangular widgets called *frames* to contain and thus group other widgets. A frame is a *parent* if some or all of its own space is allocated to additional frames, which are its *children*. The frame which has no parent is called the *root frame*; its size is specified by the user (in the input data). This problem requires that you determine the allocation of space to, and the position of frames placed in root frames of various sizes.

The *cavity* in a frame is the space in the frame that is not occupied by its children. When a new child frame is created, it is allocated an entire horizontal strip along the top or bottom edge of the cavity (this is called a *horizontal child*) or an entire vertical strip along the right or left edge of the cavity (this is called a *vertical child*). Thus, as a result of creating a new child, the cavity becomes smaller, but it remains rectangular. The process of placing children inside the enclosing frame is called *packing*. Children are positioned in the cavity according to the order in which they are packed.

The figure below shows the child frames of a parent frame. Frame 1 along the right edge was packed first, then frame 2 along the bottom edge, frame 3 along the left edge, and finally frame 4 along the right edge. The cavity, shown in white, contains available space for packing subsequent child frames.



Each frame covers a rectangular grid of pixels. If the root frame covers c columns and r rows of pixels, then the pixel in the top left corner is at coordinate (0,0) and the pixel in the lower right corner is at coordinate (c–1, r–1). The position of a frame is specified by the coordinates of its upper left corner pixel and its lower right corner pixel.

Each frame has minimum dimensions determined by an input parameter d and the minimum dimensions of its children. A frame must be at least large enough to pack all of its children. The minimum dimensions of each frame are determined as follows:

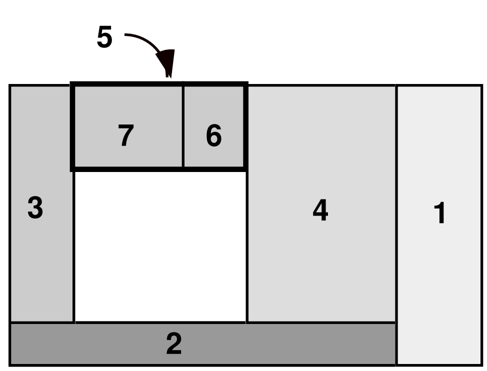
Packing Side	Frame Type	Minimum Width	Minimum Height
Right or left	Vertical	Maximum of *d* and the width necessary for the frame's children	Maximum of 1 and the height necessary for the frame's children
Bottom or top	Horizontal	Maximum of 1 and the width necessary for the frame's children	Maximum of *d* and the height necessary for the frame's children

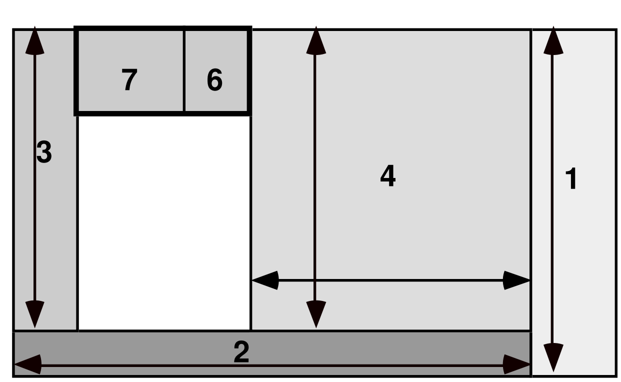
When a frame is larger than the minimum dimensions just specified, the additional interior space is apportioned to its children and/or its cavity. Each frame has an expansion flag (which is an input parameter) that, when set, indicates a vertical frame can grow wider or a horizontal frame can grow taller. For example, a frame with its expansion flag set, allocated space along the top of the cavity, can grow taller, with the extra height extending downward.

The distribution of additional horizontal space in a frame is handled as follows. Let x be the number of horizontal pixels by which the parent frame exceeds its minimum width. If n, the number of the vertical children in the frame with their expansion flags set, is non-zero, then the x pixels are distributed among the n vertical children. If q is the quotient of x divided by n and r is the remainder, then each of the n vertical frames grows wider by q pixels and the first r of them that were packed in the frame grow wider by 1 pixel in addition to the q. If n is zero, then none of the vertical children grow wider, and the x pixels are added to the width of the cavity. In either case, the horizontal children in the enlarged frame become wider, if necessary, in such a manner as to ensure the single cavity remains rectangular.

The distribution of additional vertical space in a parent frame to its children and/or its cavity is handled in a manner similar to that used to distribute additional horizontal space, with the appropriate change in direction of growth. Only the horizontal children with their expansion flags set grow taller to utilize the additional vertical pixels, and if none of the horizontal children have their expansion flags set, the additional pixels are added to the height of the cavity. As expected, the vertical children also become taller, if necessary, to ensure the rectangular and uniqueness properties of the cavity.

In the next illustration, the root frame on the left has been enlarged to yield the one on the right. Frames 6 and 7 are horizontal and vertical children, respectively, of frame 5. Only frames 4, 6 and 7 have their expansion flags set. In the frame on the right, the additional horizontal and vertical space has been distributed to the children so as to result in the growth indicated by the arrows. Note that frame 7 does not change size because no room is available for expansion in its parent, frame 5. Frame 6 does not change size for the same reason.





Input

The input consists of a sequence of root frames, their descendants, and different potential root frame sizes. Each item in the sequence corresponding to a single root has the following format:

M N — *M* is the total number of frames excluding the root. *N* is the number of different root sizes (both are positive integers).

followed by *M* lines of the form:

n p s d e where: *n* is the name of the frame (a positive integer);
p is the name of the parent (where 0 is the root frame);
s is one of the characters “L”, “R”, “T”, and “B” indicating packing side;
d is the minimum dimension (a positive integer); and
e is 0 or 1, where 0 means the expansion flag is cleared, 1 means it is set;

followed by *N* lines of the form:

c r — where *c* is the number of columns of pixels, and *r* is the number of rows of pixels in the root frame (both positive integers).

Root frames are not listed. Frame numbers for a given root are distinct. Children of a frame will not appear in the input before their parents. Frames are packed in the order in which they appear in the input. The end of input is signified by a line with *M* and *N* both 0.

Output

Begin the output of each root by writing its record number (1 for the first, 2 for the second, etc.). For each size corresponding to that root, write the size (rows × columns) and then list the name of each frame along with the coordinates of its upper left and lower right corners. List the frames in the order in which they are packed in their parents, with the root’s first child and its descendants first, the second child and its descendants second, and so on. If the root size is too small to pack its frames, print the message “is too small” instead of attempting to list the frames. Separate output for different root sizes by a line of dashes.

Sample Input

```
7 1
1 0 R 50 0
2 0 B 10 0
3 0 L 40 0
4 0 R 20 1
5 0 T 30 0
6 5 R 20 0
7 5 L 10 1
1000 1000
2 2
1 0 R 100 1
2 0 T 30 1
100 50
200 100
0 0
```

Output for the Sample Input

```
Root Frame #1
--------------------------------------------
  Display: 1000 X 1000
   Frame: 1  (950,0)  (999,999)
   Frame: 2  (0,990)  (949,999)
   Frame: 3  (0,0)  (39,989)
   Frame: 4  (70,0)  (949,989)
   Frame: 5  (40,0)  (69,29)
   Frame: 6  (50,0)  (69,29)
   Frame: 7  (40,0)  (49,29)
--------------------------------------------


Root Frame #2
--------------------------------------------
  Display: 100 X 50 is too small
--------------------------------------------
  Display: 200 X 100
   Frame: 1  (1,0)  (199,99)
   Frame: 2  (0,0)  (0,99)
--------------------------------------------
```