Twenty-seventh Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
You're Too Slow!	slow
Rock Paper Shenanigans	shenanigans
Find X	find
Battle of the Best	best
Greatest Cosmic Devourers	cosmic
Squirrel Territory	squirrels
Crafty Trolling	craft
Shot Through the Heart…	heart
Musical Chairs	chairs
Nine-Piece Puzzle	puzzle
Zzzyyx or I'm Out	zzzyyx

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

For example, if you are solving Greatest Cosmic Devourers:
Call your program file: cosmic.c, cosmic.cpp or cosmic.java
Call your input file: cosmic.in
Call your Java class: cosmic

You're Too Slow!

Filename: `slow`

Dr. Eggman has been foiled by Sonic for the last time! Every Saturday morning when the new episode airs at 10 AM that blasted hedgehog blows through Eggman's empire and wrecks all of his dastardly plans. All Dr. Eggman wants is a little recognition. He worked very hard to get his Ph.D. in computer science from the University of Robotropolis; he deserves world domination! Dr. Eggman's degree may finally pay off as he has an idea that may take care of that pesky hedgehog. There are only so many new episodes in a season of the show. If Dr. Eggman can create a set of obstacles that takes Sonic long enough to overcome that the season runs out of episodes, then the season finale will have to contain him achieving world domination! That's it! By creating rooms connected by long corridors and filling them with so many robot enemies Sonic will surely fail. If Sonic makes it through all the rooms, even with zero minutes left in the season, he will foil Dr. Eggman and the plot will be a failure.

Dr. Eggman has made several plans for robot filled labyrinths of infinitely small rooms that he needs you to check if Sonic is fast enough to complete. Each enemy takes three minutes to defeat and must be defeated before continuing. If Sonic cannot defeat an enemy before the episode ends, he must wait for the next episode to start the fight. When traveling between rooms Sonic takes time to accelerate. He will cover the first 10 km of the distance at 10 km/min and then cover the rest of the distance at his top speed of 20 km/min. When entering a room Sonic must instantaneously stop to scan the room for the presence of enemies and the location of the exit. Each episode is 22 minutes except for the final episode, the season finale, which is 30 minutes long. Note that the fortresses are built such that extending the duration of any episode by one second will not change their successfulness.

The Problem:

Given the coordinate pairs of rooms and the number of enemies in them, output whether or not Sonic can foil Dr. Eggman before the season ends.

The Input:

Input will begin with a single integer, s ($0 < s < 100$), which represents the number of times Dr. Eggman wants to run the simulation. Each simulation begins with two integers separated by a single space, n ($0 < n < 10000$) and r ($1 < r < 1000$), which represent the number of episodes left in the season and the number of rooms Dr. Eggman plans to make, respectively. This is followed by r lines of three non-negative integers separated by single spaces, x, y and e ($0 \leq x \leq 1000000$; $0 \leq y \leq 1000000$; $0 \leq e < 1000$), which represent the x and y coordinates in kilometers and the number of enemies in that room. No two rooms will be at the same coordinates and each room is connected to the next room by a long straight empty corridor. Sonic starts in the first room listed and must proceed to visit all rooms in the order they are listed until he reaches the final room.

The Output:

For each simulation, first output "`Simulation #i: `" where *i* is the number of the simulation (starting with 1). Then, if Sonic can still foil Dr. Eggman, output "`Abort fortress!`", or if Dr. Eggman finally succeeds, output "`You're too slow!`"

Sample Input:

```
3
5 3
0 0 1
120 160 2
120 1940 1
5 3
3 9 0
123 169 3
123 1949 1
8 5
0 0 8
200 200 7
400 100 8
300 500 3
3000 4000 0
```

Sample Output:

```
Simulation #1: Abort fortress!
Simulation #2: Abort fortress!
Simulation #3: You're too slow!
```

Rock Paper Shenanigans

Filename: `shenanigans`

Andrew and Gabe like to play games in the programming lab. One such game involves a variant of Rock-Paper-Scissors. Kyle is always wondering what in the world is going on when they play, so he would like you to write a program to help him find the winner of the game. There are a few rules he has deduced:

- There are two players
- A *match* is a single play of Rock-Paper-Scissors while a *game* is a series of those matches
- Each player starts without an *advantage*
- Standard Rock-Paper-Scissors rules apply for determining who wins a *match* (rock beats scissors, scissors beats paper, paper beats rock)
- When a player wins a match of Rock-Paper-Scissors, that player gains the advantage
- If no player has the advantage and the two players tie that match, nothing happens
- If a player *x* holds the advantage and that same player *x* wins the match, nothing happens
- If a player *x* holds the advantage but ties player *y* in a match, player *x* then wins the game
- If a player *x* wins a match when player *y* had the advantage, player *y*'s advantage is lost and player *x* gains the advantage

However, Andrew and Gabe like to keep playing even after the winner of the game was decided, making Kyle very confused, which is where you come in!

The Problem:

Given a list of matches that were played during (and possibly after) the game that Kyle is watching, determine the winner.

The Input:

The input will begin with a single positive integer, n ($n \leq 100$), on its own line where n is the number of games that Kyle watched. Each game will be defined across multiple lines. The first line of each game will contain a single positive integer, m ($m \leq 100$), where m is the number of matches in that game (some of which may be played after the game has actually been decided). The next m lines will contain two characters that are either an 'R', 'P' or 'S' representing rock, paper, or scissors, respectively. The two characters will be separated by a single space. The first character is the move from Andrew and the second character is the move from Gabe. Once the game is decided, subsequent matches do not affect the outcome, so ignore the remaining matches in the game. Note that there will always be a winner in each game.

