Twenty-first Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	**Filename**
Revenge of the Egg Guy	roboto
Will and Ned’s Excellent Cram Session	radio
House of the Living Impaired	zombies
The Epic of Aldez: The Mirror of Deceit	epic
Ali's Disorganized Discs	organize
Guitar Zero	zero
Spirit Splash	splash
Cheaper Processing Units	cpu
QUEE MAR	marquee
Kart Racing	karts

Call your program file: *filename*.c, *filename*.cpp, or *filename*.java
Call your input file: *filename*.in

For example, if you are solving Guitar Zero:
Call your program file: zero.c, zero.cpp or zero.java
Call your input file: zero.in

Revenge of the Egg Guy
(A.k.a. Domo Arigato, Dr. Roboto)

Filename: `roboto`

Dr. Roboto, sometimes referred to as the Egg Guy due to his rotund physique, is an evil genius with a particular aptitude for the construction of robots, space stations, and the like. In fact, his robots have just recently finished construction of a monumental space station, known as the Doom Egg. In a night of celebration and villainy, Dr. Roboto designed the Doom Ray, the ultimate weapon, capable of wiping all porcupines, hedgehogs, and any other spiky mammals off the face of the Earth. Surprisingly enough, a certain hedgehog has actually given Dr. Roboto a great deal of trouble in the past, so he's got a bit of a grudge. There's a problem with this Doom Ray, though. Dr. Roboto was celebrating a little too hard when he designed it. Although he's certain that the Doom Ray will work, he suspects that there might be a short circuit in the wiring that would cause the central battery of the Doom Egg to overload, blowing the station to smithereens and him along with it. The wiring diagram is a huge mess, so Dr. Roboto needs you to write a program to determine whether or not a short circuit exists.

The Problem:

Dr. Roboto's design consists of a number of circuit junctions and connections between pairs of different junctions. Two of the junctions represent terminals of the Doom Egg's central battery. If there is any zero-resistance path between the two terminals of the Doom Egg's central battery, this is a short circuit which will destroy the Doom Egg at the same time it destroys the world's spiky critters. If the zero-resistance path goes through multiple junctions, this is still a short circuit as the junctions themselves are all made out of cryogenically cooled superconductors that have no resistance. For example, suppose that the positive and negative terminals are both connected to junction 3 with zero resistance, but have no direct connection to each other. This would be a short circuit since current can flow unimpeded from the negative terminal to junction 3 to the positive terminal. It is possible that a particular circuit design has no path linking the two terminals at all. This is just fine since the circuit may operate by principles of inductance and capacitance which aren't modeled in the input.

The Input:

Input consists of a positive integer indicating the number of Doom Ray designs, followed immediately by that many designs. A Doom Ray design begins with an integer, n ($2 \le n \le 50$), indicating the number of circuit junctions in the design, including the battery terminals. The first circuit junction represents the positive terminal of the Doom Egg's central battery and the second junction represents the negative terminal, respectively. The next n lines each contain n space-delimited integers in the range from -1 to 500, with the i^{th} integer on the j^{th} line indicating the resistance of the connection between the i^{th} and j^{th} junction. A resistance value of -1 represents the absence of a connection. It is guaranteed that the i^{th} integer on the j^{th} line will be equal to the j^{th} integer on the i^{th} line, and that the i^{th} integer on the i^{th} line will be 0.

