Twenty-fourth Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
I Love Appliances!	wakkienunu
It's All Good	good
XKCD	xkcd
Choose Your Weapons & Warriors	weapons
Outline Builder	outline
Balls, Bins & Lollipops	balls
Montgomery Anaconda: The Life of Ryan	ryan
Egg Drop	eggdrop
Tentaizu!	tentaizu
Elementary Cellular Automata	automata
Choose Your Own Adventure	adventure

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

For example, if you are solving Choose Your Own Adventure:
Call your program file: adventure.c, adventure.cpp or adventure.java
Call your input file: adventure.in
Call your Java class: adventure

I Love Appliances!

Filename: `wakkienunu`

Throughout Central Florida there is a guy named Sam, infamous for selling his appliances in commercials and infomercials. He has gained quite a following with his signature exclamation, "I love appliances!", and his prowess at selling features of the appliances like white porcelain or rubber wheels. Indeed, his commercials can be found on popular online video sites, and even a series of parodies has been created!

Obviously, Sam gets very excited about appliances. He wants to offer the best prices (never retail!) over the competition. He needs your help in comparing the prices.

The Problem:

Given the price of a model of appliance, determine whether it is cheaper than full retail price.

The Input:

Input will begin with a single, positive integer, *n,* on a line by itself, representing the number of appliance types to check. Each appliance type will start with the name of the appliance (a single word of at most 20 letters) followed by the retail price in whole dollars, separated by a single space. The next line will contain a positive integer, *m*, representing the number of appliance models within the current appliance type. The following *m* lines will each contain a price of that model.

The Output:

For each appliance type, output a line reading "`Appliance g:`" where *g* is the name of the appliance type exactly as given in the input. Then for each model of that appliance type, determine how its price compares to the retail price. If the model price is less than the retail price, then output the line "`I love appliances!`" If the model price is instead greater than retail, output the line "`You paid too much!`", and if it is exactly equal to retail, output "`Wakkie Nu Nu!`" The output for each appliance type should be followed by a blank line.

(Sample Input and Sample Output on following page)

Sample Input:

```
3
Dishwashers 100
2
95
125
Dryers 135
2
150
100
Microwaves 125
1
125
```

Sample Output:

```
Appliance Dishwashers:
I love appliances!
You paid too much!

Appliance Dryers:
You paid too much!
I love appliances!

Appliance Microwaves:
Wakkie Nu Nu!
```

It's All Good

Filename: `good`

Jimmie Flowers, known as Agent 13, is back! However, he has a secret that until now nobody has noticed (although we aren't sure how we missed it!). Jimmie can't stand to have objects that are not properly aligned. If any object is slanted, he feels compelled to adjust that object.

Jimmie needs your help, though! He wants you to take bricks (which we will represent by just one of their edges) and determine if they are slanted or not.

The Problem:

Given two unique points on a line, determine if the line is a horizontal or vertical line.

The Input:

Input will begin with a single, positive integer, *n,* on a line by itself, representing the number of objects. For each object, there will be four non-negative integers, x_1, y_1, x_2 and y_2 (all ≤ 1000), on a single line each separated by a single space where (x_1, y_1) represents one point on the edge and (x_2, y_2) represents a second (and different) point on the same edge.

The Output:

For each object, if it is slanted (not horizontal or vertical), output "`We need to fix this`" or output "`It's all good`" if it is not. Each output should be on a separate line.