The Output:

For each game, print "`Game #i: `" where *i* is the game number (starting with 1). On that same line, output "`Oh snap! Gabe beat Andrew!`" if Gabe won the game; otherwise, if Andrew won the game, output "`Looks like Andrew won again.`"

Sample Input:

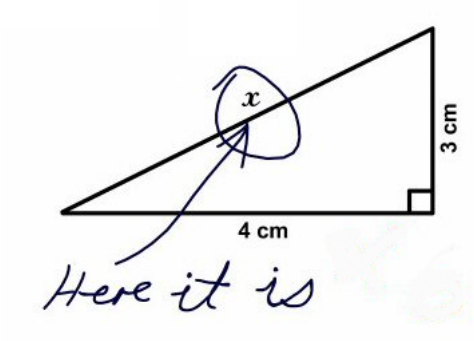
```
2
3
R P
S R
S S
9
R R
S S
P P
S P
R R
S P
P P
R P
P R
```

Sample Output:

```
Game #1: Oh snap! Gabe beat Andrew!
Game #2: Looks like Andrew won again.
```

Find X

Filename: `find`



Skyler is really good at math. The thing he is best at is finding X. But he doesn't like algebra at all (it's just too boring); so, instead he likes to find Xs on the page!

In middle school he was satisfied with just finding the letter 'X'; however, after he advanced to high school he is now interested in finding larger Xs. He noticed that large Xs (consisting of two intersecting diagonals, each with consecutive Xs) can occur in various designs. Some large Xs intersect between characters while others intersect precisely on a single character. Within the latter group, some intersect exactly on the center X; call these large Xs *perfect*. For example, Page A below has a perfect X of size 5 because the two diagonals (each containing 5 Xs) intersect through a single character and it is the center X in both diagonals. Page B has an X (of size 4) that intersects between characters so it is not a valid perfect X; however, a single X is always considered perfect so Page B has multiple perfect Xs of size 1. Page C has an X that intersects through a single character but is only a perfect X of size 3 (the larger X does not intersect through the center X of both diagonals). Note that Xs on the end of a diagonal (or even on other parts of the page) may be ignored in the interest of forming a larger perfect X.

Page A	**Page B**	**Page C**
X000**X**	X00X	X000X
0**X**0**X**0	0XX0	0**X**X**X**0
00**X**00	0XX0	0X**X**X0
0**X**0**X**0	X00X	0**X**X**X**0
X000**X**		X0000

Valid Perfect X of Size 5 *Multiple Perfect Xs of Size 1* *Valid Perfect X of Size 3*

Skyler would like to be able to find the largest such perfect X on a page. He would like you to write a program that can help him! The largest perfect X is the perfect X with the largest size.

The Problem:

Given lines from a page of paper, find the largest perfect X on the page.

The Input:

The first line of input will be a single positive integer, *n*, representing the number of pages to read. Each page will begin with two postive integers, *l* and *w* ($l \leq 50$; $w \leq 50$), representing the number of lines on this page and the length of each line, respectively. Each of the next *l* lines of input will each contain exactly *w* characters. Each character will be an uppercase letter, and it is guaranteed that the letter 'X' will appear at least once on each page.

The Output:

For each page, first output "`Page #i:` " where *i* is the number of the page (beginning with 1), followed by a single integer representing the size of the largest perfect X on the page.

Sample Input:

```
3
3 3
UPU
CXC
FTF
5 5
XABCX
DXEXF
GHXIJ
KXLXM
XNOPX
3 6
AXEAXE
SIXXES
OXYOXY
```

Sample Output:

```
Page #1: 1
Page #2: 5
Page #3: 1
```

Battle of the Best

Filename: `best`

Now that the 2012 apocalypse has been prevented, the world's strongest heroes have gotten extremely bored and have decided to hold a tournament to decide who is the strongest! Since everyone wants to compete in this tournament, no-one wants to be the announcer… Your task is to write a program to automate this process so that everyone can compete!!!

The Problem:

Given two hero's names, and a list of their "super moves" (in the order which they want to use them), output the proper narration of the battle, and who is defeated!

The Input:

The first line of the input file will contain a single positive integer, *t*, the number of battles to narrate. Each battle will be represented by multiple lines. The first line of each battle will contain only the name of the first hero, *x*. The second line will contain only the name of the second hero, *y*. The third line will contain a single positive integer, *n* ($n \leq 10$), representing the number of super moves known by hero *x*. The next *n* lines that follow each contain the name of a single super move that *x* knows. Following the *n* lines, there will be a line with a single positive integer, *m* ($m \leq 10$), representing the number of super moves known by hero *y*. The next *m* lines that follow each contain the name of a single super move that *y* knows.

All names will contain less than 42 characters and will contain only alphanumeric characters (A-Z, a-z and 0-9), spaces and hyphens. In addition, all names will not start or end with a space.

The Output:

For each battle, first output "`Battle #`*b*`! BEGIN!!!`" to begin the battle where *b* is the number of the current battle taking place (starting with 1). Follow this by outputting the super moves of each combatant (each on its own line) using their super moves in order, alternating between the two combatants (starting with hero *x*): first, output "*x* `uses` *a*`!`" where *a* is the corresponding super move that hero *x* knows, and then, output "*y* `uses` *b*`!`" where *b* is the corresponding super move in the input that hero *y* knows, and so on.

