Twenty-fifth Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
The Stapler!	stapler
Programming Logo	logo
Starter's Game	starter
Ice Mice	mice
Base-ic Genetics	genetics
Digit Sums	sums
The Floor is Lava	lava
Tag Cloud	cloud
Circle Gets the Square	circle
Tray Stacker	tray
Bitland Tower Challenge	tower

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

For example, if you are solving Ice Mice:
Call your program file: mice.c, mice.cpp or mice.java
Call your input file: mice.in
Call your Java class: mice

The Stapler!

Filename: `stapler`

Chuck Garrett is a mild-mannered programmer by day. However, by night, he dawns his disguise, and becomes….The Stapler! Yes, that's right…The Stapler. And while his name may seem odd, his goal is just, and, actually, his name is quite fitting! You see, The Stapler aims to address all those evil piles of paper in the world by stapling them together! Then, and only then, will a pile stay together! The Stapler's arch-nemesis, known only as the Jaji Mkuu, loves to leave piles unfastened, allowing pages within a pile to become separated from each other. While The Stapler does constant battle with the Jaji Mkuu, his #1-prized weapon (his miniature stapler) can only fasten at most 30 pages at a time.

The Problem:

Given multiple piles of paper, sum the quantities of pages in each pile up and determine if The Stapler can fasten each pile.

The Input:

Input will begin with a single, positive integer, n, on a line by itself, representing the number of piles from the Jaji Mkuu that The Stapler must face. Following this will be n battles. Each battle begins with a single, positive integer, p ($1 \leq p \leq 1{,}000{,}000$), on a line by itself. On the next line there will be p positive integers, each separated by a single space, representing a quantity of pages in that pile of paper. You will be guaranteed that the sum of all quantities of pages within each pile will fit within an integer.

The Output:

For each battle, begin with a header "`Battle #i:`" on a line by itself (where i is the number of the battle, beginning with 1). Then, output a line that says "`The Stapler must face t pages`" where t is the grand total of all quantities of pages within the Jaji Mkuu's pile with which The Stapler had to contend. Then, on the next line, if The Stapler makes it through the battle (i.e., the pile is not too thick for his miniature stapler), output "`The Stapler saves the day!`"; otherwise, output "`Foiled again!`" Leave a blank line after the output for each battle.

(Sample Input & Sample Output are on following page)

Sample Input:

```
2
3
3 5 2
5
12 15 3 10 2
```

Sample Output:

```
Battle #1:
The Stapler must face 10 pages
The Stapler saves the day!

Battle #2:
The Stapler must face 42 pages
Foiled again!
```

Programming Logo

Filename: `logo`

Logo is an old programming language with a turtle that holds a pen. The commands in the language allow the turtle to move in the direction he is positioned. If the pen is down, then a line is drawn in his path of movement. The UCF Programming Team has decided that it wants to resurrect Logo and market the new tool to middle schools in the hope of inspiring the next wave of UCF Programming Team Champions.

In NewLogo, the turtle draws on a grid, 20 characters high and 30 characters wide with the top left corner with the coordinates (0,0) and the bottom right corner with the coordinates (19, 29). Each grid square will contain a single text character. At the beginning of any NewLogo program, the grid contains all blank squares and the turtle is located at (0,0) headed right with his pen down. In addition to being able to put his pen up and down, the turtle can change the character with which his pen draws (his starting character is '*'). This will allow the display to have more variety than before. Finally, the turtle can turn right or left in increments of 45 degrees, so it is possible to move diagonally. Here are the NewLogo commands:

```
START
PEN UP
PEN DOWN
CHANGE CHARACTER C
MOVE X
RIGHT TURN R
LEFT TURN L
END
```

The START command simply indicates the start of a NewLogo program while the END command indicates the end of a NewLogo program. Both PEN commands do not move the turtle at all, but simply change the state of the turtle's pen to either UP (not writing) or DOWN (writing) as indicated. The CHANGE CHARACTER command does not move the turtle at all or draw anything. It simply changes the character with which the pen will draw (if it is down) on subsequent commands after this command is executed.

The MOVE command simply moves the turtle from its current spot *X* number of steps. If the pen is down, then this character is drawn in each grid square on the path. The only square for which this isn't necessarily true is the initial square of the path. If this initial square was previously drawn on, then we *don't* draw over it with the potentially new character. If this initial square was previously blank, then we *do* draw over the square with the new character. Thus, if the first line after START in a program is MOVE 5, then 6 stars are drawn. However, if this is followed by MOVE 3, only 3 more stars will be drawn. The RIGHT TURN and LEFT TURN commands turns the turtle, in place, *R* degrees to the right or *L* degrees to the left, respectively (*R* and *L* are given in multiples of 45 degrees).

