Thirtieth Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
Disc Jockey	disc
Nick and Moriarty	nick
Welcome to Moe's!!!	moes
Baby Names	names
The Number Thirty	thirty
Too Much to Two	two
Game of Plinko	plinko
Guessing Game	guessing
Chomp Chomp!	chomp
Chessboard Construction	chessboard
Corn Maze	maze

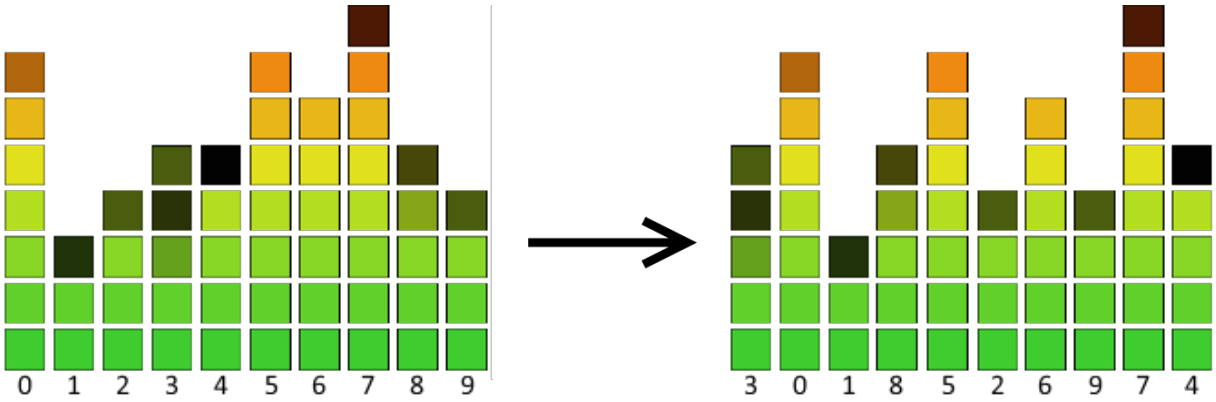
Call your program file: *filename*.c, *filename*.cpp, or *filename*.java

For example, if you are solving Nick and Moriarty:
Call your program file: nick.c, nick.cpp or nick.java
Call your Java class: nick

Disc Jockey

Filename: `disc`

Gerald's brother, Charlie, recently had a birthday party. Since all of Charlie's friends know that he wants to become a D.J. when he grows up, they gave him many cool new gifts related to music. However, the best gift was a new stereo which had a fancy graphic visualizer on the front. A graphic visualizer is a feature found in electronic music devices which generates animated imagery synchronized to music being played. Charlie's visualizer has *n* different columns of squares which represent the amplitude of a particular frequency.



Gerald is very jealous of his little brother's new stereo and can't stand to listen to the music that his brother plays through its speakers, so he has devised a plot to solve his problem. He has a theory for how sound works, and it is this: All sounds can be assigned a value *s*. To calculate *s*, you must sum all the absolute difference in heights between each pair of adjacent columns. For example, in the left image above, column 0 and column 1 are 4 apart, column 1 and column 2 are 1 apart, and so on. If you add all these differences you would get a value of $s = 15$. Gerald also believes that if a sound wave's columns are rearranged in such a way that *s* is as large as possible and then is played at the same time as the original sound, the two will cancel out and no sound will be heard. There may be more than one valid way to do this. If you are to rearrange the columns in the original wave to the order on the right, you can achieve a maximum $s = 24$. Note that an entire column must be rearranged when done so and not just portions of it.

The Problem:

Help Gerald find the maximum value of *s* for a given sound wave represented by *n* columns when he is able to swap around any columns. This way he can cancel out all of his brother's music and never be bothered again!

The Input:

The first line will contain a single, positive integer, *t*, representing the number of different wave forms that need to be processed. The next *t* lines will start with an integer, *n* ($1 \leq n \leq 10$), followed by *n* integers, c_0 through c_{n-1} ($0 \leq c_i \leq 10^8$), representing the height of each column in a wave representation.

The Output:

For each wave, output a single line of the form "`Wave #i: s`" where *i* represents the number of the wave which you are processing (starting from 1), and *s* represents the maximum *s* value of that wave.