When a hero has run out of super moves and it is that hero's turn, that hero has lost the combat. Output "*p* `is defeated by` *q*`'s` *w*`!!!`" where *p* is the losing hero, *q* is winning hero, and *w* is the last move the winning hero used. Output a single blank line after each battle.

Sample Input:

```
2
Batman
James Bond
5
Cloaking Device
Remote-Controlled Batarang
Collapsible Bat-Sword
Bat Beacon
Bat-Shark Repellent
4
Pen Gun
Grappling Suspenders
Radioactive Lint
Shark Gun
Rebecca Black
Chief Judge Glenn
1
Horrible Pop Song
1
Programming
```

Sample Output:

```
Battle #1! BEGIN!!!
Batman uses Cloaking Device!
James Bond uses Pen Gun!
Batman uses Remote-Controlled Batarang!
James Bond uses Grappling Suspenders!
Batman uses Collapsible Bat-Sword!
James Bond uses Radioactive Lint!
Batman uses Bat Beacon!
James Bond uses Shark Gun!
Batman uses Bat-Shark Repellent!
James Bond is defeated by Batman's Bat-Shark Repellent!!!

Battle #2! BEGIN!!!
Rebecca Black uses Horrible Pop Song!
Chief Judge Glenn uses Programming!
Rebecca Black is defeated by Chief Judge Glenn's Programming!!!
```

Greatest Cosmic Devourers

Filename: `cosmic`

In the faraway galaxy of Unarium there burns a single lonely star, orbited by a single lonely rock, inhabited by a single lonely hermit. This single lonely hermit is accompanied by a single lonely text, which tells a single lonely tale about the surprisingly non-singular end of the universe. The hermit is unsurprisingly well-versed in this lore, but tragically has no audience with which to share it. Sadly, he can only dream of what he might say if there were anyone to listen. For example, if a gifted programmer were to suddenly find himself on the hermit's lonely rock, the hermit would be unable to contain himself from blurting out something along the lines of:

"Hello friend! You sure don't talk much. Have I ever told you about the end of the universe? It's actually quite fascinating. You see, when the end is nigh, two visitors will arrive on this plane of existence! They are the primordial serpents Alisteroth and Bobbobadon, tasked with the disposal of our cosmos! These vile worms will enact their plan by first placing all of the stars in existence into an evenly-spaced straight line, then numbering them sequentially starting from zero (the most despicable of numbers). As a rather interesting sidenote, every star in the known universe can end up in any position in this line with equal probability- isn't that just horrifying?! Once the stars have aligned, the worms will feast! Both Alisteroth and Bobbobadon will begin at the star numbered zero and proceed down the line, one star at a time. Alisteroth will only eat a star if its position is divisible by some awful integer *a*, while Bobbobadon will only eat stars whose positions are divisible by some baleful integer *b*. If a star's position in the line is divisible by both *a* and *b*, the two worms will break into a heated argument spanning millions of millenia, ultimately forgetting about the star and moving on down the line, leaving it unharmed! Once the serpents have reached the end of the line, they will withdraw back to their home dimension to sleep off the meal, marking the end of the end of the universe. When the next end is nigh, the monstrosities will awaken once more to start the next end of the universe!

"As you know, there are an infinite number of stars in our universe, of which Unarium has but one. Wouldn't it be just awful if our time together were to suddenly come to an end? I know! Let's work through some scenarios and find out the probability that our conversation won't be interrupted by the first end of the universe! You're pretty good at that sort of thing, right? You sure don't talk much. Have I ever told you about the end of the universe? It's actually quite fascinating. You see…"

The Problem:

Given *a* and *b*, the wretched integers Alisteroth and Bobbobadon use to decide which stars to eat, respectively, determine the probability that the single lonely star of Unarium will survive the first end of the universe. Remember that a star will go unharmed during an end of the universe if its position is divisible by both *a* and *b*, or divisible by neither (see illustration on next page).

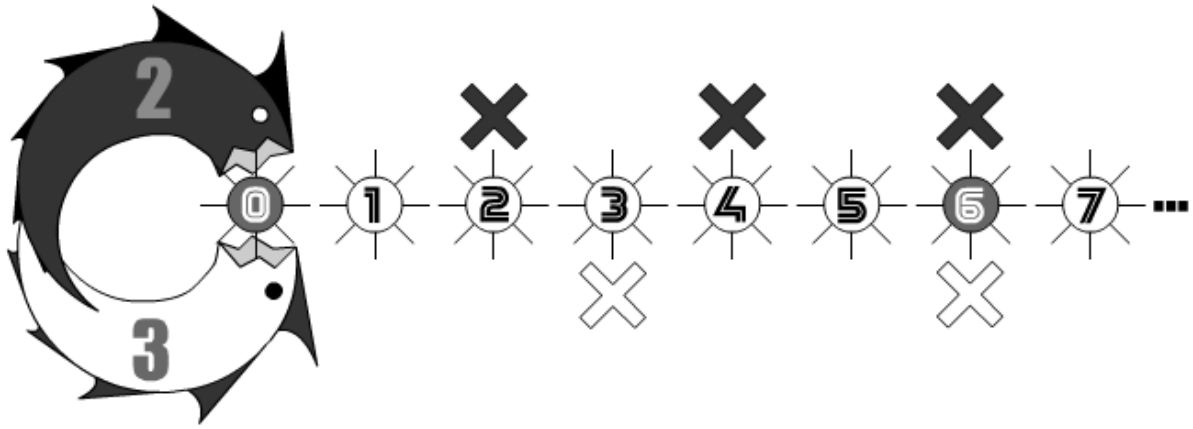


Illustration of the second test case, where a gray star is contested. A single* X *indicates a worm will eat this star. Only unmarked or contested stars (such as stars 0 and 6) will be spared.