Sample Input:

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

Sample Output:

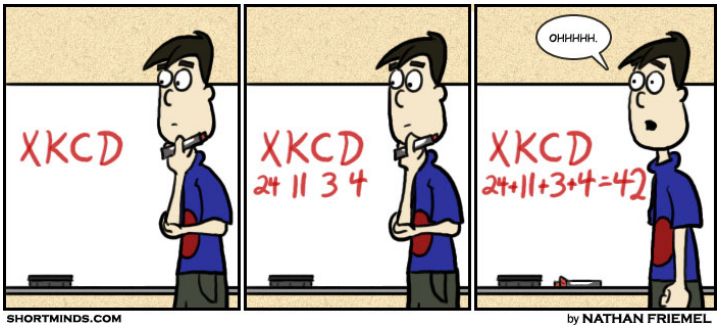
```
We need to fix this
It's all good
```

XKCD

Filename: `xkcd`

Matt is a fan of the web comic XKCD. Being such a fan he would like to come up with an online name similar to xkcd (he wants to keep his lower-case). Your job will be to help Matt come up with a name similar to xkcd.

Looking at this name it can be seen that taking the sum of the ordinal's of each letter in the word totals to 42 or, as we know, the answer to life, the universe and everything. Ordinals are defined as the 1-based index for each letter in the alphabet (a=1, b=2, c=3, …, z=26).



Matt also remembered watching an interview with Randall Munroe (the creator of xkcd) and learning one of the requirements of the online name he chose for himself: "xkcd" (xkcd the comic was later given this same name). Randall required that his name be non-pronounceable. Therefore, Matt would also like his name to be non-pronounceable as well. To ensure this, Matt would like to choose a name that does not have any vowels in it. To be extra careful he would like to include the letter "y" as a vowel.

Lastly, Matt noticed that the values of each of the ordinals of the letters in "xkcd" are strictly decreasing if you swap the last two letters (i.e., xkcd → xkdc where x=24 > k=11 > d=4 > c=3). To make his name even more similar to xkcd, Matt would like his name to have a strictly decreasing order when the last letters are swapped.

Given these rules, Matt would like to see what names he could possibly choose from a list of names that meet these requirements. Even though he is a huge fan of XKCD, he still wants to be unique, however. He has decided to have one major difference in his chosen name. He would like to be able to look at names of different lengths that meet his requirements. Your job is to write a program that takes in a length desired and prints out all the names of that length in alphabetical order that meet his XKCD-like requirements.

The Problem:

Given a name with a particular length, print out all names of that same length whose ordinals for each character add up to 42, are strictly decreasing when the last letters are swapped and are non-pronounceable (meaning that it contains no vowels including "y").

The Input:

The input will be lines containing single positive integers, n ($2 \leq n \leq 42$), representing the length of the names that you must print out that are "xkcd-like." The number 42 indicates end of input.

The Output:

For each length requested in the input, output "`XKCD-like name(s) of length: n`" where *n* is the length given in the input. On the following lines, output all names (in alphabetical order) that are XKCD-like that are also in the universe of *n* letter strings. If there are no such strings for the input length, output the phrase "`Mostly Harmless`" instead. Note that the end of input marker of 42 should not be processed normally; instead, you should output the phrase "`The answer to life, the universe and everything!`" and end.

Sample Input:

```
2
6
28
42
```

Sample Output:

```
XKCD-like name(s) of length: 2
pz
rx
sw
tv
XKCD-like name(s) of length: 6
kjhfcd
kjhgbd
ljgfcd
ljhfbd
ljhgbc
lkgfbd
lkhfbc
lkjdbc
mjgfbd
mjhfbc
mkgfbc
mlhdbc
nhgfcd
njgfbc
nkhdbc
nlgdbc
nmfdbc
phgfbc
pjgdbc
pkfdbc
qjfdbc
rhgdbc
shfdbc
tgfdbc
XKCD-like name(s) of length: 28
Mostly Harmless
The answer to life, the universe and everything!
```