If a NewLogo program commands the turtle to go off the 20 x 30 grid, the program has caused a TURTLE OUT OF BOUNDS ERROR.

The Problem:

For each NewLogo Program, produce the output of the program. You are guaranteed that these programs are syntactically correct. Namely, only the commands listed above will be used, the angles for the turns will be positive integers that are multiples of 45, the values for movement will be positive integers and the character for the change character command will always be a non-whitespace printable character (these are characters with ASCII values in between 33 and 126, inclusive). If the program produces a TURTLE OUT OF BOUNDS ERROR, then simply display this error message instead.

The Input:

The first line of the input file will contain a single positive integer, *n (n < 100)*, designating the number of NewLogo programs your program will have to interpret. All of the programs follow, with one command per line. You are guaranteed that the first line of each program is the command "START" and that the last line of each program is the command "END". On each line of code, all words and numbers will be separated by a single space from each other. No extra spaces will be included at the beginning or end of any lines.

The Output:

The first line of output for each program should have the following format:

```
Program #k
```

where *k* represents the program number, starting at 1.

If there is no error in the program, then output the entire grid. In order for the output of a program to be accurate, there must be spaces (not tabs or other characters) in each spot where no output was written during the course of the program. In addition, place the grid inside of a rectangle of +, - and | characters where - makes up the horizontal line, | makes up the vertical line and + represents the corners (see Sample Output). Each grid will be output as 22 lines of 32 characters each (which is the 20 by 30 grid surrounded by the rectangle).

If there is a TURTLE OUT OF BOUNDS error in the program, simply identify the line number of the first time the error occurs and output a single line with the following format:

```
LINE x: TURTLE OUT OF BOUNDS ERROR
```

where *x* represents the line number of the first line of code in the program that causes a TURTLE OUT OF BOUNDS ERROR. The START line is line number 0. All subsequent lines are numbered in regular counting order.

Skip a blank line in between the output for each program.

Sample Input:

```
2
START
MOVE 5
RIGHT TURN 90
CHANGE CHARACTER &
MOVE 4
RIGHT TURN 45
CHANGE CHARACTER @
MOVE 3
END
START
MOVE 25
RIGHT TURN 90
MOVE 5
LEFT TURN 90
MOVE 6
RIGHT TURN 45
END
```

Sample Output:

```
Program #1
+------------------------------+
|******                        |
|       &                      |
|       &                      |
|       &                      |
|       &                      |
|      @                       |
|     @                        |
|    @                         |
|                              |
|                              |
|                              |
|                              |
|                              |
|                              |
|                              |
|                              |
|                              |
|                              |
|                              |
|                              |
+------------------------------+

Program #2
LINE 5: TURTLE OUT OF BOUNDS ERROR
```

Starter's Game

Filename: `starter`

"Tango at 9 o'clock!", Lima shouted. Starter whipped around with pin-point precision and snapped off two wifflelasers. Much to his dismay, the "tango" had actually been a few meters to the left of his "9", and so his shots missed entirely. Even more to his dismay, the tango had now spotted him.

"Fart-smeller at 3:05!" Starter heard the enemy shout a moment before his vorpotron endosuit froze up from incoming fire. It wasn't long before the rest of his squad was hunted down and frozen up as well, and not much longer still, before the lights in the Conflict Room came on and the battle was over.

"Looks like Tilapia Army... has gone bottom-up!"

"Shut up, Horatio!" Starter yelled, as he ordered you and the rest of Tilapia Army into the hall. You now find yourself quietly standing in line, awaiting your commander's next orders.

"Alright, listen up," begins Starter. "Rough estimates on tango positions aren't going to cut it anymore- you got that, Lima? We need accuracy, we need precision. We need... a revolution! Here's the plan. Lima, you'll continue calling out the enemy's angle relative to me, still encoded as the time on a clock, only you'll do it to the nearest second!"

"But Starter, that's insane-" Lima begins to protest.

"There's more! Remember Lima, if you want to win, you have to change the game. Not only will you report their angle, but you'll also report their distance! You'll encode it as a date, and round it to the nearest day! Do you understand? With this amount of precision, Tilapia Army will be subject to Horatio's terrible one-liners no more!"