The Input:

The first line of the input file will contain a single positive integer, *u*, the number of end-of-the-universe scenarios the hermit will pose. On each of the next *u* lines will be two positive integers, *a* and *b* ($a \leq 1000$; $b \leq 1000$).

The Output:

For each end-of-the-universe scenario, output "`Universe #`*i*`: There's a` *x*`% chance we'll survive!`" where *i* is the scenario number (starting at 1) and *x* is the percent chance the single lonely star of Unarium will not be consumed by Alisteroth or Bobbobadon, rounded to 4 decimal places. As an example of rounding, 35.749650 would be rounded up to 35.7497, while 35.749649 would be rounded down to 35.7496.

Sample Input:

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

Sample Output:

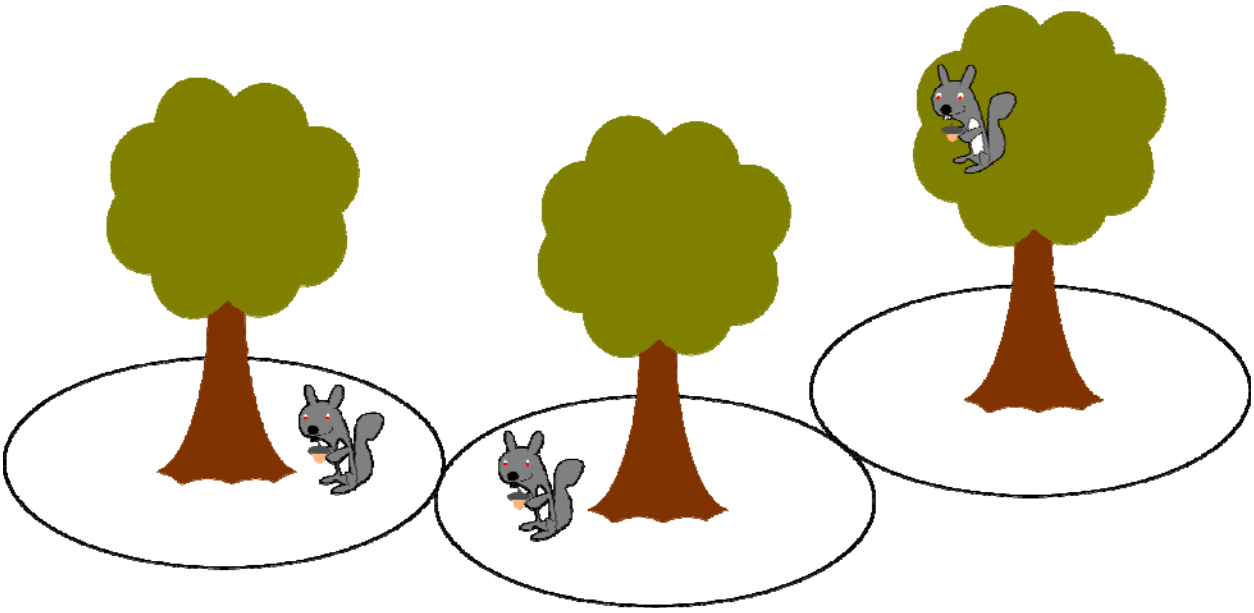
```
Universe #1: There's a 50.0000% chance we'll survive!
Universe #2: There's a 50.0000% chance we'll survive!
Universe #3: There's a 60.0000% chance we'll survive!
Universe #4: There's a 71.4286% chance we'll survive!
```

Squirrel Territory

Filename: `squirrels`

UCF squirrels are some of the fiercest and bravest squirrels ever. Don't let these ultra-cute furry critters fool you! They will steal your food and try to scare you away! Ali has become fascinated with these squirrels. After some intense study, he has found evidence that these squirrels have an elaborate system for maintaining good relations with other squirrels while allowing each squirrel to have adequate space to steal tasty treats from unsuspecting students.

Each squirrel is assigned the territory around the tree in which they live. The squirrel is allowed to move some number of *squirrel steps* in any direction away from the tree. Ali likes this as the squirrels own a circular area around each tree!



Naturally the number of squirrel steps that each squirrel is able to move can vary. To help keep the peace, the squirrels have agreed that all squirrel territories must take up the same area. Furthermore, to prevent conflict, no two squirrel territories can overlap.

The Problem:

Given the locations of trees where the squirrels live, determine the maximum area each squirrel can occupy while following the above rules. Use 3.141592653589793 for the value of π.

The Input:

The first line of input will contain an integer, *c*, representing the number of campuses to examine (UCF has a multitude of satellite campuses after all!). For each campus, the first line consist of a single integer, *t* ($2 \leq t \leq 300$), representing the number of trees on campus where squirrels live. The following *t* lines will contain two integers, *x* and *y* ($-3000 \leq x \leq 3000$; $-3000 \leq y \leq 3000$), representing the location (in squirrel steps) of a tree on the campus. Obviously, no two trees will be located in the same exact spot within a single campus.