Choose Your Weapons & Warriors

Filename: weapons

Dexter loves to play fantasy tabletop RPGs, and his favorite of all is Advanced Combat & Magic (AC&M). Nothing is better than picking up dice, gathering with friends, planting his big sharp weapon into the skull of the nearest enemy monster, and taking its loot. Of course, when killing a monster, he sometimes has a hard time choosing his weapon. Each weapon comes with its own to-hit bonus, and its own damage dice and damage bonuses. The rules of combat are such: when attacking a monster, Dexter has to roll a 20-sided die numbered from 1 to 20 with each number having an equal chance of appearing and add his to-hit bonus to the result. If the result meets or beats the monster's Base Armor (BA), then it's considered a hit, and he rolls damage. Otherwise, no damage is dealt. There are two exceptions. If he rolls a 1 on the 20-sided die, it automatically counts as a miss. If he rolls a 20, it's an automatic hit and is considered a "critical" hit; in this case, he automatically rolls the maximum on each damage die. Damage is given in this format: *x*d*y*+*z* where *x* is the number of dice you roll for damage, d*y* is the type of dice you roll for damage where *y* is the number of sides on the dice, and *z* is the extra damage that is added on top of the rolled damage. For example, 4d6+3 means that, if he hits, he rolls 4 six-sided dice and adds 3 extra damage to the sum of the numbers rolled.

Let's say Dexter is doing battle with a Bloodrage Orc Berserker who has an BA of 14 and Dexter is wielding a mighty battle axe with a to-hit bonus of 6 and his damage is 2d4+5. Dexter would need to roll an 8 or higher on his 20-sided die to hit it. If he rolls an 8 to 19, he would roll 2 4-sided dice, sum them up and add 5 for his damage. If he rolls a 20, he would deal 13 damage automatically. The average damage on each individual 4-sided die is 2.5, so on a normal hit, his average damage is 10. When taking into account misses, which deal an average of 0 damage, normal hits, which deal an average of 10 damage, and critical hits, which deal an average of 13 damage, his average damage per swing becomes 6.65.

The Problem:

Given the specifications for two weapons and the monster's BA, he wants to know which weapon has a greater average damage per swing and is, thus, more suited for slaying this particular monster.

The Input:

There will be multiple weapon comparisons. Each weapon comparison will begin with a positive integer no greater than 1000 on a line by itself, detailing the monster's BA. The next two lines will contain the information for the 2 weapons to be compared. Each weapon will come in the form of "*n h* x*d*y+*z*" where *n* is a unique string containing the name of the weapon (of no more than 20 alphabetical characters), *h* is an integer ($0 \le h \le 100$) indicating the weapons to-hit bonus and *x*d*y*+*z* indicates the weapon's damage ($1 \le x \le 100$, $2 \le y \le 100$, and $0 \le z \le 1000$). The last line of input will contain only the digit 0 and should not be processed.

The Output:

For each pair of weapons, determine the average damage per swing of each weapon and output on a line by itself "n_1 `is better than` n_2`.`" if one weapon has a greater expected damage than the other where n_1 is the name of the superior weapon and n_2 is the name of the inferior weapon. If the weapons are equivalent, output "`They're both the same.`" Two weapons are equivalent if their average damage differ by no more that 10^{-7}.

Sample Input:

```
15
Longsword 3 1d8+5
Spear 2 1d10+6
18
Dagger 5 1d4+2
Greatclub 1 1d6+4
21
Greataxe 1 1d12+13
Hammer 8 2d2+0
0
```

Sample Output:

```
Spear is better than Longsword.
Dagger is better than Greatclub.
They're both the same.
```

Outline Builder

Filename: `outline`

Stephen's cute li'l kitteh cat haz shredded his notes for class! But seeing this was even more funnier than a dog eating your homework—in fact Stephen laughed out loud. Then he took a picture of his "slasher cat" napping on the shreddings, and posted it the interwebs.

Now it's back to work…and hopefully back to better grammar. Stephen has to turn in outlines of all his research before he writes the actual papers. He can remember bits and pieces of his notes—specifically, how the outline topics relate to each other—and he has reconstructed one outline about Neal Stephenson books and one about classic cartoons. He needs your help to do the rest.

The Problem:

Given a list of pairs of outline topics—each pair being a subtopic and the topic it belongs under—present a nicely-formatted outline.