"Alright Starter, but it's going to take some cunning on my part…" Lima thinks for a moment, then turns to you, the only cadet who happened to bring his powerdesk to the fight (What's up with that, anyway? What are you, some kind of nerd that bottles and labels his own farts?). He begins whispering, "Listen launchy, my plumbomatic datascrobblizer can give me the relative coordinates of any position just by pointing at it. You'll need to write a program that can take a point and output its angle and distance from the origin. I'll upload the coordinates to your powerdesk, and your program will tell me what to call out to Starter! But not a word of this to anyone, you hear?"

"And Lima?" adds Starter.

"Yeah?"

"Don't forget that here in Conflict School we use the intergalactic 360 day space calendar, in which there are 30 days to a month and 12 months to a year."

"What do I look like, some kind of boogerbutt?" responds Lima, as he turns and whispers to you, "Did you know that?"

The Problem:

Lima needs you to write a program that, given the coordinates of an enemy "tango", determines its angle and distance from the origin and encodes these values as a time and date. The encodings should be rounded to the nearest second and day, respectively.

The Input:

Your program will need to locate many "tangoes" in the coming battle. The first line of the input will be a positive integer, *t*, the number of tangoes to process. The next *t* lines of the input will each contain two integers, *x* and *y* (-5000 ≤ *x*, *y* ≤ 5000), separated by a single space representing an enemy's location relative to Starter in meters. Note that *x* and *y* will never both be 0.

The Output:

For each enemy Lima spots, output the line "Tango at *H:mm:SS, MM/DD/YYYY*!", where *H:mm:SS* is the target's orientation relative to Starter encoded as a time (rounded to the nearest second), and *MM/DD/YYYY* is the target's distance relative to Starter, rounded to the nearest day, where one meter is equal to 500 days. As an example of rounding, a value of 0.49 would be rounded down to 0, while a value of 0.5 would be rounded up to 1. You may assume Starter is facing in the positive y-direction, so an enemy on the y-axis in front of Starter is at 12:00:00. Similarly, Starter's immediate right (+*x*) is 3:00:00, directly behind Starter (-*y*) is 6:00:00, and Starter's immediate left (-*x*) is 9:00:00 (see the diagram below). Be sure to include leading zeroes for all seconds, minutes, months, days and years, but not for hours. Hours, months, and days should be numbered beginning at 1. Years, minutes, and seconds should be numbered beginning at 0. Once all tangoes have been located, it is safe to assume that they have also been dispatched, and so you should print the line "The enemy's gate is down!" to end the battle.

Sample Input:

```
7
1 0
1 5
-5000 -5000
-622 533
-2422 -777
1 -5000
1024 1024
```


Sample Output:

```
Tango at 3:00:00, 05/21/0001!
Tango at 12:22:37, 02/01/0007!
Tango at 7:30:00, 12/05/9820!
Tango at 10:21:11, 09/06/1137!
Tango at 8:24:26, 10/02/3532!
Tango at 5:59:59, 06/11/6944!
Tango at 1:30:00, 04/28/2011!
The enemy's gate is down!
```
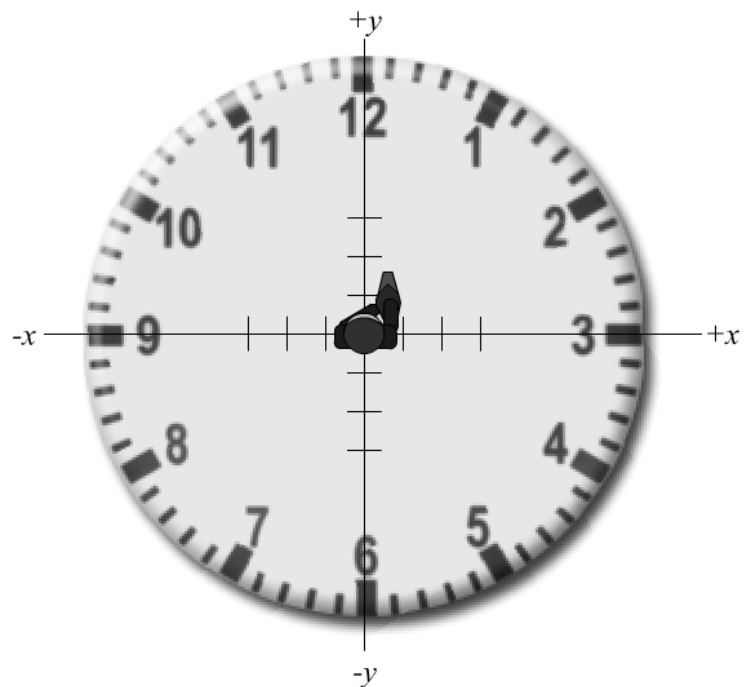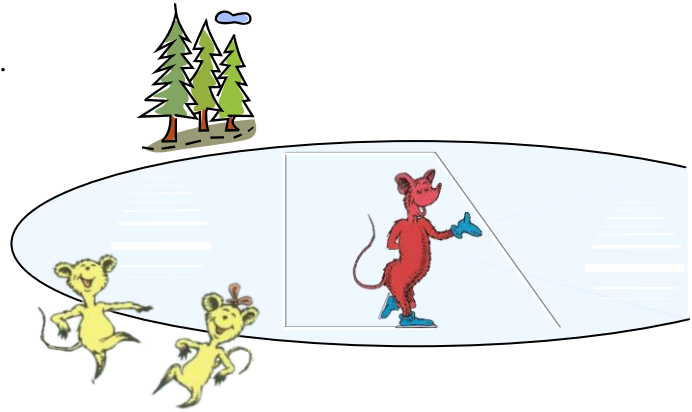
Ice Mice

