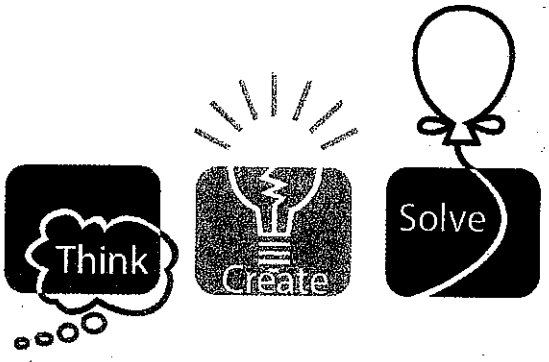
University of Central Florida



2010 (Fall) Local Programming Contest

Problems

Problem#	Filename	Problem Name
1	nih	NIH Budget
2	binarize	Binarize It
3	knights	Knights Exchange
4	texting	g2g c u l8r
5	tipit	Tip to be Palindrome
6	plate	Plate Spinning
7	soda	The Eternal Quest for Caffeine
8	lca	Lowest Common Ancestor
9	shortcut	Pegasus Circle Shortcut
10	soccer	Soccer Standings
11	stars	Interstellar Love
12	exam	Dr. Orooji's Exam
13	maze	The Puzzle Master

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

For example, if you are solving Knights Exchange:

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

UCF Local Contest — September 4, 2010

NIH Budget

filename: `nih`

Recently, a job for an algorithms specialist opened up at NIH. You never thought you'd be using your expertise in algorithms to save lives, but now, here is your chance! While the doctors are very good in carrying out medical research and coming up with better cures for diseases, they are not so good with numbers. This is where you come in.

You have been tasked to allocate money for all disease research at NIH. The interesting thing about disease research is that the number of lives saved doesn't linearly increase with the amount of money spent, in most cases. Instead, there are "break-points". For example, it might be the case that for disease A, we have the following break-points:

Research Funding	Lives Saved
10 million	5
50 million	100
100 million	1000
250 million	1100

If you spend more money than one breakpoint and less than another, the number of lives saved is equal to the amount saved for the previous breakpoint. (In the above example, if you spent $150 million, you'd still only save 1000 lives, and if you spent any amount more than $250 million, you'd still save 1100 lives.)

The doctors have figured out charts just like this one for all the diseases for which they do research. Given these charts, your job will be to maximize the number of lives saved spending no more than a particular budget.

The Problem:

Given several charts with information about how much has to be spent to save a certain number of lives for several diseases and a maximum amount of money you can spend, determine the maximum number of lives that can be saved.

The Input:

The first input line contains a positive integer, n ($n \leq 100$), indicating the number of budgets to consider. The first line of each budget contains two positive integers, d ($d \leq 10$), representing the number of diseases for which there is data and B ($B \leq 100000$), the total budget, in millions of dollars. The following d lines contain information about each of the d diseases. Each of these lines will contain exactly four ordered pairs of positive integers separated by spaces. Each pair will represent a dollar level (in millions) followed by the number of lives saved for that dollar level of funding. Each of the pairs will be separated by spaces as well. Each of these values will be less than or equal to 100,000. Assume that the dollar levels on an input line are distinct and in

increasing order, and that the number of lives saved on an input line are also distinct and in increasing order.

The Output:

For each test case, just output a line with the following format:

```
Budget #k: Maximum of x lives saved.
```

where k is the number of the budget, starting at 1, and x is the maximum number of lives saved in that budget.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
3
2 2000
10 5 50 100 100 1000 250 1100
100 1 200 2 300 3 1900 1000
3 100
10 100 40 200 70 300 100 500
5 1 25 2 35 3 50 4
200 10000 300 20000 400 30000 500 40000
1 10
100 2 200 3 300 5 400 6
```

Sample Output:

```
Budget #1: Maximum of 2000 lives saved.

Budget #2: Maximum of 500 lives saved.

Budget #3: Maximum of 0 lives saved.
```

UCF Local Contest — September 4, 2010

Binarize It

filename: `binarize`

Professor Boolando can only think in binary, or more specifically, in powers of 2. He converts any number you give him to the smallest power of 2 that is equal to or greater than your number. For example, if you give him 5, he converts it to 8; if you give him 100, he converts it to 128; if you give him 512, he converts it to 512.

The Problem:

Given an integer, your program should binarize it.

The Input:

The first input line contains a positive integer, *n*, indicating the number of values to binarize. The values are on the following *n* input lines, one per line. Each input will contain an integer between 2 and 100,000 (inclusive).

The Output:

At the beginning of each test case, output "`Input value: v`" where `v` is the input value. Then, on the next output line, print the binarized version.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
3
900
16
4000
```

Sample Output:

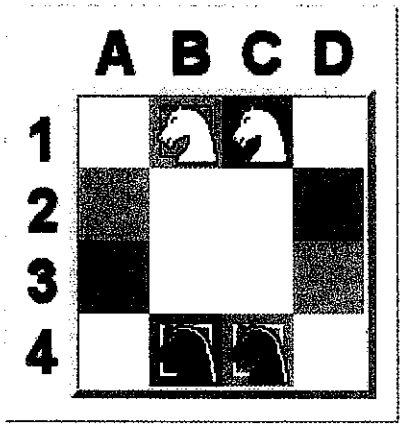
```
Input value: 900
1024

Input value: 16
16

Input value: 4000
4096
```

UCF Local Contest — September 4, 2010

Knights Exchange

filename: `knights`

No programming contest would be complete without a problem involving chess pieces. This is that problem. Jack has been playing a new game recently called Knights Exchange. The premise of the game is very simple. He is given a special chess board and a set of black and white knights, and his goal is to exchange the position of all the black and white knights on the board in as few moves as possible. The chess board he is given is special in that knights may only occupy certain positions on the board. Of course, no spot can ever be occupied by more than one knight, and knights are only allowed to make their normal movements.[1] Consider the following board:



Note the knights on the top row are white and the knights on the bottom row are black. Also notice that on this board, the knights are only allowed to be on the following spots: B1, C1, A2, D2, A3, D3, B4, or C4. Following the rules of the game, the fewest number of moves required to swap all black and white knights is 8 moves, which can be achieved by the sequence C1-D3 => B1-A3 => B4-A2 => C4-D2 => D3-B4 => A3-C4 => A2-C1 => D2-B1.

The Problem:

Given the description of a chess board and where the knights are, determine whether or not it is possible to exchange all black and white knights and, if it is possible, determine the fewest number of moves to do so.

The Input:

Input will begin with a positive integer, T, indicating the number of test cases. Each test case will begin with two space separated integers, R,C ($2 \leq R,C \leq 5$), giving the number of rows and columns in the chess board, respectively. This will be followed by exactly R lines of text, each containing exactly C characters (starting in column 1). Each character will be either 'W', 'B', '.', or 'X' indicating the position is a white or black knight, is open, or is closed, respectively. There

[1] If a knight moves x squares horizontally and y squares vertically, it *must* be true that $x^2 + y^2 = 5$.

will be no more than 15 open positions (including the original knight positions) on any board in the input. Assume that there is at least one black and one white knight and that there are the same number of black/white knights.

The Output:

For each test case, output a single line:

`Case #x: y`

where *x* is the case number beginning with 1, and *y* is the minimum number of moves required to exchange all knights. If it is not possible to exchange the knights on the given board, output -1.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
3
4 4
XWWX
.XX.
.XX.
XBBX
3 3
WWW
...
BBB
3 2
W.
..
.B
```

Sample Output:

```
Case #1: 8

Case #2: 8

Case #3: -1
```

UCF Local Contest — September 4, 2010

g2g c u l8r

filename: `texting`

According to the national statistics, a teenager sends/receives 100+ text messages a day. Dr. Orooji's teenage children are no exception but the problem is Dr. O (an old-fashioned, face-to-face communicator) has difficulty reading text messages full of abbreviations (short-hands) sent to him by his children. Dr. O needs your help reading these text messages.

The Problem:

Given the list of abbreviations and a paragraph, you are to expand the text (paragraph) so that Dr. O can read it easily.

The Input:

The first input line contains an integer, n *(*$1 \leq n \leq 20$*)*, indicating the number of abbreviations. These abbreviations are on the following n input lines, one per line. Each input line starts in column 1 and contains an abbreviation (1-5 characters, consisting of only lowercase letters and/or digits). The abbreviation is followed by exactly one space, and this is followed by the expanded version of the abbreviation (1-50 characters, consisting of only lowercase letters and spaces; assume the expanded version does not start or end with a space and contains no multiple consecutive spaces between words). Assume that all abbreviations are distinct, i.e., no duplicates.

The list of abbreviations is followed by a positive integer, p, indicating the number of input lines containing the paragraph to be expanded. The paragraph is on the following p input lines. Assume these input lines do not exceed column 50, do not start or end with a space, and each line contains at least one word. The paragraph will contain only lowercase letters, digits, and spaces. Assume that there will not be multiple consecutive spaces in the input paragraph.

A word is defined as a consecutive sequence of letters/digits. Assume that a word will be entirely on one input line, i.e., a word is not broken over two or more lines.

The Output:

Each line of the input paragraph must be on one line of output. The input line must be printed in the output exactly the same (spacing). The only exception is that each abbreviation must be replaced by its expanded version, i.e., when an abbreviation is found in the input, its expanded version must be output.

Note that an abbreviation must match a word completely and not just part of a word. For example, if u is an abbreviation for "you", then u must appear as a word by itself in the paragraph in order to be replaced, i.e., if the abbreviation is part of a word in the paragraph (e.g., the paragraph contains the word buy or ugly or you), the u in these words should not be replaced.

Sample Input:

```
8
g2g got to go
g good
c see
l8 late
l8r later
d i am done
u you
r are
6
hi
how r u
you tell me
you are l8
d
c u l8r
```

Sample Output:

```
hi
how are you
you tell me
you are late
i am done
see you later
```

UCF Local Contest — September 4, 2010

Tip to be Palindrome

filename: `tipit`

One of the cool UCF CS alumni is Dr. Greg, The Palindrome Tipper. A palindrome is a string that reads the same forward and backward, e.g., madam, abba, 3, 44, 525.

One cool thing about Dr. Greg is that he leaves at least 20% tip when he eats out, e.g., if the meal is $30, Dr. Greg leaves $6 (30*.20) for tip. If the tip (20%) is not a whole dollar amount, he rounds up the tip to make it a whole number. For example, if the meal is $12, a 20% tip would be $2.40 (12*0.20) but Dr. Greg would leave $3 for tip.

Another cool thing about Dr. Greg is that he is a palindrome guru. If his total bill (meal plus tip) is not a palindrome, he will increase the total (by adding to the tip) to make the total a palindrome. He will, of course, add the minimum needed to make the total a palindrome.

The Problem:

Given Dr. Greg's meal cost, your program should determine the tip amount for him (according to his rules) and the total bill.

The Input:

The first input line contains a positive integer, *n*, indicating the number of times Dr. Greg ate out. The meal costs are on the following *n* input lines, one per line. Each input will contain an integer between 5 and 10000 (inclusive).

The Output:

At the beginning of each test case, output "`Input cost: c`" where `c` is the input cost. Then, on the next output line, print the tip amount and the total bill, separated by one space. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
2
12
84
```

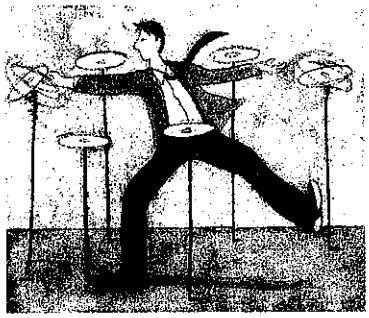
Sample Output:

```
Input cost: 12
10 22

Input cost: 84
17 101
```

UCF Local Contest — September 4, 2010

Plate Spinning

filename: `plate`



(Picture from Firefighternation.com)

Plate spinning is a circus act where a person spins various objects (usually plates and bowls) on poles without them falling off. It involves spinning an object and then returning back to the object in order to add additional speed to prevent it from falling off the pole.

In this problem you will simulate plate spinning where the plates are placed in a circular arrangement (much like the picture to the right). You must determine whether Chester the Clown will be able to maintain the plates spinning or whether one or more plates will end up falling off poles.

The Problem:

Given the number of poles/plates in a circular arrangement and the speed up to which Chester the Clown spins the plates (in degrees per second), determine if he can maintain the act or if plates will fall. For this problem, we will assume that plates degrade (slow down) at a constant rate of 5-degrees-per-second per second and that Chester can move from one pole to any other pole in 0.5 seconds. In addition, assume that Chester can spin up a plate with zero time cost.

A plate falls off when its rate is zero. However, if Chester arrives at a plate exactly at the same time the rate reaches zero, Chester will spin the plate and prevents it from falling, i.e., the rate must reach zero before Chester arrives for the plate to fall.

The Input:

The first line of the input will be a single positive integer, a, representing the number of acts to evaluate. Each of the following a lines will represent a single act and will contain two positive integers, n and p, separated by a single space, where n represents the number of poles ($1 \leq n \leq 15$) and p represents the speed up to which Chester spins a plate ($0 < p \leq 100$) in degrees per second. At the very beginning of each act, all plates are initially spinning at this speed, and he is currently at a plate in the circle (he can choose which plate to start at in order to maximize his chance of success).

The Output:

For each circus act, output a header "Circus Act *i*:" on a line by itself where *i* is the number of the act (starting with 1). Then, on the next line, output "Chester can do it!" if Chester can maintain the act, or output "Chester will fail!" if one or more plates will fall.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
3
2 10
5 7
2 12
```

Sample Output:

```
Circus Act 1:
Chester can do it!

Circus Act 2:
Chester will fail!