The Output:

For each Doom Ray design, print a message on a line by itself indicating whether or not the design contains a short circuit. Begin with a header "`Circuit Design #`*`n`*`: `" where *n* represents the number of the circuit design (beginning with 1) for each message. Then, if the design contains a short circuit, print the statement:

```
Back to the drawing board
```

Otherwise, print the statement:

```
No more hedgehog troubles
```

Sample Input:

```
2
3
 0 -1  0
-1  0  0
 0  0  0
5
 0 -1  0 27 -1
-1  0 -1 -1  0
 0 -1  0 -1 15
27 -1 -1  0  0
-1  0 15  0  0
```

Sample Output:

```
Circuit Design #1: Back to the drawing board
Circuit Design #2: No more hedgehog troubles
```

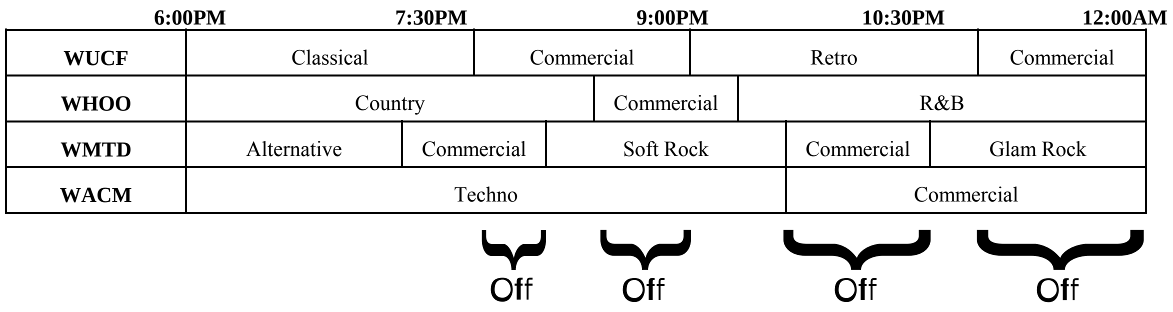
Will and Ned's Excellent Cram Session

Filename: `radio`

It's the night before a big history exam and Will and Ned, being the procrastinators that they are, have not studied one bit. In the hopes of not failing and flunking out of high school, they desperately want to cram as much as they can. Unfortunately for them, they enjoy listening to the radio too much and will not dare turn it off while their favorite stations are playing. As a result, their productivity seems to drop when it's on. Even worse, when they listen to too many commercials on their favorite stations they just break down and stop working altogether. As they ponder this problem, a bright flash of light and booming sound appear and a giant time traveling cardboard box appears. Out of the box, their friend, Roojus, appears and says "Hello Everybody! How are you doing?" Roojus has come to help Will and Ned. Excellent!

Using his time traveling cardboard box, Roojus is able to gather the radio airing schedules for their favorite stations, including a complete list of commercial breaks. Roojus has planned a study session with Will and Ned between 6:00PM and 12:00AM and is going to set up a schedule of which time frames have too many commercials. While Will and Ned are listening to the radio, Roojus is going to keep the studying light. But when there are too many commercials on, Roojus will turn off the radio so they can get down to what Roojus calls "pure cramming." While he's preparing for the session, he'd like you to help plan the study schedule. Will and Ned can't concentrate when there are two or more commercials on; so during that time Roojus wants the radio off.

Radio Schedule



The Problem:

Your job is to find out how much "pure cramming" time they can get in within that 6 hour period. Assume that the studying ends at 12:00AM, sharp. Roojus wants them to be well rested for the big exam. Because there may be multiple exams in the future, Roojus wants to be able to use your program to schedule all their study sessions.

The Input:

The first line of input will contain, *n*, the number of study sessions in the input. The *n* study sessions follow. For each study session, there will be a positive integer, *k* ($2 \leq k \leq 10$), on the first line by itself, representing the number of stations to which Will and Ned like to listen. For the next *k* lines, each line will contain two non-negative integers, *x* and *y* ($0 < x, y < 360$), detailing the commercial schedule of a particular station. The first, *x,* is the amount of time the station runs its programming between commercials in minutes (within a station, all programming between commercials is the same length). The second, *y,* is the length of commercial breaks in minutes (again within a station, each commercial break is always the same length). Each station starts off at 6:00PM with programming and alternates between programming and commercial breaks.

The Output:

The output for each test case should be a single line of the following format:

```
Study session #s has m minute(s) of pure cramming.  Excellent!
```

where *s* is the number of the current study session starting from 1, and *m* is the total number of minutes that Roojus will keep the radio off for pure cramming time.