Filename: `mice`

The circular ponds near the town of Kreiss
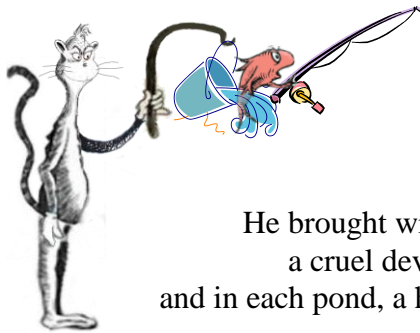host the winter game of some carefree mice.

They skate a rectangle upon the ice
with proportions which
they find quite nice:
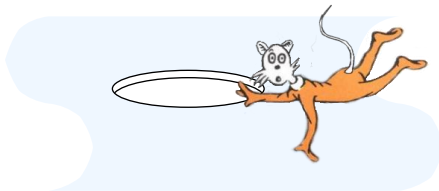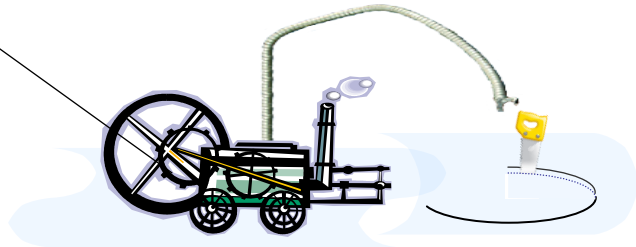
Its length must be the same
as its breadth twice.
No other shape could ever suffice!

A pesky cat, that cat named Brice,
caused some mischief
on the ice.

He brought with him
a cruel device
and in each pond, a hole did slice.

"What's this?" cried the stymied mice.
"How can we skate upon this ice?"
"Help us, please,
detective Phoenix Weisse!"

Sample
Pond
#3

The mouse sleuth studied each pond thrice.
"OBSTRUCTION!" he cried; analysis concise.

"Now, let me give you some advice…"
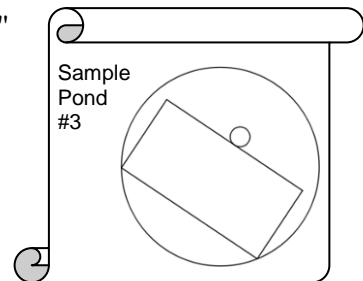
The Problem:

Given a description of a frozen circular pond with a circular hole sliced into its ice, find the area of the largest rectangle that can be used for the mice's game.

9

The Input:

The first line of input will contain only a single, positive integer, n, which represents the number of ponds in the input to evaluate. The next n lines of input will each contain exactly 4 integers, separated by single spaces: r_P, r_H, x and y ($1 \leq r_H < r_P \leq 10000$ and $-10000 \leq x, y \leq 10000$). The radius of the circular pond is r_P. The radius of the sliced circular hole is r_H and its position is the Cartesian coordinate (x, y) with the pond center as the origin. The circular hole will always fit entirely within the pond.

The Output:

For each pond in the input, output a line with the message `"Pond #p:"` where p is the number of the pond, starting from 1. On the next line, output the concise analysis for Phoenix Weisse, followed by a space and then the area of the largest possible rectangle on the sliced pond. If this area is the same as the area of the largest possible rectangle *before Brice sliced the hole*, Weisse's analysis is `"ICE CLEAR!!!"` If the hole causes the area to be different, then Weisse's analysis is `"OBSTRUCTION!"` Round the area to two decimal places for making the comparison and also for output. For example, 0.3450 rounds to 0.35 and 0.3449 rounds to 0.34. Output a blank line after the area of each pond.