Circus Act 3:
Chester can do it!
```

UCF Local Contest — September 4, 2010

The Eternal Quest for Caffeine

filename: `soda`

Like most engineering students, Brandon relies on caffeine, soda in particular, to get him through long nights in the lab. He's not too picky about the brand; as long as it's fizzy, loaded with caffeine, and not the "diet" version, he's happy. When the new Harris Engineering Center opened on the UCF Campus, he was pleased to see a soda machine on every floor. Not just any soda machine, either. These machines are the fancy kind that display all of the soda stock arranged in rows stacked on top of each other (like most snack machines). When a soda is purchased, a conveyor belt rises to the correct row, where the soda is surgically picked up by a robotic clasp that can travel left and right on the conveyor. The conveyor then descends to the vending slot, where the clasp gently deposits the soda. Finally, the slot unlocks and tilts forward, allowing the buyer to retrieve his or her soda. Engineering perfection! And as a bonus, the soda isn't subjected to all the usual mechanical clatter that causes it to fizz over when it's opened.

Unfortunately, these elaborate machines seem to have a propensity for component failure. On one soda mission from his lab, Brandon discovered that the vending slot was broken on the machine on his floor, which prevented it from working altogether. He went to the next floor down and saw that that machine's vending slot was fine, but the conveyor was broken. He went down to the ground floor and saw that that machine was in perfect order, but only had caffeine free diet soda! At this point, Brandon devised a plan. It's a simple matter for him to open up the working machine and harvest the parts he needs for the machine upstairs, then to hike back upstairs and repair the machine that houses the soda he needs. Sure, he *could* just take the soda he wants while the machine is open, but what fun would that be?

The one issue with this plan is that while Brandon does enjoy the engineering challenge, he hates the walking between various broken machines each time he goes to get a coke, so he's asked you, a computer science student and fellow resident of the Harris Engineering Center to help. He can devise a way to monitor each machine in the building and ascertain what parts are working. He needs you to write a program that will allow him to get all the parts he needs from the various machines in the building, traveling up and down as few flights of stairs as possible (he doesn't trust the elevators because he's never been allowed to see how they work). Assume he can carry an unlimited number of parts. He also wants this algorithm to work for various kinds of coke machines and various buildings, in case the vendors decide to change out their machines one day, or the administration decides to relocate the EECS department again (you still can assume that there will always be exactly one coke machine on each floor).

The Problem:

Given the number of floors in the building, the number of parts required for a working machine, a description of the working parts in each machine in the building, and whether or not each machine has the desired kind of soda, determine the smallest number of floor changes required to assemble a working machine that is already stocked with the desired soda. Brandon will always start from his lab and return there after getting his soda.

The Input:

There will be multiple soda machine arrangements to process. Input will begin with three integers, *N*, *F*, and *P* (1 <= *N*,*F*,*P* <= 10), each separated by a single space with no leading or trailing spaces. *N* describes the number of floors in the building, *F* indicates which floor Brandon's lab is on, and *P* indicates the number of different parts in each of the building's soda machines.

On the next *N* lines will be a set of integers followed by a single letter. Each line describes the soda machine on one floor (starting with the ground floor, and proceeding upward in order). The characters on a line are separated by a single space, with no leading or trailing spaces. The first integers on each line will be *S* (0 <= *S* <= *P*), indicating the number of working parts in the machine. *S* integers will follow, each indicating a working part in the machine (each of these integers will be unique and will be between 1 and *P*). Finally, there will be a single character "Y" or "N", where "Y" indicates that the machine has a kind of soda that Brandon likes, and "N" indicates that it does not.

End of input will be indicated by a value of 0 for *N*, *F*, and *P*. This case should not be processed.

The Output:

For each soda machine arrangement, print the case number (starting with 1) and a single integer indicating the minimum number of times Brandon will have to travel up or down a staircase to collect the parts he needs to repair a soda machine, get a soda that he wants, and return to his lab. If there is no way for Brandon to get a soda, print "`Impossible`" instead of the integer.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
4 2 5
5 1 2 3 4 5 N
4 1 2 3 5 Y
4 1 2 4 5 Y
5 1 2 3 4 5 Y
4 2 6
1 1 Y
2 2 3 Y
3 1 4 5 Y
0 Y
0 0 0
```

Sample Output:

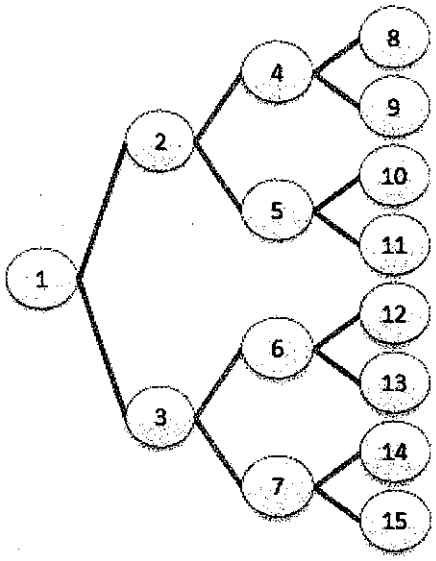
```
Test case #1: 2

Test case #2: Impossible
```

UCF Local Contest — September 4, 2010

Lowest Common Ancestor

filename: `lca`

Perfect binary trees are one of the coolest structures that computer scientists study. They have a lot of properties that make them very nice to work with. One of the nicest properties is that they can just be described by a single integer n giving the depth of the tree. For instance, the perfect binary tree for $n = 3$ looks like:



In general, a perfect binary tree with depth n will have exactly $2^{n+1} - 1$ nodes, and can be numbered by following the pattern seen above (there are of course other ways to number the nodes of the tree, but this is the scheme we will use in this problem).