The Output:

For each campus, first output "`Campus #i:`" where *i* is the current campus being examined (starting with 1). On the next line write "`Maximum territory area = a`" where *a* represents the maximum area (in squirrel steps squared, of course) that a squirrel can occupy. Output this value rounded to 3 decimal places. As an example of rounding, 14.3965 would be rounded up to 14.397, while 14.3964 would be rounded down to 14.396. Leave a blank line after the output for each campus.

Sample Input:

```
2
2
1 1
2 2
3
-1 -1
-1 1
1 1
```

Sample Output:

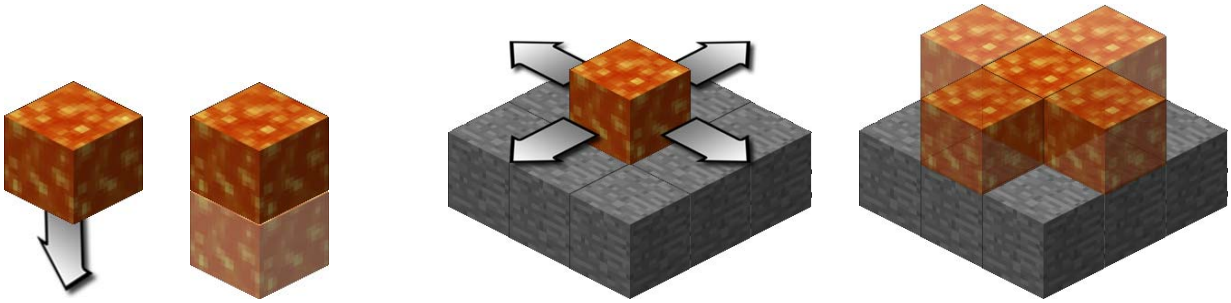
```
Campus #1:
Maximum territory area = 1.571

Campus #2:
Maximum territory area = 3.142
```

Crafty Trolling

Filename: `craft`

Many of your teammates enjoy a popular block-based game involving building virtual structures out of 3D blocks. However, the game has cut into their algorithm study time and you want to help persuade them off of the game. You decide to place a single block of lava (which is actually an extra-dimensional portal, which can float in the air if necessary, that links to a never-ending supply of lava) somewhere in the map, which will cascade into a waterfall of flames and destruction! That's certain to get them off of the game!

The lava moves in a specific way. The lava will never shrink in size; it can only grow. Any block of lava will first try to spread down to fill the space directly beneath it. However, if there is a solid block below (a block of lava is *not* a solid block), the lava will instead attempt to spread to an adjacent space in all four perpendicular directions. If a space contains a solid block, the lava will not spread there. Lava will continue to spread until it has exhausted every possibility, and it will never ascend. If the lava reaches the edge of a given map, those edges are to be considered as solid blocks (the lava will never flow out of the given map).



Now, before decimating your teammates' virtual creations, you want to be sure that the space you've chosen to set the lava will cause a sufficient amount of damage. You decide that the best metric of destruction is to count the number of blocks that resulting lava flow will occupy, and the best way to accomplish this is to create an algorithm to solve the problem.

The Problem:

Given a description of the map, including the starting place of the lava, calculate the total number of blocks filled over time.

The Input:

The first line of the input will be a single positive integer, n, representing the number of maps to analyze. Each map will be defined across multiple lines. For each of these maps, the first line will contain three positive integers, l, w, h ($l \leq 50$; $w \leq 50$; $h \leq 50$), each separated by a single space, representing the length, width, and height of the map, respectively. Next, there will be h slices of the map (each represented by l lines of w consecutive characters for each slice) with the first slice representing the highest point of the map, and each slice below is one block level lower. Each element will be a character: '`.`' for an empty space, '#' for a solid block, and '`*`' for the initial source of lava ('`*`' will occur exactly once per map).

The Output:

For each map, the output will be a line containing "`Map #i: s`", where *i* is the number of the map (starting with 1), and *s* is a single integer describing the number of blocks that the lava takes up at the end *including* the original lava source block.

Sample Input:

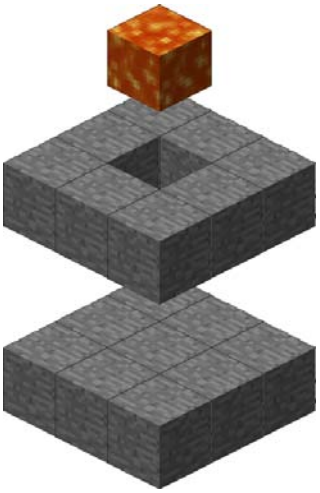
```
2
3 3 3
...
.*.
...
###
#.#
###
###
###
###
4 4 4
##..
#*..
....
....
###.
##..
#...
....
..##
..#.
##..
#...
..##
..##
###.
##..
```

Sample Output:

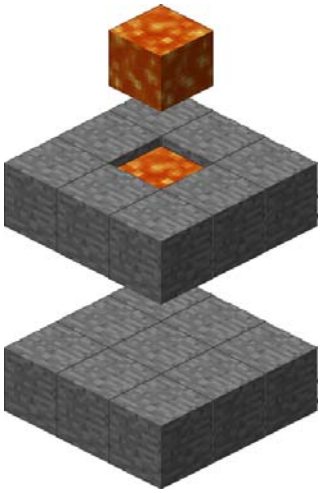
```
Map #1: 2
Map #2: 16
```

Map #1:

Time 0

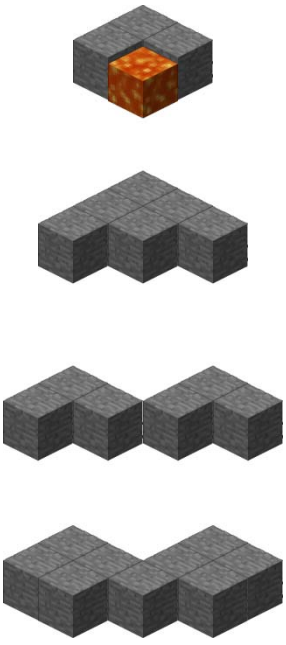


Time 1

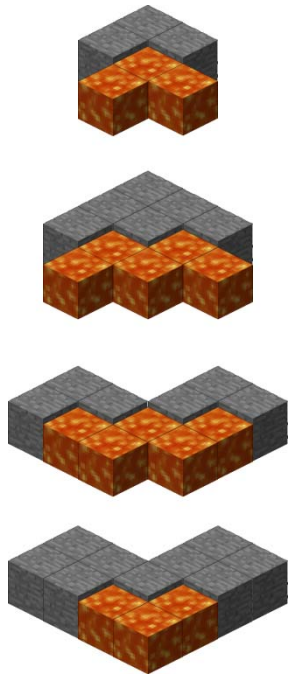


Map #2:

Time 0



Time 7



Shot Through the Heart…

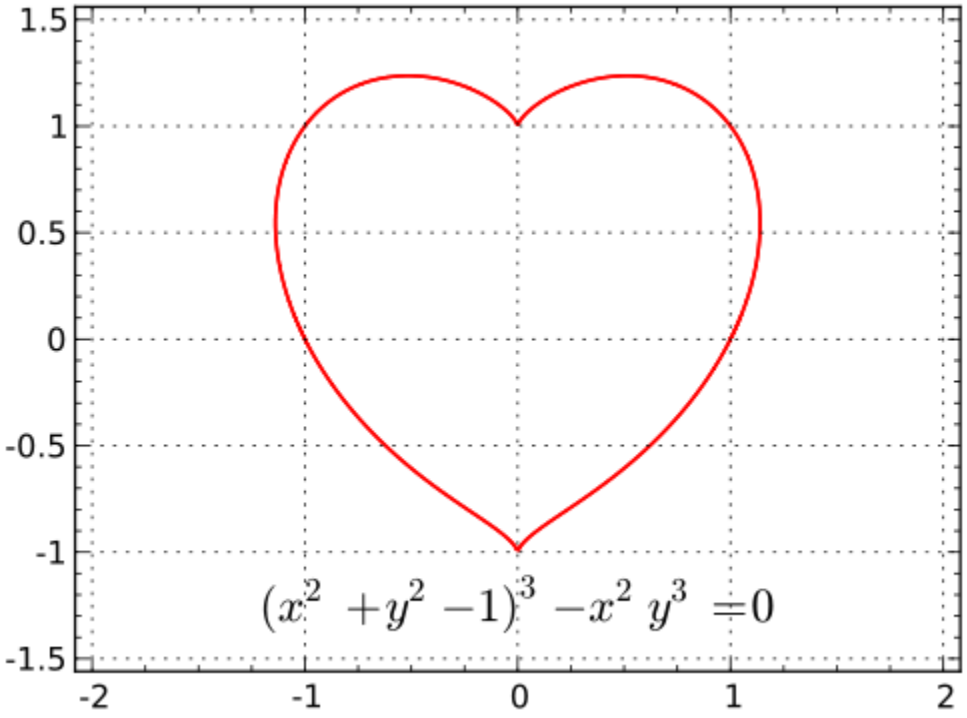
Filename: `heart`

…And you're to blame…Darlin'…You give love a bad name.

In a mirror universe, this song is currently a huge hit! The rock band that sings it, Bon Joswell, with its lead singer, John Bon Joswell, is the envy of guys and the heartthrob of girls. Despite countless hits that reached #1, Bon Joswell makes little money on each album sale. Therefore, as other bands do, they have a new concert tour planned.

In this new concert, the roadies are, of course, planning a pyrotechnics and laser show. It's sure to be the coolest one ever! The concert is also including massive heart-shaped banners (showing cool-looking flaming hearts, naturally). Unfortunately, in the first concert, the roadies found that some of the banners ended up blocking the lasers!

The heart-shaped banners are shaped as below (in feet) and each is described by the given equation shown:



The roadies need to test various points that the lasers will pass through to make sure they do not hit the banners. Obviously, they need your help!

The Problem:

Given points that represent different locations where the lasers go through, determine if each falls within the heart-shaped banner (centered at the origin) or will miss it. No given point will be on or within 0.01 feet of any edge of the flaming heart banner.

The Input:

The first line of input will contain a single positive integer, *n*, representing the number of points. Each of the next *n* lines will contain two real numbers, *x* and *y* ($-100 \leq x \leq 100$; $-100 \leq y \leq 100$), representing the location in feet where the laser goes through.

The Output:

For each point, first output "`Point #i: `" where *i* is the number of the point in the input (starting with 1). Following this, output "`You give love a bad name.`" if the point falls within the heart-shaped banner, or output "`Let it rock!`" if it does not. Output the result for each point on a line by itself.

Sample Input:

```
2
0.0 0.0
2.0 -1.0
```

Sample Output:

```
Point #1: You give love a bad name.
Point #2: Let it rock!
```

Musical Chairs

Filename: `chairs`

Musical Chairs is a fun children's game played when children dance around some chairs in an arrangement. When the music stops the children quickly sit down. At the start of each round some chairs are removed from play to prevent some of the children from sitting. The children who cannot find a chair are eliminated and do not play the next round.