Sample Input:

```
3
10 1 0 3
1000 2 0 996
10 1 2 3
```

Sample Output:

```
Pond #1:
OBSTRUCTION! 128.00

Pond #2:
ICE CLEAR!!! 1600000.00

Pond #3:
OBSTRUCTION! 136.22
```

Base-ic Genetics

Filename: `genetics`

The biology lab at UCF has recently been having problems transmitting genetic sequences between computers. Unfortunately, the current system represents the genetic sequences as a string of the characters: 'A', 'C', 'G' and 'T'. In order to speed up the data transfers you have been tasked with compressing the data. Luckily for you, your professor devised an easy way to do this, but he gave you the task of actually implementing it. He said that if you consider each of these letters as a number in base-4, such that 'A' = 0, 'C' = 1, 'G' = 2, and 'T' = 3, then any genetic sequence has a unique base-10 representation. He needs you to implement a program that will convert a string describing a genetic sequence to a base-10 number, or vice-versa (given a base-10 number, output the corresponding genetic sequence).

The Problem:

Given a genetic sequence, output the corresponding base-10 number. Likewise, given a number in base-10, output the corresponding genetic sequence.

The Input:

The first line of the input will contain a positive integer, q, representing the number of sequences to convert. The next q lines will contain either an integer, n ($1 \le n \le 2^{30}$-1), which represents a base-10 number and is guaranteed to have no leading zeroes, or a genetic sequence (represented by a string of the characters 'A','C','G' and/or 'T', and guaranteed to have no leading A's), between 1 and 15 characters in length.

The Output:

For each sequence, output the header "`Sequence #j: `" where j represents the sequence being processed (beginning with 1). Following the header, output either the corresponding genetic sequence for the integer given, or the corresponding base-10 number for the genetic sequence. The output for each sequence should each be on its own output line.

Sample Input:

```
3
CAACAGAT
16675
GAGGTATAC
```

Sample Output:

```
Sequence #1: 16675
Sequence #2: CAACAGAT
Sequence #3: 142129
```

Digit Sums

Filename: `sums`

Tom is very excited about the 25th Anniversary of the UCF High School Programming Tournament, so much so that he has been trying to find different ways of finding the number 25 in other numbers. He especially likes summing up the digits of a number and seeing if the sum is evenly divisible by 25, but this can be difficult to do for very large numbers. He would like you to write a program for him to check if the sum of the digits of a number is evenly divisible by 25.

The Problem:

Given a number find the sum of all of its digits, and determine whether the sum is evenly divisible by 25 or not.

The Input:

The first line of the input file will begin with a single positive integer, *n*, representing the number of integers to check. Each of the next *n* lines will contain a number, *x*, where *x* is a positive integer with at most 9 digits.

The Output:

For each given number print "`Yes, it's 25!!!`" if the sum of the number's digits is divisible by 25 or "`Bummer, no 25.`" if the sum of the digits is not divisible by 25. Print the answer to each case on a separate line.

Sample Input:

```
5
55555
25
239164
123456789
324937886
```

Sample Output:

```
Yes, it's 25!!!
Bummer, no 25.
Yes, it's 25!!!
Bummer, no 25.
Yes, it's 25!!!
```

The Floor is Lava

Filename: `lava`

Roger and Pete are playing a game of "The Floor is Lava" in their living room. The game is played by walking on furniture to avoid touching the "lava" floor, and whoever touches the lava first loses. However, their mom has forbidden them from jumping on the furniture, so Roger and Pete are only allowed to walk onto adjacent furniture (they can move forward, backward, left, or right but not diagonally). To make this easier, they organized the furniture into a grid before they started playing. Unfortunately, in the middle of their game, the floor actually turned into molten lava! Luckily, their mom only buys lava-proof furniture, so the boys can keep playing safely. Roger and Pete want you to figure out how many pieces of furniture they can still reach within a certain number of steps.

The Problem:

Given the layout of the furniture and lava, Roger and Pete's starting location, and the maximum number of steps they can make, find the number of pieces of furniture the boys can reach.

