Twenty-eighth Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
Fishy Business	fishy
Journey to Batlantis!	batlantis
The Children of the Brood	brood
Dueling Progress Bars	bar
Dinner Games	dinner
The Diagonal Crease	crease
Anti-absolute Values	absolute
My Spidey Sense is Tingling!	tingling
Dragon Fire	fire
Menger Sponges	sponge
Ant Tunneling	ant

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

For example, if you are solving Fishy Business:
Call your program file: fishy.c, fishy.cpp or fishy.java
Call your input file: fishy.in
Call your Java class: fishy

Fishy Business

Filename: `fishy`

Your good friend Nikolai owns the best fish stand in all of Russia! Some call it a "fishy" business, but Nikolai has very convincing Russian friends that tell customers otherwise[1]. Nikolai gets so much business that he needs your help to calculate his earnings! And you REALLY don't want to let your good friend Nikolai and his Russian friends down…

Each morning Nikolai fills his fish stand with various fish and fish parts. Unfortunately, Nikolai's customers are only interested in purchasing whole fish (he's hoping the fish parts side of the business will take off one day). Nikolai sells fish of many sizes, but these fish always have a very similar shape. Nikolai's fish always consist of a head "`<o`", followed by one or more torso segments "`(`" always facing towards the head, followed by a tail "`><`". The fish can also be laid in the opposite direction, and, thus, would consist of a tail "`><`", one or more torso segments "`)`", followed by a head "`o>`".

Some examples of whole-fish include:

```
><))o>          ><)))))))o>          <o((><          <o(((><
```

Some examples that are *not* whole-fish include:

```
<o)))><          ><((()))o>          ><o>          )
```

When a customer wants to purchase a specific fish, Nikolai's Russian friends cleave the head and tail off (while saying witty Russian jokes[2,3]) and give the torso unit(s) to the customer leaving the head and tail in the same positions on the stand. Nikolai charges the customer based upon the length of the torso. More specifically, the customer is charged the length of the torso squared (in rubles). For example, a torso of length 5 would cost 25 rubles.

Since the head and the tail remain on Nikolai's stand, customers may mistake an adjacent fish to be a whole fish. Thus, these customers don't end up with a fresh fish, but a fishy one! This doesn't bother Nikolai, because money is money and there are plenty more fish in the sea.

The Problem:

Each morning Nikolai fills his stand with fish and various leftover fish parts. Given the contents of Nikolai's stand on a given morning, calculate how much money Nikolai will receive by selling all of his "whole" fish.

---

[1] Nikolai's Russian friends are also great at feeding his fish! But are very bad at finding people that "disappear"…

[2] What was the Tsar of Russia's favorite fish? Tsardines!

[3] What do you call a fish with no eyes? A fsh!

The Input:

The first line of the input will begin with a single, positive integer, *d*, representing the number of days for which you need to compute Nikolai's income. For each day there will be a line containing one integer, *s* ($1 \leq s \leq 500$), representing the length of Nikolai's stand for that day, followed by a line containing a string of *s* characters from the set {`<`, `>`, `o`, `(`, `)`} representing fish and leftover fish-parts.

The Output:

For each day, output a single line "`Day #`*`i`*`: `*`x`*` rubles`" where *i* is the number of the day in the input (starting with 1) and *x* is the amount of money (in rubles) Nikolai will receive on that day.

Sample Input:

```
5
7
><)))o>
7
<o(((><
10
><((()))o>
11
<o((><)))o>
20
><))(><()o>><)o><><<<
```

Sample Output:

```
Day #1: 9 rubles
Day #2: 9 rubles
Day #3: 0 rubles
Day #4: 13 rubles
Day #5: 1 rubles
```

Journey to Batlantis!

Filename: `batlantis`

As a senior member of the Junior Exploratory League, it is your sworn duty to investigate any and all mysteries, riddles or conundrums that come across your desk. For days you've been puzzling over the battered journal of the late Dr. Eagle "Bones" Falconhawk, esteemed expert in the field of submarine-chiroptera archaeology. His nephew has reason to believe that this weathered tome holds the key to locating the fabled sunken city of Batlantis!