The Problem:

Given the number of children playing a single round of musical chairs and the number of chairs they are dancing around, determine the number of children that will be eliminated when the music stops.

The Input:

The first line of input will contain a positive integer, *r*, representing the number of rounds. For each round, on a single line there will be two positive integers, *n* and *m* ($m \leq n \leq 300$), representing the number of children and number of chairs, respectively, for that round.

The Output:

For each round, first output "`Round #i:` " where *i* is the current round (starting with 1). Then, on the same line output "*c* `children eliminated`" where *c* represents the number of children that were eliminated that round.

Sample Input:

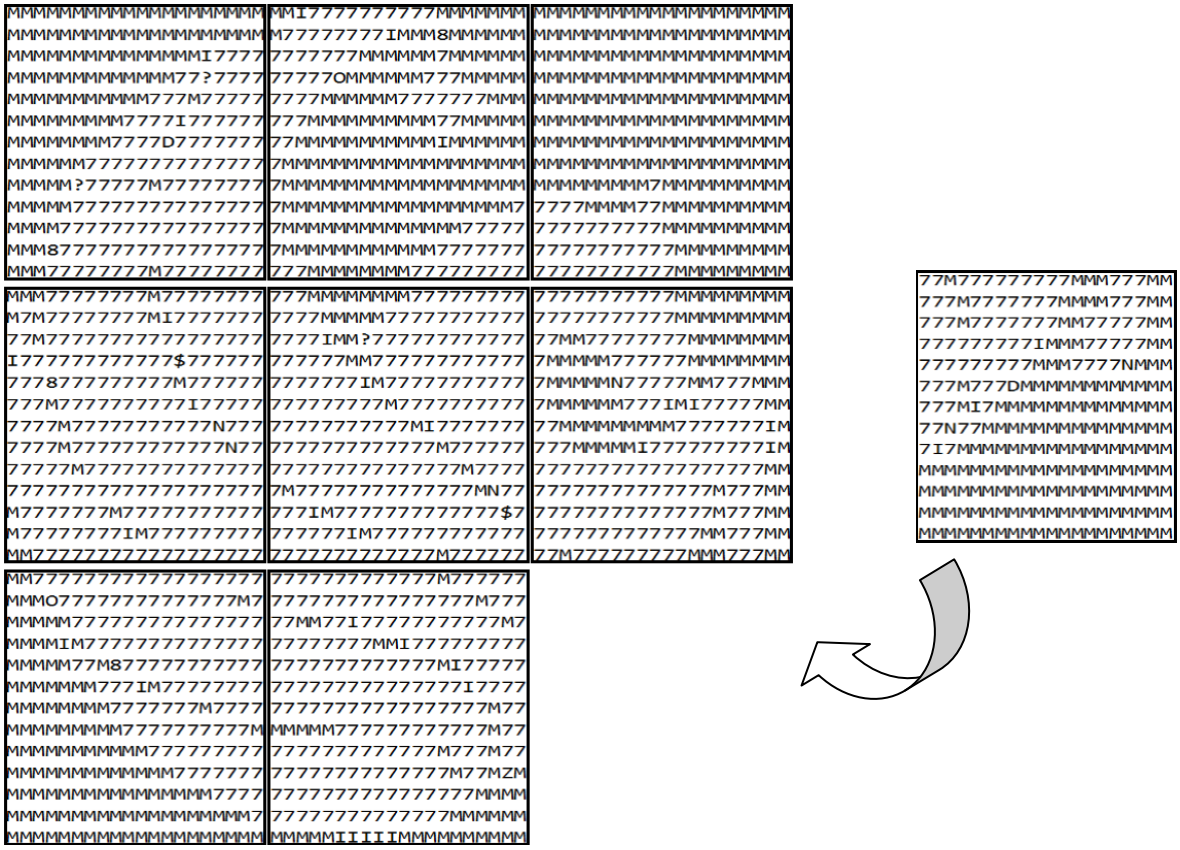
```
3
1 1
3 2
8 4
```

Sample Output:

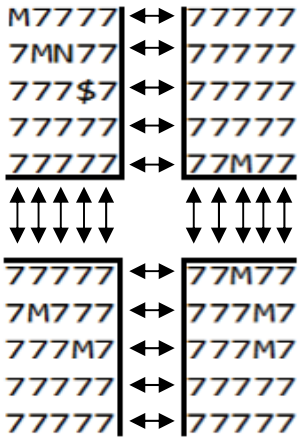
```
Round #1: 0 children eliminated
Round #2: 1 children eliminated
Round #3: 4 children eliminated
```

Nine-Piece Puzzle

Filename: `puzzle`



Patrick loves solving puzzles, but some are too hard for him (he likes easy puzzles). His favorite puzzles are nine-piece puzzles. These puzzles have nine square pieces which fit together to make a 3-piece by 3-piece square. Since these pieces have flat edges you might wonder how he knows which pieces are adjacent. The answer is he looks at the picture on them. Two pieces fit together if and only if every pixel along their edges match perfectly. Taking a closer look at the edges in the above puzzle we can see that each pixel (represented as a single character) matches along the edges of the pieces:



Sometimes the puzzle pieces get mixed up and Patrick isn't certain that there is a way to solve the puzzle any more. Patrick would like your help to tell him if there is a way to assemble the puzzle. Patrick knows which way the pieces need to go, so you will never need to rotate a piece. Note that sometimes when the pieces get mixed up there might be multiple ways to solve the puzzle. Alert Patrick if this is ever the case.