The Input:

Input will begin with a single positive integer, *t*, indicating the number of rooms to analyze. Each room will begin with a line containing three integers, *r*, *c* and *n*, where *r* and *c* represent the number of rows and columns in the furniture grid ($0 < r, c \leq 50$), respectively, and *n* represents the maximum number of steps they can take. Each integer will be separated by a single space. This is followed by exactly *r* lines of exactly *c* characters each, indicating the layout of the floor and the obstacles: '`.`' indicates a piece of furniture, '`L`' indicates lava, and '`S`' indicates Roger and Pete's starting location. They always start on a piece of furniture. '`S`' will occur exactly once in every room's layout, and no characters other than '`.`', '`L`', and '`S`' will occur as part of the layout.

The Output:

Output should be in the following format:

`Room #i: They can reach p pieces of furniture.`

where `i` is the number of the room (starting at 1) and `p` is the number of pieces of furniture Roger and Pete can reach in at most `n` steps from their starting location, including their starting location. If they can only reach one piece of furniture, the output should instead be

`Room #i: They can reach 1 piece of furniture.`

Output for each room should be separated by an empty line.

Sample Input:

```
3
4 4 2
....
..S.
....
....
5 5 7
SL...
.L...
.L...
.L...
.....
3 6 1
LLLLLL
LLL.SL
LLLLLL
```

Sample Output:

```
Room #1: They can reach 11 pieces of furniture.

Room #2: They can reach 9 pieces of furniture.

Room #3: They can reach 2 pieces of furniture.
```

Tag Cloud

Filename: `cloud`

Goofus is working on his blog, a website where he writes articles about his life so others can read them. He wants a way to automatically summarize his articles based on their subject so that users can find the articles they want quickly. Goofus consults his friend Gallant for help. Happy to oblige, Gallant explained the basics of Natural Language Processing, Document Classification, and Information Extraction, all of which went right over Goofus' head.

Gallant then remembered a couple of websites that he had seen with Tag Clouds. Tag Clouds are a visual representation of the document's *word frequencies*, or how often a particular word occurs in the document. Words which occur more frequently are larger than words which occur rarely. Here is an example of what a Tag Cloud can look like:

His site's viewers can quickly determine if the article is relevant to their interest by looking for the largest words. Additionally, Tag Clouds can be quite aesthetically pleasing and attract people to look at articles they might otherwise have skipped. Goofus is ecstatic about the idea, but he's not that good of a programmer. He's hired you to help him make the Tag Cloud for his blog.

The Problem:

You will be given a list of words which makes up the blog article's contents. You will need to count the number of times each words occurs, and output this list of words sorted by the number of times each word occurs in the article.

The Input:

The first line of input contains a positive integer, *t*, where *t* is the number of articles that need tag clouds. Each article starts with an integer, *w* ($1 \leq w \leq 200$), on a line by itself which is the number of words in that article. The following *w* lines will each contain a single word from that article with no leading or trailing spaces. A word is made of up of 1 to 15 (inclusive) lower-case letters. A word will never have capital letters, numbers, whitespace, or any other symbol. You will be guaranteed that no article will contain more than 10 different words, though each word may appear more than once.

The Output:

For each article, you will first print a header "Article #a:" where a is the number of the article being processed (starting with 1). After that, output q lines, where q is the number of unique words in that article. For each line, output the word and the number of times it occurred, separated by a single space. Output the most frequently occuring word first, followed by the other words in descending order of frequency. No two words will have the exact same number of occurrences; the order of the words is unique. Finally, output a blank line after the output for each article.

Sample Input:

```
2
1
test
15
hello
high
school
programming
team
hello
high
school
programming
hello
high
school
hello
high
hello
```

Sample Output:

```
Article #1:
test 1

Article #2:
hello 5
high 4
school 3
programming 2
team 1
```

Circle Gets the Square

Filename: `circle`

Ali played a bunch of games of Tic-Tac-Toe with Mack and Zack. For some reason, he never won the games where he played the 'X', and always won or tied games when he played the 'O'. Maybe it's because he likes circles so much.

Tic-Tac-Toe is played on a 3x3 board which begins with all 9 squares blank. The first player marks an 'X' in any blank square, then the second player marks 'O' in any remaining blank square. The players continue to take turns until one player wins, or until the board is full. The winner of a Tic-Tac-Toe game is the one who first gets 3 of their marks in a row—horizontally, diagonally, or vertically. If there is a tie (no winner), some people call it a "cat's game."

Here are some example games. The gray line shows how that player won.
Note that the last board didn't have a winner; it was a tie.

The Problem:

Given a Tic-Tac-Toe board at the end of the game, determine who (if anybody) won.