Sample Input:

```
3
3 1 2 3
5 9 2 5 3 1
10 7 3 4 5 5 7 6 8 5 4
```

Sample Output:

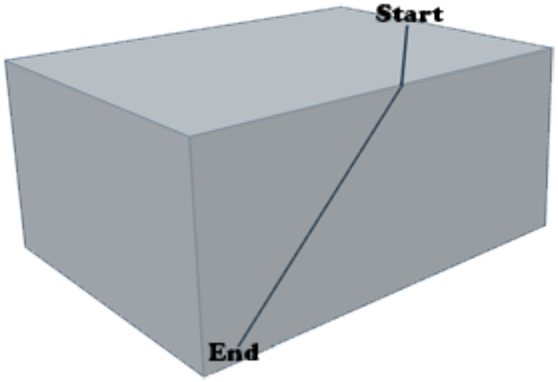
```
Wave #1: 3
Wave #2: 21
Wave #3: 24
```

Nick and Moriarty

Filename: `nick`

Yet again, mad scientist Nick and his nervous nephew Moriarty find themselves in a predicament with very low probability of survival. This time they are stuck in the Infinite Hyperspace of Orthogonal Prisms (or IHOP for short), a universe comprised of rectangular prism "planets" all hovering around but not touching. All prisms are axis-aligned and use separate coordinate systems such that one of the corners of the prisms is the origin.

The organisms of IHOP are all made of axis-aligned tiny cubes, and are extremely prejudiced against non-breakfast themed and non-axis-aligned organisms. Unfortunately, that means they want to disintegrate Nick and Moriarty to rid the universe of their repulsive curves and non-breakfast relatableness. Thus, the duo must escape to the shuttle station in the shortest way possible! However, they cannot tunnel through the prism planet. Instead, they must travel on its surface, and over the edges or corners if necessary.



The Problem:

Given the x, y, and z coordinates of the location of Nick and Moriarty on the rectangular prism and the x, y, and z coordinates of the shuttle station on the same rectangular prism, output the distance of the shortest path from Nick and Moriarty to the shuttle station along the surface of the prism.

The Input:

The first line of the input contains a single, positive integer, p, representing the number of parallel universes to process. This is followed by p lines, each with 9 integers. The first three are x_m, y_m, and z_m ($1 \leq x_m, y_m, z_m \leq 1{,}000$), which are the maximum x, y and z dimensions of the prism's coordinate system. The next three are x_1, y_1, and z_1 representing the current location of Nick and Moriarty on the prism. The final three are x_2, y_2, and z_2 representing the location of the shuttle station on the prism.

Since Nick and Moriarty's location and the location of the shuttle station are on the surface of the prism, the x, y and z locations have the following properties (where i is 1 or 2):

At least one of the following six is true:			**All of the following are true:**
$x_i = 0$	$y_i = 0$	$z_i = 0$	$0 \leq x_i \leq x_m$
$x_i = x_m$	$y_i = y_m$	$z_i = z_m$	$0 \leq y_i \leq y_m$
			$0 \leq z_i \leq z_m$

The Output:

For each parallel universe, output "`Universe #`*i*`: `*d*" where i is the parallel universe's number in the input (starting from 1), and d is the length of the shortest path from Nick and Moriarty's location to the shuttle station's location rounded to two decimal places (for example, 5.345 rounds to 5.35, and 5.3449 rounds to 5.34).

Sample Input:

```
2
10 7 5 9 4 5 1 0 1
10 7 5 9 4 5 5 5 0
```

Sample Output:

```
Universe #1: 11.31
Universe #2: 10.77
```

Welcome to Moe's!!!

Filename: `moes`

Everyone knows that whenever you walk into a Moe's restaurant you are greeted by enthusiastic shouts from all of the employees welcoming you to their establishment. They all greet every customer by shouting "Welcome to Moe's!!!" whenever someone new walks in the door. If someone walks in the door that has already been there that day, the workers just continue working instead of repeating the welcoming shout.

The Problem:

Your task is to create a program to help these busy Moe's employees by telling them exactly when they should shout "Welcome to Moe's!!!" because a customer that *has not been there yet* has walked into the store.