Dr. Eagle "Bones" Falconhawk
1979 - 2014
They never found the body.

What's perplexing is that, in an attempt to obfuscate his findings from rival aquabat historians, Dr. Falconhawk appears to have employed a strange encryption protocol. For example, when referring to the mythical city in his text, rather than "BATLANTIS" he instead uses the term "BAT ATLANTIS" (which, of course, is utter nonsense). In an entry describing a trip to the dentist, the jibberish "TOOTH HURTY" is repeatedly scribbled upon the page, which the good doctor's nephew has pointed out may, in fact, be an encoding of "TOOTHURTY" (a world-renowned dental and orthodontics chain). It seems as though the doctor has developed a masterful replacement scheme! If only there were a way to make sense of it all!

Although the doctor's encryption scheme is a tough one, it may be possible to reconstruct the source material through some clever observation. From what you can tell, a source word will always be replaced by exactly two words, *ab* and *bc*. Each of these words can be further divided into a prefix and suffix, the strings *a* and *b* for the word *ab* and the strings *b* and *c* for the word *bc*. Note that the suffix of *ab* and the prefix of *bc* are equivalent! Then, the original word can be constructed as *abc*, where *b* is as long as possible! There are encrypted pairs littered throughout the journal, so it'd be best to write a program to handle them all. Give it a shot!

The Problem:

Given an encoding as two strings, *ab* and *bc*, find and construct the original string *abc*, where the common string *b* is as long as possible. Note that there might not be a common string at all!

The Input:

The first line of the input will begin with a single positive integer, *t*, representing the number of encrypted journal entries to decipher. Each of the next *t* lines will contain an encoded word, given as two strings *ab* and *bc*, each composed of only uppercase letters and separated by a single space. Each word will be at least 1 and at most 100 letters long.

The Output:

For each journal entry, output a single line in the form "`Entry #i: abc`", where *i* is the journal entry number (starting at 1), and *abc* is the reconstructed journal entry in uppercase letters.

Sample Input:

```
3
BAT ATLANTIS
TOOTH HURTY
REVENGE VENGEANCE
```

Sample Output:

```
Entry #1: BATLANTIS
Entry #2: TOOTHURTY
Entry #3: REVENGEANCE
```

The Children of the Brood

Filename: `brood`

Ms. G is a mighty dragon at the mythical Uber Cool Flying school in the sky. She takes her brood of teen dragons out every morning for flying lessons. Each class flies in a line formation with mama dragon (Ms. G) in the middle. She noticed that many dragons of her brood form small social groups that disrupt her lessons. To prevent disruptions before they take off every morning, she assigns each broodling a place in line. She needs your help arranging the teens in a way that will not cause any disruptions during her flight lesson. A teen will cause a disruption if either of the adjacent teens to him or her are in the same social group. Ms. G only needs to know if any arrangement exists that will avoid disruptions. If one does exist, she can easily determine the actual arrangement on her own.

The Problem:

Given the group number of each dragon in Ms. G's brood, determine if there exists an order for the dragons that will not cause any disruptions during the flight lesson.

The Input:

The input begins with a line containing a single positive integer, *t*, representing the number of classes (Ms. G teaches multiple classes). This line is followed by *t* class descriptions. Each class starts with a line containing a positive integer, *n* ($n \leq 1{,}000$), representing the number of dragons in the brood. The next line contains *n* integers, representing the social group number of each dragon in the brood. The group number for each dragon will be within the range 1 to *n*. If the dragons cannot be divided into two evenly-sized groups, then they should instead be divided into two groups of size $(n - 1) / 2$ and $(n + 1) / 2$.

The Output:

The output for each class should start with "`Class #i: `" where *`i`* is the number of the class in the input (starting with 1), followed by "`YES`" if an arrangement exists and "`NO`" if it does not.