A common question that arises when dealing with trees is the query of the lowest common ancestor (commonly called LCA) of two nodes in the tree. Formally, the LCA of x and y is the node z of greatest depth in the tree such that z is an ancestor of x and y. Node a is an ancestor of node c if c exists in the sub-tree rooted at node a. Notice that 1 is trivially a common ancestor of any two nodes in the tree, but is not always the *lowest* common ancestor. For instance, the common ancestors of nodes 7 and 12 are 1 and 3, and 3 is the LCA since it is the node of greatest depth. The LCA of 2 and 13 is node 1, and the LCA of 5 and 11 is node 5. The definition of LCA guarantees that the LCA of any two nodes will always be unique.

The Problem:

Given two nodes in the tree using the numbering scheme shown above, determine the LCA of the two nodes.

The Input:

Input will begin with a positive integer, $T \leq 2 \cdot 10^6$, indicating the number of test cases. This will be followed by T test cases, each on a separate input line. Each test case will contain two space separated integers, X and Y, represented in hexadecimal. X and Y will each contain at most 1000 characters from the set {0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f}, where a-f represent 10-15, respectively. You are to determine the LCA of X and Y.
Note: The hexadecimal (base 16) number $d_n d_{n-1} \cdots d_1 d_0$ is converted to a decimal number (base 10) by the following formula: $d_0 \cdot 16^0 + d_1 \cdot 16^1 + \cdots + d_{n-1} \cdot 16^{n-1} + d_n \cdot 16^n$.

The Output:

For each case, output a single line:

```
Case #x: y
```

where x is the case number beginning with 1, and y is the LCA in hexadecimal with no leading 0's. Leave a blank line after the output for each test case. Follow the format illustrated in the Sample Output.

Sample Input:

```
7
7 c
2 d
b 5
10 11
a020fac a030ccf
12afcdb 12afcdc
100000000 ffffffff
```

Sample Output:

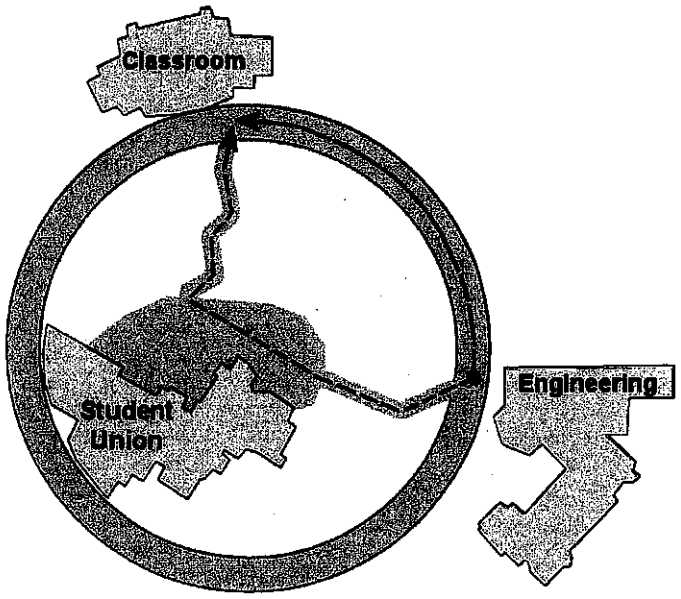
```
Case #1: 3

Case #2: 1

Case #3: 5

Case #4: 8

Case #5: 501

Case #6: 255f9b

Case #7: 1
```

UCF Local Contest — September 4, 2010

Pegasus Circle Shortcut

filename: `shortcut`

For the 2010 High School Programming Tournament, the judges were located in the Engineering building, and most of the teams were in the Classroom building, which is on the other side of Pegasus Circle.



Chris was walking to the Classroom building for the first time, and was joined by Jeremy, who had made the hike a couple of times already.

> "Jeremy, is it faster to stay on the circle, or to cut through the middle using the boardwalks that go to the Student Union?" asked Chris.
>
> "I don't know." Jeremy answered. "I think it's about the same, but it might be slightly faster to use the walkways."
>
> "Well, if it's about the same, let's stick to the circle. I don't want to be attacked by squirrels."*

The Problem:

Given two points on a circle, and two paths to get from one to the other—one following the perimeter of the circle, and the other by a sequence of connected straight line segments through the interior of the circle—determine the shorter of the two paths.

---

* No guarantee is made that these are exact quotes. Also, Chris is not really afraid of squirrels.

The Input:

The input will contain multiple test cases, each consisting of two lines. The first line of each test case contains six floating-point numbers: x_c, y_c, x_s, y_s, x_f, and y_f, where (x_c, y_c) is the center point of the circle, (x_s, y_s) is the start point for both paths (e.g., the Engineering building), and (x_f, y_f) is the finish point for both paths (e.g., the Classroom building). The circle will always have a radius greater than 1, and the start and finish points are both guaranteed to be at distinct points on its perimeter, with an accuracy of at least 3 places after the decimal. The path along the perimeter is always in the direction counter-clockwise around the circle.