The Input:

The first line of the input will contain a single, positive integer, c ($1 \leq c \leq 10{,}000$), representing the number of potentially new customers to process. Each of the next c lines will contain a one-word name of the customer (made up of only letters) with the first letter capitalized and all others lowercase. No name will be longer than 25 letters.

The Output:

For each name given, output one line containing "`Customer #`*`i`*`: `" where *i* is the customer's number (starting from 1) in the order given in the input (note that repeat customers will have more than one customer number), followed by either "`Welcome to Moe's!!!`" if this is the first time you have seen this customer, or "`**continue working**`" if this customer had already been seen and greeted with the resounding shout. No two customers have the same name, but a single customer may appear in the given list multiple times.

(Sample Input and Output are on following page)

Sample Input:

```
7
Ethan
Jack
Ethan
Billy
Susan
Jack
Jack
```

Sample Output:

```
Customer #1: Welcome to Moe’s!!!
Customer #2: Welcome to Moe’s!!!
Customer #3: **continue working**
Customer #4: Welcome to Moe’s!!!
Customer #5: Welcome to Moe’s!!!
Customer #6: **continue working**
Customer #7: **continue working**
```

Baby Names

Filename: `names`

A new fad has arisen in the area of baby names. Parents no longer prefer to call their children standard, meaningless names. Rather, they now commonly combine both of their names to form one for the child which will be perfect and unique. The parents combine their names in the following way:

1. Both the expectant mother and father write their names out on paper.
2. Next, the parents decide whose name will be the beginning and whose will be the ending.
3. Then, they take the name for the beginning and cross out letters from the back of that name. At least one letter will be crossed out and at least one letter will be left. The same crossing out of letters is done for the name being used for the ending but from the front this time. However, now when crossing out the letters, if the beginning part ends in a vowel, the ending part must start with a consonant (Y is considered a consonant, *not* a vowel); similarly, if the beginning part ends in a consonant, the ending part must start with a vowel. Furthermore, neither the beginning nor the ending should be left with no letters.
4. Finally, the shortened ending is appended to the shortened beginning and a potential baby name has now been created.

An example of the parents doing this process can be seen below (note: this is just one of many ways the parents could combine their two names to form a baby name).

1. Write names out
   - ➢ Father's name: **JAMES**
   - ➢ Mother's name: **MARY**
2. Decide ordering
   - ➢ First half: **JAMES**
   - ➢ Latter half: **MARY**
3. Shorten the names
   - ➢ Shortened first half: **JA**
   - ➢ Shortened second half: **Y**
4. Combine the names
   - ➢ Baby name: **JAY**

The Problem:

Your task is to write a program which, given the names of the two parents, will output every possible valid name that can be formed from the process above. Furthermore, the parents want this list in alphabetical order with no repeats so they can easily sort through it.

The Input:

The first line of the input will contain a single, positive integer, *n*, representing the number of couples that are expecting a baby and need your help in creating the perfect name. The next *n* lines will each contain the Father's name followed by a single space and then the Mother's name. Each name consists entirely of capital letters from the standard English alphabet. Each of the parents' names will be at least 2 letters long and no longer than 20 letters.

The Output:

For each couple, first output "`Couple #`*`i`*`: `*`p`*` possible names`" where *`i`* represents the couple number from its order in the input (starting with 1) and *`p`* represents the number of different valid names that can be formed. Then, the next *`p`* lines should each contain one possible valid baby name consisting only of capital letters. All the valid names for one couple should be in alphabetical order with no repeats. It is, in theory, possible that one potential name for the baby is the same as one of the parent's names. This is okay, and the parents still want that name included in the list. Output an extra line after the output for each couple.