The Input:

The input will contain data for multiple outlines that need to be constructed. The data for each outline starts with a line containing only a positive integer, n ($n < 300$). The next n lines each contain exactly one subtopic and the topic it belongs under, in the form:

`subtopic/topic`

The `subtopic` and `topic` will both start with a letter, and both be limited to 40 characters. Each may contain any characters other than the "`/`" separator, but will not contain more than one consecutive space, nor end with a space.

Note that the `subtopic` on one line of the input might appear—identically, character for character, with letters case-sensitive—as a `topic` in one or more other input lines for the same outline, since it might have subtopics of its own. Each topic for an outline is guaranteed to appear in that final outline exactly once.

The data for the last outline will be followed by a line containing only a zero.

The Output:

For each outline, output the message "`Outline #`*m*" where *m* is the number of the outline from the input, starting at 1.

Beginning on the next line, output the outline. Show all the top-level topics (the ones that aren't subtopics of any other topics) in sorted ASCII order (where capital Z comes before lowercase a), with each on its own line, and starting at the beginning of the line. Starting on the line immediately after each topic, show its subtopics—also ASCII sorted—on separate lines and indented 4 spaces from where that topic starts. For example, a subtopic of a subtopic of a top-level topic will be indented 8 spaces, and its subtopics, in turn, will be indented 12 spaces.

Output a blank line after each outline.

Sample Input:

```
16
Characters/Anathem
WW2/Cryptonomicon
sphere/avout
nanotechnology/Diamond Age
Erasmas/Characters
bolt/avout
Jad/Characters
avout/Vocabulary
TL;DR/Baroque Cycle (3 books)
concent/Vocabulary
Cord/Characters
cord/avout
Arsibalt/Characters
dot-com/Cryptonomicon
Neo-Victorian/Diamond Age
Vocabulary/Anathem
5
Betty/Rubble
Fred/Flintstone
BAMM-BAMM/Rubble
Wilma/Flintstone
Barney/Rubble
0
```

Sample Output:

```
Outline #1
Anathem
    Characters
        Arsibalt
        Cord
        Erasmas
        Jad
    Vocabulary
        avout
            bolt
            cord
            sphere
        concent
Baroque Cycle (3 books)
    TL;DR
Cryptonomicon
    WW2
    dot-com
Diamond Age
    Neo-Victorian
    nanotechnology

Outline #2
Flintstone
    Fred
    Wilma
Rubble
    BAMM-BAMM
    Barney
    Betty
```

Balls, Bins & Lollipops

Filename: `balls`

At a recent trip to the state fair, Ali fell in love with a carnival game called Balls and Bins. In this carnival game, there is a single line of bins, and you are given balls to throw into the bins. If Ali gets exactly one ball in each bin (regardless of the order in which he makes them), he wins a lollipop[1]! Otherwise, he loses.

Of course, this is a hard game to win; however, Ali has a distinct advantage over everyone else. He is unnaturally lucky, and has found that no matter what he does, every ball he throws makes it into a bin. Unfortunately, he seems to have no control over which bin a ball lands in, and each ball he throws is equally likely to land in any of the bins (including any bins that already contain balls). Naturally, he begins to wonder what his odds are of winning the game.

The Problem:

Given the number of bins and balls, determine the probability that Ali will win the game, given his special ability. Since the game may change from day to day, your program should be able to determine the probability of winning for multiple numbers of bins.

The Input:

There will be multiple games in the input file. The first input line contains a positive integer, t, indicating the number of games to be processed. This will be followed by t games. Each game will consist of a single line containing only a single integer, n ($1 \leq n \leq 20$), representing the number of balls and bins.

The Output:

At the beginning of each game, output the line "`Balls and Bins Game #`x`:` p", where x is the game number (starting from 1), and p is the probability of winning the game as a percentage between 0 and 100, rounded to 6 decimal digits (for example, 2.4876935 should be rounded to 2.487694 and 2.4876934 should be rounded to 2.487693). Leave a blank line after the output for each game.