The second line of each test case will start with an integer, n ($1 \leq n \leq 10$), followed by n pairs of floating-point numbers, x_1, y_1, x_2, y_2, ... x_n, and y_n, where each pair (x_i, y_i) is a point inside the circle. The interior path traveled will be from point (x_s, y_s) to point (x_1, y_1), then from (x_1, y_1) to (x_2, y_2), then from (x_2, y_2) to (x_3, y_3), ..., then from (x_n, y_n) to (x_f, y_f).

The last test case will be followed by a line containing six zeros. All numbers on an input line will be separated from each other by one space, with no extra spaces at the beginning or end of lines. Assume that all the input floating point numbers will be less than 1000.0 and greater than -1000.0, with at most 6 places after the decimal.

The Output:

For each test case in the input, output a line in either the format
`Case #n: Stick to the Circle.`
if the perimeter path is shorter, or
`Case #n: Watch out for squirrels!`
if the interior path is shorter, where n is the number of the input test case, starting at 1.
Assume that the two paths will not be equal, i.e., it is guaranteed that the two distances will not be equal. In particular, assume that the two paths will differ in length by 0.001 or more.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
5.0 5.0 10.0 5.0 5.0 10.0
6 8.5 4.7 6.9 5.0 3.6 6.5 4.2 7.1 4.2 8.3 4.7 8.8
2.0 8.0 0.5 16.87412 7.5 0.8761
2 3.25 9.25 7.0 7.0
0 0 0 0 0 0
```

Sample Output:

```
Case #1: Stick to the Circle.

Case #2: Watch out for squirrels!
```

UCF Local Contest — September 4, 2010

Soccer Standings

filename: `soccer`

Soccer fever has gripped the world once again, and millions of people from dozens of countries will be glued to their TV sets for the World Cup. Being an enterprising sort, you've started up your own internet World Cup Soccer Channel for streaming matches online. Recently you came up with the idea of filling up the time between matches by having a couple of 'experts' offer critical analysis of games. For this purpose, you have devised a unique ranking system for soccer teams, which you must now implement.

The Problem:

Given a list of teams and a list of match scores, you must compute several quantities for each team. These are: the total number of goals scored over all their games, the total number of goals scored against them (goals allowed, for short), the number of wins, draws and losses, and the number of points scored so far. Points are to be computed as follows: winning a match nets a team 3 points, losing gets them nothing. In the event of a tie, both teams get 1 point.

In addition to this, you must order the teams correctly according to your new system. Teams are ordered according to points, from highest to lowest. In the event of a tie in points, the team that has a higher *goal difference* comes first. The goal difference is defined as the total number of goals scored by the team minus the total number of goals scored against them.

If there is still a tie (i.e., two or more teams have the same points and the same goal differences), the team with higher total goals scored comes first. If even this is tied, the team whose name comes first in alphabetical order goes first.

The Input:

The first input line contains a positive integer, n, indicating the number of data sets to be processed. The first line of each data set consists of two positive integers T ($T \leq 6$) and G ($G \leq 21$) – the number of teams in this group and the total number of games played by them. The next line contains T unique names separated by single spaces. Each name is a single uppercase word with no more than 15 characters.

Each of the next G input lines will contain the results of a match. Each line is of the form *<country_1> <score_1> <country_2> <score_2>*. For example, "Greece 2 Nigeria 1" indicates that Greece and Nigeria played a game with score 2-1. All four terms will be separated by single spaces.

The Output:

At the beginning of output for each data set, output "`Group g:`" where g is the data set number, starting from 1. Next you should print a single line for each team, ordering teams as

mentioned above. For each team, the line you print should be of the form "*<name> <points> <wins> <losses> <draws> <goals scored> <goals allowed>*". These items should be separated by single spaces.

Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

Sample Input:

```
2
2 1
Kasnia Latveria
Kasnia 0 Latveria 1
4 6
ENGLAND USA ALGERIA SLOVENIA
ENGLAND 1 USA 1
ALGERIA 0 SLOVENIA 1
SLOVENIA 2 USA 2
ENGLAND 0 ALGERIA 0
SLOVENIA 0 ENGLAND 1
USA 1 ALGERIA 0
```

Sample Output:

```
Group 1:
Latveria 3 1 0 0 1 0
Kasnia 0 0 1 0 0 1

Group 2:
USA 5 1 0 2 4 3
ENGLAND 5 1 0 2 2 1
SLOVENIA 4 1 1 1 3 3
ALGERIA 1 0 2 1 0 2
```

UCF Local Contest — September 4, 2010

Interstellar Love

filename: `stars`

After two years of sharing a bedroom with you in a college dorm, Jeff finally has his own room. Excited about inviting girls over to his room, he ponders over what decorations the fairer sex will enjoy.[1] He decides upon setting up a "fake" planetarium with a black ceiling and glow in the dark stars that form constellations. Unfortunately, in his haste, he has made several "errors" in setting up his constellations. See, everyone knows that constellations don't have cycles in them. Instead, whenever we visually connect the stars together with lines, a tree is always formed. (A cycle is formed when you can start at a star and, using connections, go to one or more other stars and then end up at the original star.)

Since you were Jeff's roommate for two years, you figure you'll help him fix his constellations. Your job will be twofold: to count the number of constellations Jeff has, and to report how many of them have cycles and need to be fixed. A constellation consists of multiple stars that are all connected to one another (directly or indirectly). A constellation that needs fixing is simply one that has a cycle.

The Problem:

Given several configurations of stars and connections between stars, determine how many constellations are defined in each configuration and how many need fixing.

The Input:

The first input line contains a positive integer, n *($n \leq 100$)*, indicating the number of night skies to consider. The first line of each night sky contains two positive integers, s *($s \leq 1000$)*, representing the number of stars for this night sky, and c *($c \leq 10000$)*, representing the total number of connections between pairs of stars for this night sky. Each of the following c input lines contains two distinct positive integers representing a single connection between two stars. The stars in each test case will be numbered 1 through s, inclusive. A connection is considered bidirectional, thus, if a is connected to b, b is connected to a. Assume that all connections in a data set are distinct, i.e., no duplicates. Also assume that there will never be a connection from a star to itself.

[1] This is based on a true story. The person who is the inspiration for this story did, in fact, major in Planetary Science and make his room's ceiling a relatively accurate depiction of the night sky, as seen in Boston circa 1995.

The Output:

For each test case, just output a line with the following format:

`Night sky #`*k*`: `*X*` constellations, of which `*Y*` need to be fixed.`

where *k* is the number of the night sky, starting at 1, *X* is the total number of constellations described in that night sky, and *Y* is how many of those constellations contain a cycle.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
3
5 4
1 2
1 3
2 3
4 5
8 5
1 2
3 4
6 7
6 8
8 7
3 2
1 2
1 3
```

Sample Output:

```
Night sky #1: 2 constellations, of which 1 need to be fixed.

Night sky #2: 3 constellations, of which 1 need to be fixed.

Night sky #3: 1 constellations, of which 0 need to be fixed.
```

Note: In the second example, star number 5 is not connected to any other stars. This star on its own is NOT counted as a constellation, leaving only {1,2}, {3,4} and {6,7,8} as constellations, of which only the last one needs to be fixed.

UCF Local Contest — September 4, 2010

Dr. Orooji's Exam

filename: `exam`

Dr. Orooji is normally a very lax, fun-loving guy. But, when it comes to giving exams, he's very strict. He wants to make sure that no one cheats! After many years of observation, he has realized that cheating is easiest when students are sitting in two adjacent columns. Thus, after students have sat down to take a test, he moves students so that no two columns that are occupied are adjacent to one another. A column in a classroom is considered occupied if one or more students are sitting in that column. Consider the following initial arrangement of students taking one of Dr. O's exams:

	Column 1	Column 2	Column 3	Column 4	Column 5
Row 1	X				
Row 2		X	X		X
Row 3	X		X		
Row 4	X		X		X

Note: The X's mark where the students are initially sitting.

In this situation, Dr. O can't give his exam because there are students in both column #1 and column #2. In order to fix this situation, Dr. O can simply ask the student in column #2 to move to the one open seat in column number #1:

	Column 1	Column 2	Column 3	Column 4	Column 5
Row 1	X				
Row 2	X		X		X
Row 3	X		X		
Row 4	X		X		X

Although this class was very easy to fix, not all cases are as simple as this one. Sometimes, Dr. O wastes precious time asking many students to move. As his independent study student, he has asked you to write a program that figures out the fewest number of moves necessary to get his class ready to take an exam. This way, he won't have to waste any more time moving students than necessary. Note that it does not matter which columns the students will eventually sit in as long as no two adjacent columns are occupied (or partially occupied).

The Problem:

Given the initial arrangement of students sitting for one of Dr. O's exams, determine the fewest number of students that have to move so that no two adjacent columns are both occupied with at least one student.

The Input:

The first input line contains a positive integer, n *(*$n \leq 100$*)*, indicating the number of seating arrangements to consider. The first line of each seating arrangement contains two positive integers, r *(*$r \leq 100$*)*, representing the number of rows in the classroom and c *(*$c \leq 100$*)*, representing the number of columns in the classroom. The following r lines contain information about the classroom. The first line represents the students in the first row, the second line represents students in the second row, etc. Each of these lines will have a string of exactly c characters, all either 'X' or 'O', representing the students sitting on that row. In particular, 'X' represents a student while 'O' represents an empty seat. The first character in each string is the location in column 1, the second character in each string is the location in column 2, etc. These input lines start in column one and contain no spaces anywhere.

The Output:

For each test case, just output a line with the following format:

`Seating arrangement #`*k*`: At least` *X* `student(s) need to be moved.`

where k is the number of the seating arrangement, starting at 1 and X is the fewest number of students that need to be moved so that no two adjacent columns both have students.