Sample Input:

```
2
JAMES MARY
ABE JO
```

Sample Output:

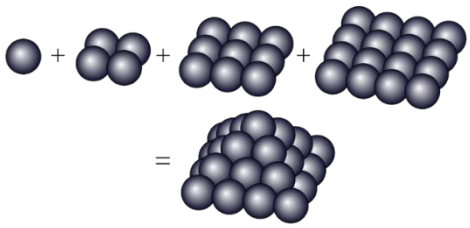
```
Couple #1: 10 possible names
JAMARY
JAMERY
JAMEY
JARY
JAY
MAMES
MARAMES
MARES
MAS
MES

Couple #2: 2 possible names
ABO
JE
```

The Number Thirty

Filename: `thirty`

Welcome to the 30th Annual UCF High School Programming Tournament! Hard to believe it all started on May 2, 1987! Speaking of 30, did you know that 30 is the sum of the first four squares, which makes it a square pyramidal number? $1^2 + 2^2 + 3^2 + 4^2 = 1 + 4 + 9 + 16 = 30$.



In addition, 30 is also the smallest sphenic number! A sphenic number is a positive integer that is the product of three distinct prime numbers. The number 30 is also the smallest sphenic number of the form $2 \times 3 \times r$, where r is a prime greater than 3. What other numbers are sphenic numbers of the form $2 \times 3 \times r$?

The Problem:

Given a sphenic number guaranteed to be in the form of $2 \times 3 \times r$, determine the prime number r.

The Input:

The first line will contain a single, positive integer, n, representing the number of integers to be processed. Following this, each of the next n lines will contain such a single, positive integer (each of the form $2 \times 3 \times r$ for some prime integer r).

The Output:

For each integer to be processed, output the prime number r on its own line.

Sample Input:

```
3
30
66
246
```

Sample Output:

```
5
11
41
```

Too Much to Two

Filename: `two`

Gabe is always looking for arbitrary patterns to remember numbers, and when the year turned 2016, he found one that practically oozed the number two! He noted, of course, *two* properties. First, it's possible to partition 2016 into exactly two continuous substrings ("2" and "016") such that each substring, when read as a number, is a power of two (aha, two powers of two!). Second, it is possible to add some power of two to 2016 (32 in this case) to make it a power of two (2048)! Wow! Gabe decided to call such numbers that have both of these properties *hyper-evens*. Now he wants your help to find more hyper-evens!

The Problem:

Given some positive integer, *n*, find the first hyper-even that is strictly greater than *n*.

The Input:

The first line of the input contains a single, positive integer, *t*, representing the number of integers Gabe wants to ask about. Each of the next *t* lines contains a single integer, *n* ($1 \leq n \leq 2^{15}$).

The Output:

For each integer, output "`The next hyper-even after` *n* `is` *e*" where *n* is the integer from the input and *e* is the smallest hyper-even strictly greater than *n*.

Sample Input:

```
3
32768
50
2016
```

Sample Output:

```
The next hyper-even after 32768 is 64512
The next hyper-even after 50 is 1008
The next hyper-even after 2016 is 2032
```

Game of Plinko

Filename: `plinko`

Step right up! Step right up! Test your skills at the wonderful game of Plinko! The rules are simple: you drop a coin at some column at the top of the board and watch your coin roll down and hopefully land in a column with a super high score! But be careful, you only get one try!

The Problem:

Given a Plinko board, find the best column to drop the coin to score the most points. Plinko works as follows:

- The first and last columns of a Plinko board will contain only "`|`". The top row will contain only the two outer walls and "`.`"s and the bottom row will contain the two outer walls and only the digits 0-9.
- A coin can be dropped in any column except for the first and last columns.
- A dropped coin will continue falling until it lands on a "_", "`\`", "`/`", "`|`" or until it reaches the bottom. Then, one of many scenarios take place:
    - After a coin reaches the bottom, it scores the number of points given by the digit in that particular column. If the coin never reaches the bottom, it scores 0 points.
    - When a coin lands on a "_" or a "`|`", it stops movement immediately (Yes, the coin can balance on the small point of the "`|`").
    - When a coin lands on a "`\`", it will start rolling straight to the right and will continue going right until one of two scenarios occur:
        - It either hits a "`\`", "`/`", or "`|`", at which point it will stop all motion and the game is over.
        - It will continue rolling until it is no longer rolling on top of a "_" and is instead on a "`.`" at which point the coin will stop all left/right motion and will fall straight down.
    - The same applies for when a coin lands on a "`/`", except the coin will roll to the left instead.