(Sample Input and Sample Output on following page)

[1] A circular one, of course.

Sample Input:

```
4
1
2
3
7
```

Sample Output:

```
Balls and Bins Game #1: 100.000000%

Balls and Bins Game #2: 50.000000%

Balls and Bins Game #3: 22.222222%

Balls and Bins Game #4: 0.611990%
```

Montgomery Anaconda: The Life of Ryan

Filename: `ryan`

Ryan, a Trojan, hates the ancient Greeks. It has never been the same after that incident with the wooden horse. He hates them so much that he joined a fellowship that dislikes the ancient Greeks just as much as he called the United Coalition of Fellows. Really, what have the Greeks done of worth, besides inventing coined money, typography, the theater, development of medicine and mathematics, and the Olympic games? Nothing! That's what! The only thing that the United Coalition of Fellows hates more than the ancient Greeks are the other ancient Greek-hating splinter groups: the Universal Fellowship, the Fellows Standing United, and the Gatherers in Togetherness (who don't even have fellows in their name, but now we're getting sidetracked). Surely something must be done to show those ancient Greeks that they are not welcome here in Troy. Ryan has the perfect plan. He will graffiti the walls with a message telling them to go back home.

That night, Ryan put his plan into action, but was soon caught by Greek soldiers while he was painting "greeks get going to home." The soldiers were outraged and now it's time for Ryan's punishment. They cannot let such offensive writings be placed upon their wall. They cannot let bad grammar go unpunished either. For one, Greeks is a proper noun and should be capitalized, and there is hardly a need for present participle phrasing when a simplistic "go" should suffice for graffiti. Also, "home" is an object and should not be preceded by "to". Now, "Greeks" should be separated by punctuation since it's an interjection, and it's obvious that Ryan feels strongly about the message, so they should be separated by an exclamation mark. Furthermore, the "Go home" message also needs to end with an exclamation mark. This would make the final message "Greeks! Go home!"

As punishment for this horrendous grammar, Ryan must write this improved message on the wall several times. As Ryan gets to work, he keeps making mistakes, which the guards won't count, and he loses track of how many times he has to write the message on the wall. So he's asked you to write a program for his abacus, to keep track of how many times he has written the phrase correctly and tell how him much more he has to finish.

The Problem:

Given the writings of Ryan, determine how many copies of the message are left to do in Ryan's punishment.

The Input:

There will be several disciplinary action sessions. Each session will begin with a line containing only a positive integer, x ($x \leq 100$), stating the number of times he's required to write the message as desired by the soldiers. Each line following will have a string of characters (of no more than 70 characters), representing Ryan's attempt to write the phrase as the Greek soldiers require. There will be no leading or trailing spaces, and no more than one space between two non-space characters. The session will end with the line "`Am I done yet?`" The last line of the input will be a line containing only the digit 0, which should not be processed.

The Output:

For each disciplinary session, output the line "`You have` *n* `left to go.`" if Ryan still has to write more where *n* is the number of times he still needs to write the phrase. Otherwise, output the line "`You're done. Now don't do it again.`" if Ryan is finished.

Sample Input:

```
5
Greeks! Go home!
Greeks! Go home!
greeks get Go home!
Greeks! Go home!
Am I done yet?
2
greeks, get going.
Greeks! Go home!
Greeks! Go home!
Am I done yet?
0
```

Sample Output:

```
You have 2 left to go.
You're done.  Now don't do it again.
```

Egg Drop

Filename: `eggdrop`

UCF has decided to bring back its high school contest Physics Olympics. Although your specialty is Computer Science, the physics kids are having trouble determining the maximum height from which an egg can be dropped while encased in their contraption without breaking. Your physics friends have given you the following formulas

$$d = 4.9t^2,\ v = 9.8t$$

where d = distance in meters, v = velocity in meters/second, and t = time in seconds. These formulas calculate the distance the egg has fallen t seconds after being dropped from a state of rest and the velocity at that time. Your friends have been able to figure out the maximum velocity their design can withstand. Given these pieces of information, they have asked you to write a computer program that calculates the maximum whole number of meters from which an egg can be dropped in their contraption without breaking. (Note: These formulas are only valid on Earth, where near the surface we experience an acceleration of 9.8 m/s^2.)