The Input:

The first line of input contains a positive integer, *t*, where *t* is the number of Tic-Tac-Toe boards that follows. Each board is represented by three lines and each line consists of three symbols which are each separated by a single space. The symbols will either be a capital X, a capital O or the # symbol (which represents a blank cell in the grid). Each board is guaranteed to be valid; there will be either a single winner or no winner (in which case the game was a tie).

The Output:

For each of the boards, first output a line "`Game #`*g*`:`" where *g* represented the game number (starting at 1). Next, output the game's winner. If X won the game, output the phrase "`Eureka! X wins!`" If O won the game, output "`Gadzooks! O wins!`" If the game was a tie, output "`Lolcat's game!`". Lastly, output a blank line after each game.

Sample Input:

```
3
X O X
O X #
X # O
O O O
X # X
# X #
X O X
X X O
O X O
```
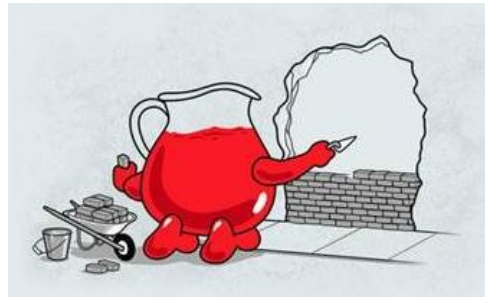
Sample Output:

```
Game #1:
Eureka! X wins!

Game #2:
Gadzooks! O wins!

Game #3:
Lolcat's game!
```

Tray Stacker

Filename: `tray`

"*OH YEAAAAAH,*" boomed the television as a giant anthropomorphized bowl of fruit punch destroyed the home of yet another middle-class family. It was at this moment that Mike knew exactly what he had to do. Fruit punch ice cubes.

There's nothing "kool" about a Class D felony

With sudden purpose, he rushed to the freezer and flung it open, searching for his ice cube trays. Digging through the years of accumulated frozen biomass (making a mental note that tonight would be as good a night as any for a gumbo), he eventually spotted a corner of what could only be one of the elusive trays. With victory in sight, he gripped the tray and attempted to pull it out, only to find that it would not budge. In a fit of frustration, Mike climbed onto the refrigerator door for extra leverage and gave a mighty tug – feeling the tray relent! With a little more effort, the tray finally gave way, soaring out of the freezer and bringing with it an additional number of trays frozen to its underside! The entire stack flew across the kitchen and into the far wall, where the trays separated on impact.

Upon inspecting the crash site, Mike was dismayed to find that, while some ice cubes remained in their cells, others had instead stuck to the bottom of the tray previously above them in the stack! Amazingly, no cubes had detached from the trays entirely. When he tried to restack the trays, Mike was shocked to find that the cubes on the bottom of some of the trays would not always align with the empty cells on the tops of other trays! Mike spent a few seconds trying to restack the trays before becoming really frustrated and giving up entirely (as Mike is prone to do). However, Mike immediately got unfrustrated and ungave up entirely when he had the idea of phoning his best friend (you!) and having you write a program to handle this stacking dilemna.

The Problem:

Given a set of ice cube trays with various ice cubes missing from their tops and cubes from other trays stuck to their bottoms, find the unique bottom-to-top ordering in which they must be stacked. A proper tray stacking is one such that, when brought to room temperature, each cell in each tray holds one ice cube's worth of water. Note that these trays are finicky, in that their design only allows stacking in a single orientation, without rotation. That is, in order to stack two trays, both of their "fronts" must be facing in the same direction.

(Specification continues on next page)

The Input:

Your program must be able to examine several sets of trays, in case this sort of thing ever happens again. The first line of the input will contain only the number of sets of trays to be processed, t ($t > 0$). Immediately following this will be t sets of trays. On the first line of each set will be the number of trays in the set, n ($0 < n \leq 1000$). On the next line will be two positive integers, r and c ($2 \leq r*c \leq 30$), the number of rows and columns in a tray, respectively. Following this will be n tray descriptions, each followed by a blank line. A tray description consists of $2*r$ lines of exactly c 1s and 0s, each separated by a single space, where a 1 indicates an ice cube and a 0 indicates an empty cell. The first r lines describe the top of the tray, while the second r describe the bottom. Tray tops will be distinct among each other, and the same is true of tray bottoms. Mike assures you that each tray was carrying a full payload of ice cubes prior to the collision.