Sample Input:

```
3
3
15 5
15 10
5 5
2
30 20
30 20
3
350 10
340 10
330 10
```

Sample Output:

```
Study session #1 has 125 minute(s) of pure cramming.  Excellent!
Study session #2 has 140 minute(s) of pure cramming.  Excellent!
Study session #3 has 0 minute(s) of pure cramming.  Excellent!
```

House of the Living Impaired

Filename: `zombies`

Crystel loves to play a video game where she goes around and shoots zombies. In this video game Crystel is given unlimited ammo for her shotgun so she can righteously vanquish as many zombies as possible. Crystel wants to write a program that will tell her the amount of zombies she can destroy if the she follows a certain path. Crystel's path can be broken down into multiple steps, each in one of four directions: North, South, East, or West. In addition, when she is in the same location as a zombie she will always destroy it (they really scare her). She had asked one of her friends to solve this problem for her, but he thought that it would be more up your alley. He did mention, though, that for Crystel to destroy a zombie she must be at the same position as the zombie due to the shotgun's short range.

The Problem:

Given the positions of the zombies and Crystel's path, determine how many zombies she destroys.

The Input:

The first line will contain a single positive integer, *n*, where *n* represents the number of game levels that will be run.

Each run will begin with a line containing two integers, *w* and *h* ($1 \leq w, h \leq 25$), where *w* and *h* represent the width and height of the playing area, respectively. The next line in the case will contain a non-negative integer, *z*, where *z* represents the number of zombies in the playing area (which will be no more than the area of the playing field since two zombies cannot stand in the same spot). This will be followed by *z* lines each containing two integers representing the *x* and *y* coordinates of each zombie. Each zombie is guaranteed to be within the playing area. Finally, there will be a non-negative integer, *s*, representing the number of steps Crystel will take. This will be followed by *s* lines that describe Crystel's path. The lines will have a single character depicting which direction Crystel is going to go, either: '`N`', '`S`', '`W`', or '`E`'.

Crystel will always start at the position (1, 1), which is the bottom-left (south-west) corner of the playing area. From the starting position, (1, 2) would be a step North; (2, 1) would be a step East. Crystel will also never leave the field, but she might retrace her steps.

The Output:

For each game level print a single line of the form:

```
Level i: Crystel vanquished v zombies.
```

where *i* is the game level, and *v* is how many zombies were destroyed.

Sample Input:

```
2
5 5
1
3 3
3
N
E
W
6 6
2
2 2
3 3
5
N
N
E
E
S
```

Sample Output:

```
Level 1: Crystel vanquished 0 zombies.
Level 2: Crystel vanquished 1 zombies.
```

The Epic of Aldez: The Mirror of Deceit

Filename: `epic`

Oh no! It's happened again! The evil dwarf magician, Cannondwarf, has taken the princess Aldez captive. This time, he's brought together pieces of an ancient broken mirror in attempt to unite the worlds of light and dark. Once merged, he plans to bring treachery and deceit to Lowrule, the kingdom of light. Like the epics passed through the generations, the land of Lowrule is in need of its green-clothed hero, Chain. But this time, Chain needs your help!

It's been almost a year now since the announcement of the next title in The Epic of Aldez series of hit games. As an anxious gamer, you have your copy preordered for your GameSquare home entertainment system. Unfortunately, the GameSquare has been unofficially discontinued and will soon be replaced by the over-hyped and relatively rare Woo system. You may replace your preorder with a version for the Woo, so you decide to do some research about the game.

In The Epic of Aldez titles, Chain has an unfortunate habit of being chosen to save Lowrule from the deceitful Cannondwarf. To do so, Chain must slash his way through many elaborate dungeons and defeat the boss in each one. This will weaken Cannondwarf. When all of Cannondwarf's minions are defeated, Chain has a chance at a man-to-dwarf fight, giving Lowrule the upper hand.