The Problem:

Given the maximum velocity a design can withstand, determine the maximum number of meters from which the egg can be dropped and not break.

The Input:

The first line of the input file will contain a single positive integer, *n*, representing the number of contraptions to be analyzed. The next *n* lines have one positive integer each less than 500, representing the velocity in meters/second that particular design can prevent an egg from breaking. None of these velocities will result in a maximum distance in meters that is within 0.01 of an integer.

The Output:

For each egg dropping contraption, output a single statement with the following format:

`Egg Drop #`*`k`*`: The maximum height is `*`x`*` meter(s).`

where *k* is the egg drop number and *x* is the maximum *whole number of meters* the egg can be dropped without breaking.

(Sample Input and Sample Output are on the following page)

Sample Input:

```
2
5
10
```

Sample Output:

```
Egg Drop #1: The maximum height is 1 meter(s).
Egg Drop #2: The maximum height is 5 meter(s).
```

Tentaizu!

Filename: `tentaizu`

Tentaizu is a Japanese game whose name means "celestial map." Some people will try to tell you this game is the same as Minesweeper. These people are incorrect.

The game is played on a 7x7 board. Exactly 10 of the 49 squares are each hiding a star. Your task is to determine which squares are hiding the stars. Other squares in the board provide clues: A number in a square indicates how many stars lie next to the square—in other words, how many adjacent squares (including diagonally adjacent squares) contain stars. No square with a number in it contains a star, but a star may appear in a square with no adjacent numbers.

Figure 1 is an example of an initial game board, and Figure 2 is the solution.

1					**3**	
	1			**0**		
			2			
	3					**3**
				1		
			1			**1**

Figure 1

1	*****			*****	**3**	*****
						*****
	1			**0**		
		*****	**2**			*****
	3	*****			*****	**3**
		*****		**1**		*****
			1			**1**

Figure 2

The Problem:

Given the description of a 7x7 Tentaizu board, your task is to find the squares containing the 10 hidden stars. You are guaranteed that each given Tentaizu board will have a unique solution.

The Input:

There will be multiple Tentaizu boards in the input file. The first input line contains a positive integer, *t*, indicating the number of Tentaizu boards to be processed. This will be followed by *t* Tentaizu boards. Each Tentaizu board will contain 7 lines, and each line will contain exactly 7 characters. Each character will be a digit from '0'-'8' or a '.' to indicate an empty square (which may be hiding a star). Each Tentaizu board will be separated by a single blank line.

The Output:

For each Tantaizu board, output the line "`Tentaizu Board #`*x*`:`" where *x* is the board number (starting from 1). Then, print the solved Tentaizu board in the same format as the input, but with a '`*`' at each of the 10 star locations. Leave a blank line after the output of each board.

Sample Input:

```
2
1....3.
.......
.1..0..
...2...
.3....3
....1..
...1..1

...1.3.
12....2
.2.1...
...31..
...2..3
..1....
.......
```

Sample Output:

```
Tentaizu Board #1:
1*..*3*
......*
.1..0..
..*2..*
.3*..*3
..*.1.*
...1..1

Tentaizu Board #2:
...1*3*
12...*2
*2*1...
...31.*
..*2*.3
..1..**
.......
```

Elementary Cellular Automata

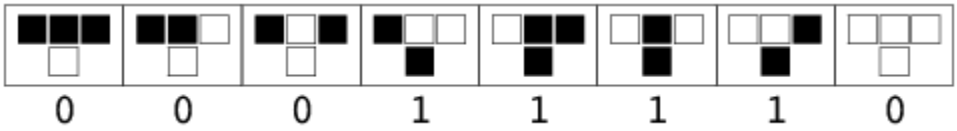
Filename: `automata`

Cellular automata are an interesting application of computer science, simulating out processes by creating local rules on a grid with individual "cells." Each unit of time, or "generation," the cells change based on a set of rules, and the process is repeated on the new cells.