If it is impossible to move students to create this type of arrangement, output a line with the following format:

`Seating arrangement #`*k*`: Can not be fixed.`

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
3
4 5
XOOOO
OXXOX
XOXOO
XOXOX
2 3
XXX
XOX
4 7
OXOOOOO
OXOOXOO
OOOOOOX
OOOOOOX
```

Sample Output:

```
Seating arrangement #1: At least 1 student(s) need to be moved.

Seating arrangement #2: Can not be fixed.

Seating arrangement #3: At least 0 student(s) need to be moved.
```

UCF Local Contest — September 4, 2010

The Puzzle Master

filename: `maze`

Will Shortz loves puzzles. In fact, he loves them so much he majored in enigmatology, the study of puzzles, at Indiana University.[1] Will loves adding new puzzles to his collection, and has recently decided that he is going to start including mazes in the *New York Times* puzzle section.

However, mazes are somewhat tedious to make by hand. Will does not have strict criteria about the difficulty of the maze (he will determine that later), and instead only requires they look somewhat random. To this end, you have come up with the following scheme for generating mazes. You will start with a grid where all walls are drawn (except the breaks shown in the top left and bottom right), and pick a starting cell and direction (e.g., a 6x4 grid with starting spot 4,2 and starting direction east, as seen in Figure 1). You then 'walk' the given direction from the current cell (if possible), and 'break' any walls you walk through. Finally, change the direction to the next cardinal direction, moving clockwise (after east, turn south). After the first move, the maze will look like the maze seen in Figure 2.

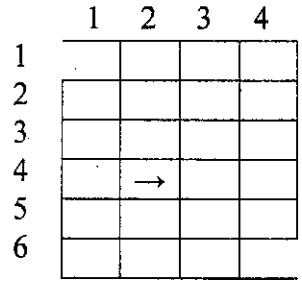


Figure 1

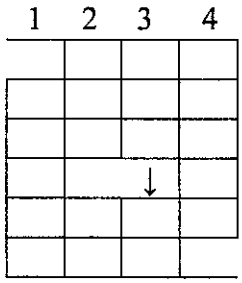


Figure 2

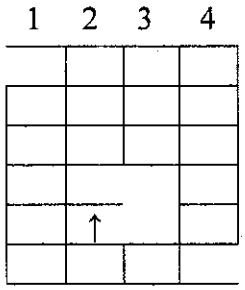


Figure 3

You continue moving in this fashion, making possible moves and breaking walls, so long as the move you would make is possible. A move is considered *impossible* if a) it takes you off the board, b) it creates a cycle in the maze, or c) there is no wall to break by going this direction. After making 2 more moves, you will end up in the position shown in Figure 3. At this point, you can't move north, since that would create a cycle in the maze. So you go to the next direction, east. However, you can't move east either because there is no wall to your east. Again, you go to the next direction, south, and make this move since it is legal.

Eventually you will get to a spot where you can't make any moves since every move would violate some rule. At this point, you should go back to the spot you came from and try the next direction *from that spot*. For example, if the last direction you moved at a spot was east and then you backed up to the spot again, then the next direction to try would be south, regardless of what directions you traveled in the spots after originally going east.

---

[1] This is actually a true story. He is the only person in the world to hold this degree, and he is now the editor of the *New York Times* crossword puzzle.

Can you help Will by writing a program to generate mazes using your scheme?

The Problem:

Given the size of the puzzle you want to create and the starting spot and direction, generate and output a maze that Will can use for the puzzle section. Because this is going to be used every day, your program should be able to generate multiple puzzles.

The Input:

Input will begin with a positive integer, T, indicating the number of test cases. This will be followed by exactly T test cases. Each test case will begin with a line containing two space separated positive integers, R and C ($1 \leq R,C \leq 50$), giving the number of rows and columns in the maze, respectively. This will be followed by a line containing two space-separated positive integers, SR and SC ($1 \leq SR \leq R$, $1 \leq SC \leq C$), giving the start row and start column, respectively. Finally, there will be a line containing only a single character 'N', 'E', 'S', or 'W', indicating a starting direction of north, east, south, or west, respectively.

The Output:

For each generated maze, first output a single header line:

```
Maze #x:
```

where x is the maze number beginning with 1. Follow this with a blank line, and then print the maze on the next $2*R+1$ lines. Your output maze must exactly follow the format of the sample output, using only the characters '+' (plus), '-' (minus), '|' (vertical bar / pipe), and ' ' (space). Each of the $2*R+1$ lines should contain exactly $2*C+1$ characters.

Follow each maze with a single blank line.

[*Sample Input/Output on next page*]

Sample Input:

```
3
6 4
4 2
E
1 5
1 3
N
3 3
1 1
W
```

Sample Output:

```
Maze #1:

+-+-+-+-+
        |
+ +-+-+ +
| |   | |
+-+ + + +
|   |   |
+ +-+-+ +
| |   | |
+ +-+ + +
| |   | |
+ + +-+ +
|   |
+-+-+-+-+

Maze #2:

+-+-+-+-+-+

+-+-+-+-+-+

Maze #3:

+-+-+-+
    | |
+-+ + +
|   | |
+ +-+ +
|
+-+-+-+
```