The Input:

The first line of the input will contain a single, positive integer, *b*, representing the number of Plinko boards to consider. Following this, for each Plinko board, the first line will contain two positive integers, *w* and *h* ($3 \leq w \leq 100$; $3 \leq h \leq 100$), representing the width and height of the Plinko board, respectively. The next *h* lines will each contain *w* characters representing the Plinko board. The first line of each Plinko board will always contain the two walls and *w*-2 "`.`"s; the last line of each Plinko board will always contain the two walls and *w*-2 single digit integers from 0 to 9 representing the score received if a coin lands there. The boards given in the input will only contain the characters described above.

The Output:

For each Plinko board given in the input, first output the header "`Board #`*`i`*`: `" where *`i`* is the Plinko board number (starting from 1 in the order they appear in the input). This will be immediately followed by one of two messages:

- If you can score at least 1 point, follow this header on the same line by the message "`Drop at column `*`x`*` for a score of `*`y`*` points.`" where *x* represents the column number that results in the most points (columns are numbered starting from 0 on the left) and where *y* represents the score that will be received by dropping the coin in that column, respectively. If more than one column would result in the most points, choose the left-most such column.
- Otherwise, if every column would give you a score of 0 points, follow the header by the message "`Don't even bother dropping a coin. You can't win!`".

Output a blank line after each statement.

Sample Input:

```
2
20 6
|...................|
|\____.\__.__/\./\.|
|._____/..|.|...\__.|
|\__|....|.|....._/|
||.......|\_.......|
|98765432112345678 9|
7 5
|.....|
|\__._|
|..._.|
|.....|
|12345|
```

Sample Output:

```
Board #1: Drop at column 14 for a score of 7 points.

Board #2: Don't even bother dropping a coin. You can't win!
```

Guessing Game

Filename: `guessing`

Adam and Jimmy love to play a simple guessing game. They start by choosing some range of numbers and then Adam thinks of any integer within that range (including the start and end). Jimmy then tries to guess the number. If he is correct, he wins. Otherwise, Adam will tell him whether the true number is lower than his guess or higher than his guess. Jimmy will then continue guessing until he finally guesses the correct number. Jimmy's score is then equal to the number of guesses it took until he guessed correctly (lower scores are better).

The Problem:

Jimmy has devised a plan which he believes may minimize his score and help him guess the number that Adam is thinking of as quickly as possible. His plan is outlined as follows:

1. Given the range of possible numbers, find the middle number (by averaging the two endpoints and rounding up to the nearest integer if necessary) and then guess that number.
2. If he is correct, he wins! Otherwise, if his guess was too high, he will repeat his strategy with a new range of possible numbers with the same lower bound as before and with an upper bound of one number less than his guess. If his guess was too low, he will repeat his strategy with a new range of possible numbers with a lower bound of one number higher than his guess and with the same upper bound.

Here is an example of Jimmy's strategy in action with the range **[1, 100]** and Adam's number being **33**:

1. Range is initially **[1, 100]**. Jimmy incorrectly guesses **51** which is the middle number between 1 and 100 rounded up (true middle number is 50.5). His guess of **51** was too high.
2. Range is now **[1, 50]**. Jimmy incorrectly guesses **26** which is too low.
3. Range is now **[27, 50]**. Jimmy incorrectly guesses **39** which is too high.
4. Range is now **[27, 38]**. Jimmy correctly guesses **33** and wins with a score of **4**.

Your task is to write a program which, given a range of possible numbers Adam can choose from and given Adam's chosen number, can calculate Jimmy's final score assuming he follows his strategy described above.

The Input:

The first line of the input will contain a single, positive integer, *n*, representing the number of times Adam and Jimmy are planning to play the guessing game. The next *n* lines will each contain 3 integers, *a*, *b* and *x* ($0 \leq a \leq x \leq b \leq 1{,}000{,}000$), where *a* to *b* is the range of numbers considered and *x* is Adam's selected integer.

The Output:

For each guessing game, output one line with the format "`Game #`*`i`*`: `*`g`*` guesses`" where *`i`* represents the game's order given in the input (starting with 1) and *`g`* represents the number of guesses it will take Jimmy to win the game. In the special case that Jimmy will win in 1 guess exactly, output "`Game #`*`i`*`: 1 guess`" instead, maintaining that *`i`* represents the game's order given in the input starting with 1.