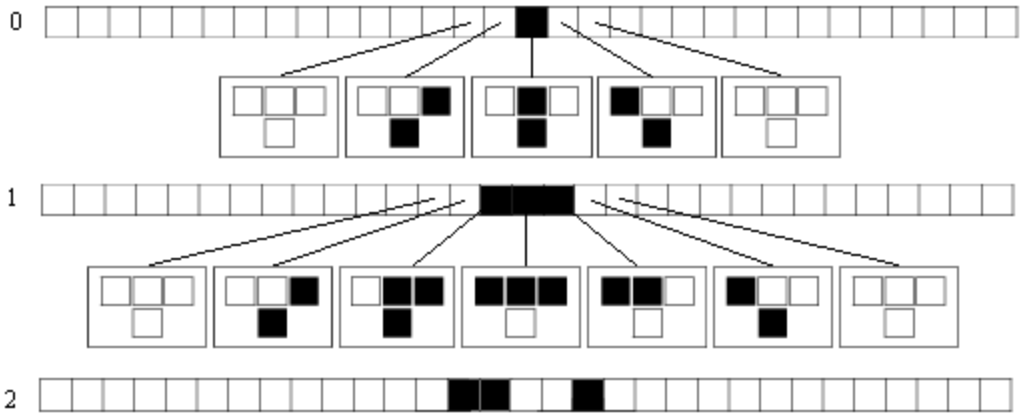
The simplest kind of cellular automata are one-dimensional or elementary cellular automata. The idea is that you start with a certain number of cells, each of which can be "dead" or "alive" arranged on a Möbius strip (i.e., it wraps from the end back to the beginning), and during each generation, consider cell and the states of its left neighbor, itself, and its right neighbor to decide the cell's state for the next step. Being the savvy mathematician, you know that means that for any given cell, there can be 2^3, or 8, combinations as shown in graphical form below.

For each of those 8 combinations, the result can be either on or off as a result, depending on the rules of the automata. This means that there are 2^8, or 256, different elementary cellular automata, one for each possible result of each possible state.

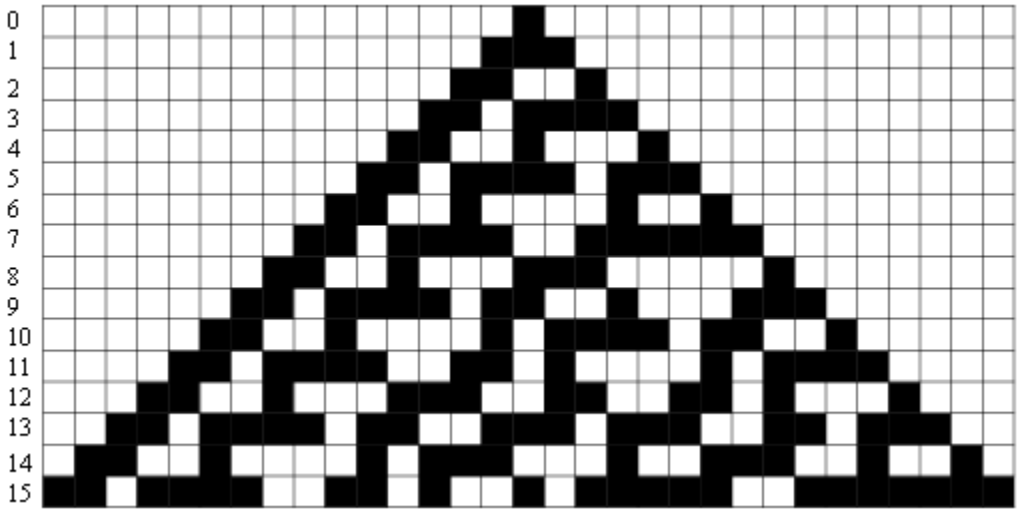
For example, here is one possible rule set, with each triplet of squares representing a state (the middle cell is the one being considered), and the square below it representing a result, with a white square being "dead" or "0", and a black square being "alive" or "1".