The Problem:

Given nine square puzzle pieces, determine if there is a way to assemble the pieces in a 3 x 3 square such that all adjacent edges match. A match occurs only when all adjacent pixels along the edge of a piece are the same as the corresponding pixel on the other piece.

The Input:

The first line of input will be a single positive integer, *n*, representing the number of nine-piece puzzles Patrick would like you to attempt to assemble. Each puzzle will begin with one positive integer, *s* ($s \leq 10$), representing the length of the side for each puzzle piece. The nine puzzle pieces are input as *9s* lines, each exactly *s* characters long. The lines may include any upper or lower case letter, number and/or punctuation, but it will not contain whitespace.

The Output:

For each puzzle, output "`Puzzle #i: `" where *i* is the number of the puzzle (starting with 1). Next, on the same line output "`YES`" if there is one distinct way to assemble the puzzle, "`NO`" if there is no way to assemble the puzzle or "`MULTIPLE`" if there are multiple ways to assemble the puzzle. Note that two puzzles are different based on the arrangement of the pieces even if their resulting appearance is the same. Output each puzzle on its own line.

(Sample Input/Output on next page)

Sample Input:

```
2
3
AXB
AXB
ZBZ
ZCC
DXX
ZFF
ZEZ
EXF
EXF
XXA
XXA
AAZ
ZBZ
XXD
ZEZ
DDZ
XXE
XXE
BXX
BXX
ZCC
AAZ
CCX
DDZ
ZFF
FXX
FXX
1
A
A
A
A
A
A
A
A
A
```

Sample Output:

```
Puzzle #1: YES
Puzzle #2: MULTIPLE
```

Zzzyyx or I'm Out

Filename: `zzzyyx`

The evil sorcerer Zzzyyx has finally snapped! Driven insane by a life in which alphabetical order dictates that he is always called last, he has hatched a final plan to strike back at the Council of Mecatol Lex. The council meets once per week to review and confirm the alphabetical ordering of the characters in Mecatolish, the official language of Mecatol Lex. If any errors are found, the Grand Vizier of the council, determined by the member whose name comes first alphabetically, will draft a proper ordering to overwrite the offending one. The rest of the council will then descend into months of argument and dispute, while the Grand Vizier retreats to rearrange the words in the Lexicon, the official dictionary of Mecatolish. In every iteration of Mecatolish since its creation, the last three letters of the alphabet have been "XYZ" and this continues to this day. In the history of Mecatol Lex, there has never been a person of lower status than the evil sorcerer Zzzyyx.

But he has a plan! If Zzzyyx could somehow become the Grand Vizier of the council, he could rearrange the alphabet and put an end to decades of ridicule. Of course, since a Mecatolian's name is engraved into his forehead at birth, Zzzyyx could never accomplish such a feat. But Zzzyyx is an evil sorcerer! In training! With a plan!

A dabbler in the art of mechromancy, Zzzyyx has decided to fabricate a clockwork council member, one guaranteed to be elected Grand Vizier and put in motion his wicked revenge! He has had no trouble assembling the mind and body of his creation, but now comes the crucial element: its name. Zzzyyx will assemble the automaton's name plate from a collection of individual character plates. He only has so many of each kind, however, and the names of council members must contain at least k characters depending on the week (a statute introduced following the maniacal magician A's thousand year reign of terror). To further complicate the situation, Zzzyyx literally just knocked over all of the character plates while explaining this to you, leaving a jumbled mess on the floor. Could you lend the poor evil sorcerer in training a hand in his efforts to bring down the governing body of Mecatol Lex and send billions of lives into chaos? He has his evil sorcerer mid-term tomorrow!

The Problem:

Given the current alphabetical ordering of the characters in Mecatolish and the character plates on Zzzyyx's floor, find the alphabetically first name he can create using at least k characters. A name is considered to come before another alphabetically if, at the first position where the two names differ, the character in the first name is alphabetically less than the character in the second name. If a string of length a matches the first a characters of a string of length b (where $a < b$), then the string of length a is alphabetically first.

The Input:

The first line of the input file will contain a single positive integer, *w*, representing the number of weeks over which Zzzyyx will be attempting to overthrow the Council of Mecatol Lex. Following this will be *w* week descriptions. A week description will begin with a single positive integer, *k* ($k \leq 150$) on its own line. On the following line will be a string of 26 uppercase characters representing the current alphabetical ordering of Mecatolish. Each of the letters between 'A' and 'Z', inclusive, will occur exactly once in this string (and the string will always end in "XYZ"). A letter's position in this string indicates that it comes alphabetically before all characters that follow it. Following this line will be another line containg a string of length s ($k \leq s \leq 300$), representing the pile of character plates on Zzzyyx's floor. This string will contain only uppercase letters between 'A' and 'Z', inclusive.

The Output:

For each week, output "`Week #`*`i`*`: His name is `*`x`*`!`" where *i* is the week number (starting at 1) and *x* is the alphabetically first name the evil sorcerer Zzzyyx can create using at least *k* character plates, printed in upper case.

Sample Input:

```
3
4
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ZTYZYZRZAZ
4
DBAHVFECIJKLMNOPQRSTUGWXYZ
PACRVGGED
6
WVUTSRQPONMLKJIHGFEDCBAXYZ
ZZZYYX
```

Sample Output:

```
Week #1: His name is ARTY!
Week #2: His name is DAVE!
Week #3: His name is XYYZZZ!
```