Each boss can only be threatened by a special weapon hidden deep within its own lair. It would be easy for Chain if the weapon was readily available, but that wouldn't be very... epic... now would it? While the dungeons seem complicated at first, Chain has noticed (through decades of unwanted experience) that the dungeons are much easier to navigate when given a descriptive map. Better yet, he can always describe the maps with an ASCII block of text. Each map indicates the locations of several locked doors, mini-bosses, treasures, and other surprises.

You have found out that the only difference between the GameSquare and Woo version of the newest installment is how Chain uses the dungeon maps. Unfortunately, Chain has a problem understanding the maps between the two versions of the game, and cannot navigate through any dungeon to the boss. Chain loves to go through dungeons as fast as possible, but he needs your help to do this in the new version. You must write the program that helps him.

The Problem:

While ASCII maps have aided Chain in his past epics, some newly added confusion complicates things. To help you help him, he has provided an ancient tablet. Unknown to him, it is a description for each of the map's markings.

Symbols:

```
B: Boss
C: Treasure Chest
D: Unlocked Door
E: Entrance
G: Giant Key
L: Locked Door
M: Mini-Boss
S: Small Key
W: Weapon
*: Wall
.: Floor
```

Example Dungeon Map:

```
*********B******
**....*...*W.*S*
*S.**...*.**M*.*
*********.**L*.*
*..............*
*L******L*****.*
*....S*.....G*C*
*E**************
```

Given this information, you must process each dungeon map to help Chain find his way out of the dungeon. Chain entered at the Entrance (`E`) and can walk through Floor (`.`) squares, but not through Wall (`*`) squares. The only way out is to find the Weapon (`W`) that kills the boss and the Giant Key (`G`) that opens the Boss (`B`) room. Once Chain enters a Boss room with the correct weapon, consider it an easy victory. There is often a Miniboss (`M`) that Chain may choose to defeat (which may or may not be required in order to exit the dungeon). Unfortunately, Chain may also need to find several Small Keys (`S`) to unlock the Locked doors (`L`) along the way. Once a door is unlocked, the key breaks. Therefore, Chain must find a small key for each locked door (note that he must find the key in an accessible area). Unlocked doors (`D`) require no key and may be used at any time. There are also Treasure Chests (`C`), but Chain does not need to open these, as he always seems to have a full wallet.

The game designers have done a very good job testing the GameSquare version of the game. In each dungeon, Chain will be able to find the weapon before he enters the Boss room. Also, in order for Chain to encounter one of Cannondwarf's minions, he will have to use all of the available small keys. You may assume that the area surrounding the map is inaccessible. Of course, there is always at least one route from the entrance to the boss room.

After investing some additional time researching the games, you have finally determined why Chain is having trouble navigating through the dungeons. When looking at a particular map, the left side in the GameSquare version has become the right side in the Woo version, and vice versa. Luckily for you, Chain still understands how to navigate through the GameSquare version. So, you must write a program to help Chain through each dungeon of the Woo version by mirroring each map along the vertical axis.

The Input:

There will be multiple dungeons described in the input. Each dungeon starts with two integers, *m* and *n* ($5 \le m, n \le 80$), on a line by themselves separated by a single space. The next *n* lines will contain exactly *m* characters each. Only characters from the tablet will appear in the ASCII map descriptions. The last dungeon will be followed by a line containing two zeroes. This line should not be processed.

The Output:

For each dungeon, you must first output the header "`Dungeon x:`" where *x* is the dungeon number, starting from 1. Then, output the corresponding mirrored dungeon map. That is, each row of the map must be reversed so Chain may find his way. After each map, output a blank line.