For example, here is two generations of an automata using the above rules, starting with a single cell.



Showing the original generation and fifteen generations at the same time yields an interesting triangular pattern:



The Problem:

Given the initial state of a cellular automata and a rules set, print a two dimensional text image representing the progression of the states over a number of generations.

The Input:

The first line of the input will contain a positive integer, t ($t \leq 1000$), of automata to follow. Each automata will consist of a pair of integers, g ($0 \leq g \leq 100$) and w ($3 \leq w \leq 100$), followed by two strings on separate lines consisting of only zeroes and ones. g is the number of generations that should be simulated and w is the number of cells in the simulated space.

The first string will be exactly w characters long and will represent the initial state of the space. A "0" denotes a dead cell, while a "1" denotes a living cell. The second string will be exactly 8 characters long and represents the rule set for an automata, given in the order shown in the picture on the previous page. A "'1" will result in a live cell, and "0" means that the state will result in a dead cell.

For example, "00011110" will result in a live cell if the current cell and the cell to the right is alive, or only a single of the three cells (left, current, and right) are alive. All other configurations will result in a dead cell.

The Output:

For each automata, output a header "`Automata #a:`" where a is the number of the automata, starting from 1, on its own line. Starting from the next line, output the initial state of the grid, followed by g states, each on their own line, of the board as a series of characters. A living cell should be denoted by an octothorpe ("#") and a dead cell by a period (".") . Each automata should be followed by a blank line.

Sample Input:

```
3
0 5
01010
11111111
3 9
000010000
00011110
3 6
010010
00010000
```

Sample Output:

```
Automata #1:
.#.#.

Automata #2:
....#....
...###...
..##..#..
.##.####.

Automata #3:
.#..#.
..#..#
#..#..
.#..#.
```

Choose Your Own Adventure

Filename: `adventure`

When you were a kid you used to read the Choose Your Own Adventure series, where, on each page, you had a question to answer. Based on your answer, you were told to go to one page or another page for the story to continue. This continued until you reached a page with an ending. The books were always arranged so that you always started on page one and you could never read the same page twice in a story. If that were possible, the story could possibly never end. You loved the suspense of these books, but were frustrated that you paid good money for them and usually only read about 5 pages before you were done with the whole story! Now that you've taken some computer science in high school, you've decided to dig those books out of the attic to see if you can find the longest possible adventure each book has to offer, so you can finally get your money's worth!

The Problem:

Given a list of each page in a Choose Your Own Adventure novella and which page(s) each page forwards to, determine the longest possible story (in number of pages) that the user could possibly read, following the proper rules of the book.

The Input:

There will be multiple books in the input file. The first input line contains a positive integer, n, indicating the number of books to be processed. The details about each book follow. The first line of each book description contains a positive integer, p ($1 < p < 1000$), representing the number of pages in the book. The following p lines will contain information about each page (from 1 to p). Namely, those pages will list each page to which the current page forwards. If the page does not have an ending on it, there will either be one or two forwarding page numbers listed. If there are two pages, the two numbers will be separated by a single space. If the page is one that has an ending, then the line will read "`ENDING`". The forwarding pages will be set up so that no page may ever be read twice in a single story.

The Output:

For each book, output a single line with the following format:

`Book #`*`k`*`: The longest story is `*`x`*` pages.`

where k is the book number, starting at 1, and x is the number of pages in the longest possible story for that book.

Sample Input:

```
2
3
2 3
ENDING
ENDING
5
4 5
ENDING
2
3
4
```

Sample Output:

```
Book #1: The longest story is 2 pages.
Book #2: The longest story is 5 pages.
```