Sample Input:

```
3
1 100 33
1 3 2
11 19 14
```

Sample Output:

```
Game #1: 4 guesses
Game #2: 1 guess
Game #3: 3 guesses
```

Chomp Chomp!

Filename: `chomp`

Chain Chomp's favorite activity is chomping. She loves chomping so much that she takes bites out of other creatures' meals when they are not looking. Today, she plans to take a few bites from Bowser's plate while he is asleep; however, if she gets caught, she will surely be roasted!

Bowser's plate contains n morsels of food, arranged in a line. Bowser is very suspicious of others stealing his food, so he has used his mathematical skills to create a test for whether or not his food has been tampered with. Each piece of food has a nutritional value, and Bowser has calculated the sum of these values for all of the food on his plate before going to sleep. Unfortunately for him but lucky for those within the range of his fire breath, he cannot remember big numbers very well. When he figured out this sum, to make sure he could remember it more easily, he divided the sum by k and instead remembered the remainder from this division. After he wakes up, he will repeat the process and ensure the same remainder is produced.

If Chain Chomp chooses which items to eat carefully, she can leave certain items on the plate which still produce the same remainder when their nutrional values are added together and divided by k. Chain Chomp plans to take two chomps out of Bowser's food. Each chomp consists of eating some non-empty chain of consecutive items from the plate. Of course, these two chains cannot have any pieces of food in common, since Chain Chomp consumes all of the food that she chomps. Also, there must be at least one piece of food between these two chains. Otherwise, it would look like only one chomp was made, which would be much less satisfying.

For example, the following is a possible arrangement of the nutritional values on Bowser's plate:

7	1	5	4	3	2	6	8

Below is an invalid choice of chains since there is no food between the two chains:

7	1	5	4	3	2	6	8

Below is a valid choice of chains for $k = 5$ since the sum of both the original plate and the plate with both chomps missing each produce a remainder of 1.

7	1	5	4	3	2	6	8

Chain Chomp is trying to decide which chains to chomp, and is weighing her options one at a time. To make sure she does not miss any possible options, she would like to know how many ways she can choose two chains. Note that chomps are unique from each other based on their positions and not their values. Note that the order in which the two chomps occur is not important (so eating items 2 and 3 followed by items 6 and 7 is the same as eating items 6 and 7 followed by items 2 and 3 and should not be counted twice). Can you help her count how many choices she has?

The Problem:

Given the nutritional values of the food items on Bowser's plate from left to right, count the number of ways Chain Chomp can eat two contiguous chains of food that have no elements in common and at least one uneaten item between them such that the remaining items pass Bowser's test.

The Input:

The first line contains a single, positive integer, *p*, representing the number of plates for which Chain Chomp would like to count her options. For each plate, the following information is given:

First is a line containing two positive integers, *n* and *k* ($1 \leq n \leq 3{,}000$; $2 \leq k \leq 1{,}000{,}000$), denoting the number of items on Bowser's plate and the number he will divide by in his test, respectively.

This is followed by another line containing *n* positive integers, a_i ($1 \leq a_i \leq 1{,}000{,}000$), where the i^{th} number on this line is the nutritional value of the i^{th} food item.

The Output:

For each plate, output a single line "`Plate #`*p*`: `*w*" where *p* is the number of the plate being considered (in the order they were given in the input, starting from 1), and *w* is the number of ways Chain Chomp can choose two chains to chomp.

Sample Input:

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

Sample Output:

```
Plate #1: 1
Plate #2: 25
```

Chessboard Construction

Filename: `chessboard`

Chester is a huge fan of chess. His birthday is coming up, and you would like to avoid spending too much money on a present for him, so you know just what to get – a handmade chessboard! All you have to do is buy a bunch of pieces of wood, glue them together into an 8x8 grid, and paint them black and white.

This week, your local hardware store is holding its annual lumber sale; if you buy any one piece of wood, you receive as many pieces identical to that one as you would like for free. You think that it is possible to get away with buying just one type of piece and using some number of copies of that piece to form a chessboard without even needing to cut the pieces. Only one question remains – which piece should you buy?