Sample Input:

```
5 5
.D..E
W*...
****.
.....
G.*B*
16 8
*********B******
**....*...*W.*S*
*S.**...*.**M*.*
*********.**L*.*
*..............*
*L******L*****.*
*....S*.....G*C*
*E**************
0 0
```

Sample Output:

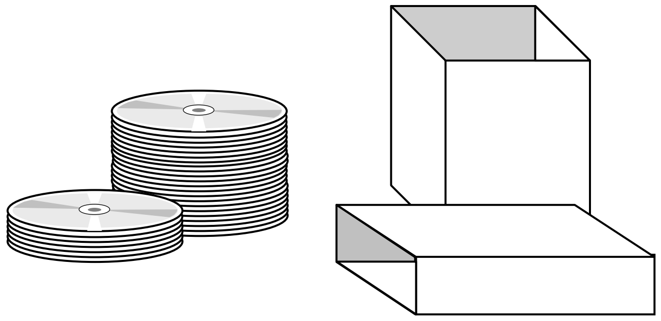
```
Dungeon 1:
E..D.
...*W
.****
.....
*B*.G

Dungeon 2:
******B*********
*S*.W*...*....**
*.*M**.*...**.S*
*.*L**.*********
*..............*
*.*****L******L*
*C*G.....*S....*
**************E*
```

Ali's Disorganized Discs

Filename: `organize`

Ali has tons of game CDs and DVDs for all of his family's different console game systems, but unfortunately his kids have lost all of the disc cases. He would like to organize these discs, so he got a bunch of different boxes to put them in.



Ali already got his kids to separate the discs into perfectly-aligned stacks, as shown. Now he wants you to help him find a box to hold each stack.

If your program works, Ali's wife might even use it to reorganize the dinner plates in their kitchen cabinets.

The Problem:

Write a program to determine whether a stack of discs will fit into an empty rectangular box, when given the interior dimensions of the box, and the number of discs in the stack.

Every CD and DVD is exactly 120mm in diameter. When stacked up, each disc is close to 2mm in thickness—for this problem, we'll assume exactly 2mm, so a stack of 5 discs is exactly 10mm high. The entire stack must be kept together with the outer edges aligned, even when placed in the box. The bottom disc in the stack must align flat against one of the interior surfaces of the box. The aligned surface might happen to be the front or back of the box, the bottom of the box, or merely a side, depending on how the box is oriented.

The Input:

The input will contain several descriptions of empty boxes and stacks of discs to be compared. Each description will contain exactly four positive integers separated by spaces, all on the same line. The first three numbers are the dimensions of the box, in millimeters. The last number is the number of CDs/DVDs in the stack, and will be no greater than 5000.

The last description line will be followed by a line containing four zeroes.

The Output:

For each description line in the input, your program should output a single line. If the stack of discs fits entirely within the box, output a line of the form:

```
Box b: Stack of s discs fits!
```

If the stack of discs does not fit into the box, output a line of the form:

```
Box b: Stack of s discs does not fit.
```

where *b* is the number of the description line in the input, starting from 1, and *s* is the number of discs in the stack.

Sample Input:

```
120 200 600 300
40  400 100  25
100 150 150  12
0 0 0 0
```

Sample Output:

```
Box 1: Stack of 300 discs fits!
Box 2: Stack of 25 discs does not fit.
Box 3: Stack of 12 discs fits!
```

Guitar Zero

Filename: `zero`

Years ago, you saw a kid playing a dancing game in your local mall. It was fairly obvious that the kid played the game several hours a day, since he was very good at it. However, he still could not actually dance. As the sales marketer for the Unfunded Computing Foundation, you have decided to see if this simplistic concept would work for your favorite instrument, the guitar. Unfortunately, the overseeing review board doesn't share your enthusiasm and has declined your proposal for research and development. They have a 24-hour appeal policy, but it started last night. You only have a few hours left to convince the pesky board members that you have a multi-million-dollar idea.