A tray will be given with both a top and a bottom, both of which are presented left-to-right with the same side designated "left." That is, were you to be holding the "top" tray in your hands with the left side in your left hand, after tilting the tray forward to look at its "bottom," that same side would remain in your left hand (see below). As a result, the top view lists the rows of the tray farthest to nearest, while the bottom view lists the rows nearest to farthest.
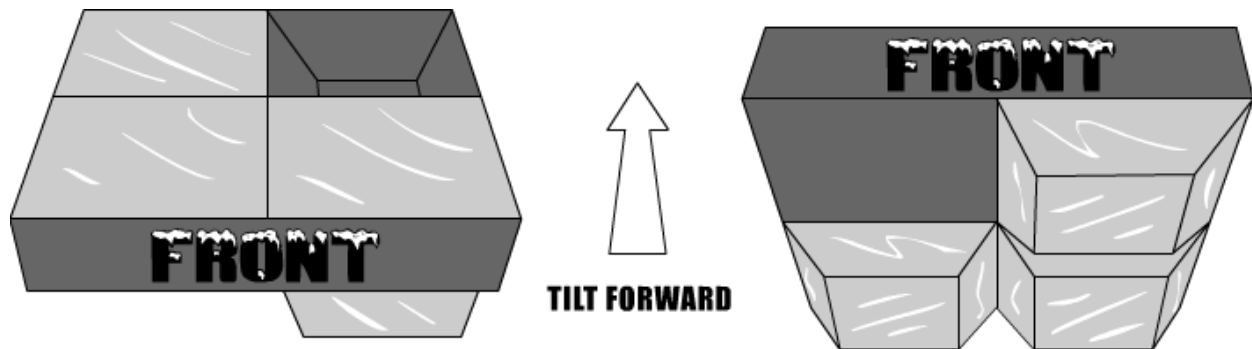
(First tray of second sample test case)

The Output:

For each stack of trays, print "Stack i: " where i is the current stack being processed (starting at 1). Follow this with the bottom-to-top order in which to stack the trays, using the order in which the trays were given (starting at 1), with each tray separated by a single space. Print stack orderings on separate lines.

(Sample Input and Sample Output on next page)

Sample Input:

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

0 1 1 1
1 0 0 1
1 0 0 1
0 0 0 0

1 1 1 1
1 1 1 1
0 1 1 0
1 0 0 0

4
2 2
1 0
1 1
0 1
1 1

1 1
1 1
0 0
0 1

0 0
1 0
1 0
0 1

1 0
0 1
0 0
0 0
```

Sample Output:

```
Stack 1: 1 2 3
Stack 2: 4 3 1 2
```

Bitland Tower Challenge

Filename: `tower`

Bitland is a city filled with many towers. Due to this, there is a famous fitness challenge in Bitland. The challenge is to go up and down each stair of each tower of Bitland. There is a time limit to complete the task. However, since they are in Bitland, they give challengers 2^{32} seconds to complete the task. Bitland uses the Intergalactic 360 Day Space Calendar, so the time limit works out roughly to 138 years. This does not pose much of a problem, so most people ignore it.

Your friend, Bobby Bit-on, has accepted the challenge of Bitland. He wants you to help him. However, he had much difficulty in coming up with a way for you to help him. He could have asked you to give him the total number of orderings to visit each tower, given the number of towers. He also could have asked you, given the heights of all the towers, to come up with a sorted arrangement of towers to get the most difficult(tallest) ones out of the way first, or he could have asked you to come up with the best route to visit all the towers to minimize the walking distance, given the positions of each tower in the Cartesian plane.

He finally settled on something simple. Instead he will count the number of stairs he climbs *up*, and you will help him determine the number of stairs to climb *down* to reach the bottom. Remember Bobby must climb up and down each stair of each tower, so you should assume he will not skip any stairs on the way to the top of each tower or on the way back down. Also once he has climbed up and down a tower and walked to the next tower a tower he will not come back to climb down a stair he might have skipped going down the first time, so tell him the *total* number of stairs to climb down in the given tower.

The Problem:

Given the number of stairs Bobby has climbed up in a tower return the number of stairs your friend will need to go down to return to the ground floor of that same tower.

The Input:

The first line contains a positive integer, *n*, which represents the number of towers Bobby will climb today. The next *n* lines each contain a single positive integer, *v* (*v* < 1000), where *v* denotes the total number of stairs Bobby has climbed in the given tower.

The Output:

For each tower, output on a separate line the total number of stairs Bobby will descend in said tower to reach the bottom floor.

Sample Input:

```
4
3
6
9
15
```

Sample Output:

```
3
6
9
15
```