The Problem:

Given a piece of wood, determine whether or not you can place copies of this piece in an 8x8 grid to fill every square in the grid. The pieces can be rotated, but they may not overlap with each other or hang over the sides of the grid. Furthermore, they cannot be flipped over. Assume that the store has an infinite supply of each piece in stock.

The Input:

The first line contains a single, positive integer, *p*, representing the number of pieces to consider. For each piece, the following information is given:

First is a line containing a positive integer, *n* ($1 \leq n \leq 64$), denoting the number of chessboard squares used to display the piece.

This is followed by *n* lines containing *n* characters each. Among these *n* x *n* characters, exactly *n* of them will be "`*`" characters and the remainder will be "`.`" characters. The shape of the piece is the shape of only the "`*`" characters and the "`.`" characters are added only for clarity. The "`*`" characters will form a connected region (any "`*`" can be reached from any other "`*`" by moving only up, down, left, and right along other "`*`" characters).

The Output:

For each piece, output a single line "`Piece #`*`p`*`: `*`w`*" where *p* is the number of the piece being considered (in the order they were given in the input, starting from 1), and *w* is "`YES`" if it is possible to make a chessboard with that piece and "`NO`" otherwise.

Sample Input:

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

Sample Output:

```
Piece #1: YES
Piece #2: NO
```

Corn Maze

Filename: `maze`

Andrew loves corn, and he loves corn mazes even more! The problem is, Andrew is not so good at mazes and he gets stuck all of the time. However, while shucking corn the other day, he came across a revelation that if he always walked along the right wall of corn, he might always reach the end. His strategy is the following:

- Andrew starts his journey facing into the maze
- Every 5 seconds, he moves one space (the time it takes him to move is negligible). He only moves to a space if there is no corn wall there.
- When it is time to move, he chooses only one move and does so in the following priority relative to how he is currently facing:
    1. He will first try to move to his right.
    2. If he can't move right, he will then try to move in front of him.
    3. If he can't move to his front, he will then try to move to his left.
    4. If he can't move to his left, he will then try to move behind him.
- When moving, he is now facing that direction (If he moved to the left, he then is facing left of wherever he was facing before).
- When he reaches the end of the maze, he is done and stops moving.
- If at any time he reaches the start of the maze again or cannot move, he gives up and decides that the maze must be impossible.

The Problem:

Your task is to create a program that, given a layout of a corn maze, will tell Andrew whether or not his strategy will result in him finishing the maze. If he will eventually reach the end, you must determine how many seconds it will take him.

The Input:

The first line of the input will contain a single, positive integer, *n*, representing the number of mazes to evaluate. Each maze description will begin with a line containing two positive integers, *r* and *c* ($3 \leq r \leq 1{,}000$; $3 \leq c \leq 1{,}000$). These represent the number of rows and columns of the maze, respectively. The following *r* lines each will contain *c* characters representing the maze. Corn walls are represented by "#", empty spaces are represented by a "`.`", and the start and end of the maze are represented by "`S`" and "`E`", respectively. The outer border of the maze will all contain "#" except the start and the end (both of which will lie on the border and will not lie in any of the four corners). There will be exactly one start and one end, and they are guaranteed to not be adjacent. It is also guaranteed that in nowhere in the maze exists a 2x2 square of "`.`"s.

The Output:

For each maze given, output "`Maze #`*`i`*`: `" where *`i`* is the maze's number in the input (starting from 1 in the order given in the input), followed by either "`Impossible`" if Andrew's strategy will never result him in reaching the end or "*`t`* `seconds`" where *`t`* represents how long it will take Andrew to reach the end of the maze (in seconds) if it is possible.

Sample Input:

```
2
10 10
########E#
#.....#..#
##.##...##
#..#..#..#
#.##.#####
#.#......#
#.######.#
#....#...#
#.##.#.#.#
#S########
5 5
#E###
#.#.#
###.#
#...#
#S###
```

Sample Output:

```
Maze #1: 300 seconds
Maze #2: Impossible
```