The Problem:

You and your team have decided that a quick proof-of-concept is all that is needed given the time crunch. Since you have no money, you won't be able to use any equipment. However, that's okay because you and the developers brought your official Air Guitars. Some of the more experienced air guitarists are already shredding it up to their favorite bands instead of working, so it's a killer time to test the scoring system. The model is as follows: You always start off with a score of 0. As you play your air guitar to your favorite MP3, your friends will increase or decrease your score by one. When the song has completed, you must have a positive (non-zero) score to prove that you have chops and deserve to be crowned a shredder. Of course, a score of zero or lower will only impress your mom, making you a guitar zero to everyone else.

The Input:

There will be multiple songs. Each song starts with a single integer, n ($1 \leq n < 80$), on a line by itself. The next line will contain exactly n characters, each either a '`+`' or '`-`'. These represent each time your score increased or decreased, respectively. The last song is followed by a line containing the number zero. This line should not be processed.

The Output:

For each song, you must first output a header, "`Song` *`x`*`: `", where *x* is the song number starting from 1. Following the song number, you should indicate how the air guitarist performed on the song. If the air guitar player passed the song (the sum of the votes is positive), output "`Shreddin`". Otherwise, output "`Guitar Zero`". Each song should be output on a single line.

Sample Input:

```
10
+-+-++-+-+
10
---+----+-
5
+---+
11
+++++++++-+
0
```

Sample Output:

```
Song 1: Shreddin
Song 2: Guitar Zero
Song 3: Guitar Zero
Song 4: Shreddin
```

Spirit Splash

Filename: `splash`

UCF celebrates homecoming each football season. There are several events planned during Homecoming Week, but none are anticipated as much as the Spirit Splash! This event is the one time during the year when any student (or faculty!) can jump into UCF's semi-circular Reflecting Pond on campus in a unique pep rally. Of course, as UCF has grown it is getting harder and harder to fit all of the students into the pond. Therefore, the administration wants to expand the Reflecting Pond by making it rectangular, but isn't sure by how much.

The Problem:

Given the length and width of a new rectangular Reflecting Pond, determine its area.

The Input:

Input will begin with a single positive integer, *n,* on a line by itself. On the next *n* lines will be two positive integers separated by a single space representing the length and width (in meters) of a potential new Reflecting Pond.

The Output:

For each line of input, print the area (in square meters) of the new potential Reflecting Pond on a separate line. All output values are guaranteed to fit within a 32-bit integer value.

Sample Input:

```
2
100 10
1750 2050
```

Sample Output:

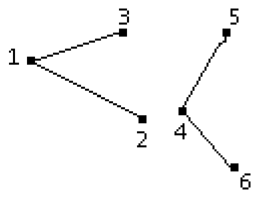
```
1000
3587500
```

Cheaper Processing Units

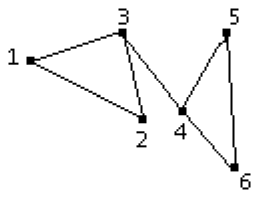
Filename: `cpu`

United Computer Fabrication (UCF) is designing a new microprocessor, the Knightron XL, that will eclipse even the latest offerings from the other two major CPU makers. UCF plans on using this new offering to take over the lucrative gamer and enthusiast markets. To do this, UCF plans to make the Knightron XL as affordable as possible and undersell the other two manufacturers (the less a gamer has to spend on his CPU, the more he or she can spend on the rest of the rig).

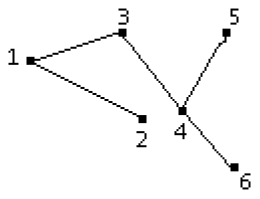
There are many potential designs for the Knightron XL. Each design incorporates several subsystems on a silicon die that must be connected to each other with gold wire traces. Even such a tiny amount of gold wire can cost a great deal as hundreds of thousands of processors are produced. So UCF's first step in reducing cost is to minimize the amount of gold wire used to connect the subsystems. There are many different ways to properly connect the subsystems. See the diagrams below:



System 1

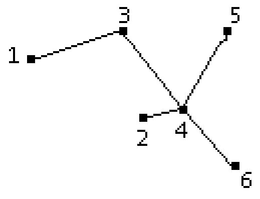


System 2



System 3

System 1 is no good because there is no way for subsystems 1, 2, or 3 to communicate with subsystems 4, 5, or 6. This design is not properly connected. System 2 is properly connected, but there are redundant connections between some of the subsystems, making the design inefficient. If we remove the connections between subsystems 2 and 3 and subsystems 5 and 6, as in System 3, the amount of wire used has been greatly reduced. However, this still is not the most efficient solution. Note that subsystem 2 is much closer to subsystem 4 than it is to subsystem 1. If we remove the connection between subsystems 1 and 2 and add a connection between subsystems 2 and 4, as in the figure below, we obtain an even more efficient design.



The Problem:

UCF wants to be able to rapidly identify Knightron XL designs that require the smallest amount of wire to connect the subsystems. Your job as head of the software development at UCF is to write a program that will automatically analyze CPU designs and output the smallest length of gold wire needed to properly connect the subsystems.

The Input:

There will be multiple designs to consider. Input will begin with a single integer, *n,* on the first line, indicating how many designs are being considered. Each design will begin with a single integer, *s* ($1 \le s \le 20$), on a line by itself, indicating how many subsystems are in this particular design. This will be followed by *s* lines, each line representing one of the subsystems (subsystem 1 is on the first line, subsystem 2 is on the second, etc.). On each of the *s* lines, there will be *s* non-negative integers, each representing the amount of wire (in micrometers) required to connect from this subsystem to another subsystem (the first integer represents subsystem 1, the second represents subsystem 2, etc.). The length of wire from any subsystem to itself is zero and the length of wire between two connected subsystems is both positive and symmetric (the same amount of wire would be required to go either direction between the two subsystems).

The Output:

For each design, print "`Design x: `" where *x* represents the number of the design (starting at 1), followed by the minimum amount of gold wire, *w*, needed to properly connect the subsystems together in the format "`w micrometers`" as shown in the Sample Output.

Sample Input:

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

Sample Output:

```
Design 1: 3 micrometers
Design 2: 5 micrometers
```

QUEE MAR

Filename: `marquee`

Glenn’s Sign Company is expanding into new markets. Previously, they focused entirely on fixed signs and hired graphical artists to do their work. Glenn is now interested in expanding his company into electronic signs – that is, marquees.

A marquee is a sign where a programmed message scrolls across an electronic display, allowing for longer messages than fixed billboards. Since this is all new to Glenn’s Sign Company, they’re hiring lots of new programmers and artists to work on these new signs.

The Problem:

The signs have a message that is to be scrolled, as well as a fixed number of characters that they can display at any one time. You are to simulate one full cycle of the marquee showing each part of the message as it would be displayed for each movement until it wraps around. A movement is defined as the leftmost character disappearing from the display, the remaining characters shifting to the left by one slot, and the next character appearing on the rightmost slot.

The Input:

The first line of the input will contain a single positive integer, *n,* representing the number of signs in the file. The next 2*n* lines will represent the individual signs. The first line of each sign will contain a string, *s* (of between 1 and 99 characters, inclusive)*,* representing the message that will be programmed into the sign. The message will contain only uppercase and lowercase letters as well as spaces. There will be no leading or trailing spaces on the line (that is, the first and last character on the line will be an uppercase or lowercase letter). The second line of each sign will contain a single integer, *m* ($1 \le m \le 50$), representing the number of characters the sign can display at any one time.

The Output:

For each sign, the output should first contain the sign header, “`Sign #`*`n`*`:`”, on a line by itself where *`n`* is the number of the sign (beginning with 1). Next, each sign should contain, in sequential order, the series of displays (each on a line by itself) the marquee should have using square brackets (‘`[`‘ and ‘`]`’) to show where the edges of the sign are. Each sign should be displayed for one full cycle, starting with the first character of the message in the leftmost slot on the sign. There should be a blank line separating the output for each sign. If *m* is greater than or equal to the length of *s*, then the sign does not need to cycle and only the original message should be displayed left-justified (use spaces in the output for unused sign characters). When the marquee is wrapping around, there should be exactly one space inserted between the last character and first character of the message. All other spacing should be preserved exactly as it appears in the input file.

Sample Input:

```
3
UCF Knights
5
Computer
10
ACM
2
```

Sample Output:

```
Sign #1:
[UCF K]
[CF Kn]
[F Kni]
[ Knig]
[Knigh]
[night]
[ights]
[ghts ]
[hts U]
[ts UC]
[s UCF]
[ UCF ]

Sign #2:
[Computer  ]

Sign #3:
[AC]
[CM]
[M ]
[ A]
```

Kart Racing

Filename: `karts`

A certain mustachioed plumber and friends are about to have their annual kart race. The winner of the race will win the prized mushroom-shaped trophy. However, a certain dragonesque king of turtle-like creatures has decided he'd be better off making a fortune in coins by betting on the races instead of trying to win for himself. In the middle of the night, the evil dragon king sneaks into the garage and takes each kart for a test drive, measuring the kart's speed (oddly, each kart travels with a constant speed, without ever slowing down or speeding up). While the evil dragon king was out, he noticed a certain giant ape (who was wearing a tie) walking around the course eating giant bananas and then carelessly dropping the banana peels on the road.

The evil dragon king found that these banana peels could not be avoided by any kart, and that every peel made a kart slip for five seconds, effectively stopping it in place. Once a kart hits a banana peel, the peel is knocked from the road and subsequent karts will not be affected by it. Note that karts can freely pass other karts at any time. It is guaranteed that every course will only have one winner (no ties), and also guaranteed that only one kart will reach a given banana (two karts won't reach the same banana at the same time). In addition, bananas are very slippery so no two bananas will ever be at the same location.

The Problem:

In order to decide who to bet on, the evil dragon king has kidnapped you to write a program to determine who will win the race.

The Input:

The first line of the input will contain a single positive integer, *y*, representing the number of circuits that the evil dragon king wants tested. Each circuit will begin with a single positive integer, *m* ($m < 9$), on a line by itself. Each of the following *m* lines will contain the data for one racer: the name of the racer (a string containing only upper and lowercase characters and hyphens, from 1 to 30 characters long) and the speed of the racer (an integer between 1 and 30 meters/second, inclusive), each separated by a single space from each other. Luckily, no racers have the same name within a single circuit so each name will appear in the list once.

Following the *m* lines, there will be a line with a single integer, *n* ($0 < n < 11$), representing the number of courses on the race circuit. On each of the following *n* lines there will be one race course description: the name of the course (a string containing only upper and lowercase characters and hyphens, 1 to 30 characters long), the length of the course in meters (an integer between 1 and 1000, inclusive), an integer *b* ($0 \leq b < 11$) representing the number of banana peels on the course, and finally *b* integers (each one between 1 and the course length, inclusive) denoting the distance from the track's starting point to a banana peel. Within a circuit, each course has a unique name. Each component of the course description will be separated by a single space.

The Output:

For each circuit, first output, on a line by itself, a header "`Circuit #`*x*`:`" where *x* represents the circuit number (beginning with 1). For each of the courses within the circuit, list the name of the course followed by a colon and one space, and then the name of the racer who first travels the entire length of the course followed by one space and the phrase "`is the winner!`". Leave a blank line after the output for each circuit.

Sample Input:

```
2
3
Apple 9
Mario 10
Frog 5
2
Brother-Track 60 3 10 14 59
Rainbow-Track 600 4 1 184 397 436
2
Mario 30
Ape 4
1
Dinosaur-Track 200 2 194 14
```

Sample Output:

```
Circuit #1:
Brother-Track: Apple is the winner!
Rainbow-Track: Mario is the winner!

Circuit #2:
Dinosaur-Track: Mario is the winner!
```