(Sample Input and Output are on following page)

Sample Input:

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

Sample Output:

```
Class #1: YES
Class #2: NO
Class #3: YES
```

Dueling Progress Bars

Filename: `bar`

Gabe and Aaron are competing against each other in speedruns of their favorite video game: *Trial of Dungeons 2*. They both record themselves playing the game and upload a video for each dungeon to YourTube with Gabe's footage on the left and Aaron's footage on the right for each dungeon in the game. Their previous videos have been quite confusing due to the non-linear style of the game's dungeons, so much so that the viewers don't even know who is in the lead at any given time! To give the viewers some idea of who is winning, Gabe and Aaron have decided to include a progress bar for each player in their videos. However, this could potentially give away the victor too early in video.

To add an element of suspense for the ending, they decided on a number of checkpoints for each dungeon. The progress bars are then split into segments of equal length, one for each checkpoint. The segments fill in sequence and the number of checkpoints completed at any given time correlate to the number of segments filled. The speed at which the progress bar for each player fills now only depends on the time it takes for that player to complete the next checkpoint.

Gabe and Aaron are interested in knowing how exciting the new videos will be. They define the excitement factor of each video as the number of times one player goes from losing to winning. A player is winning if their progress bar is currently filled strictly more than the other player's and a player is losing if their progress bar is currently filled strictly less than the other player's. For example, if Gabe is losing and then ties up with Aaron, the excitement factor does not increase. However, if Gabe were to at some later time surpass Aaron (before returning back to a losing state), then the excitement factor would increase by 1 (in other words, one player going from losing to tied to winning would cause an excitement increase of 1; one does not need to go straight from losing to winning to gain excitement). Similarly, if Gabe is winning, and then Aaron ties it up, but then Gabe again goes in the lead, the excitement factor again does not increase. Finally, of course, at the beginning of any given video both Gabe and Aaron are neither winning nor losing.

The Problem:

Given the times at which Gabe and Aaron complete each checkpoint in a dungeon, determine the excitement factor of the resultant video.

The Input:

The input will begin with a line with a single positive integer, v, representing the number of videos. Following this, there will be v video descriptions. Each description begins on a new line with an integer, c ($1 \leq c \leq 50$), representing the number of checkpoints for the current dungeon. Following this, there will be a line consisting of c non-decreasing integers (each from 1 to 1,000 and each separated from the next by exactly one space) representing the time in minutes from the start that Gabe completes each of the checkpoints. Then, on the next line there is an identically formatted line telling the times when Aaron completes the checkpoints.

The Output:

For each video, output a separate line with "`Video #`*x*`: `*e*" where *x* is the video number in the input (starting with 1) and *e* is the excitement factor for that video.

Sample Input:

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

Sample Output:

```
Video #1: 2
Video #2: 1
```

Dinner Games

Filename: `dinner`

John, Stephen, and Nick like to eat dinner together sometimes, but they've found one aspect of these dinners to be boring - paying the check! Splitting it evenly is simple: just divide by three. The most interesting part of this method is when the check doesn't split evenly and someone may have to pay a penny or two more than the others. Of course, this is still hardly interesting.

John has devised a game to make paying the check more fun! John will only pay with $10 bills, Nick will only pay with $5 bills, and Stephen will only pay with $2 bills (assume everyone has an ample supply of their respective bill type to be capable of paying more than the entire bill). They will each keep laying down bills in any order in a pile until that pile adds up exactly to the total on the check (luckily, they only go to restaurants that are priced so this actually works!). They are all smart, so they never make mistakes that force the sum over the total.

Now Nick is worried: how fast will this game get boring? Or, more specifically, given an amount representing the check total, how many different piles can be made with this method to sum up to exactly this amount? A pile is defined as different from another if there is either a different number of one or more of the bills or if the bills occur in a different order. Bills of the same denomination are indistinguishable, though. So, for example, all of the following piles are different from each other for various reasons: 2+2+5, 2+5+2, 5+2+2, 5+5+5, 2+5, 2+5+10, 10+10.

The Problem:

Given the amount of the check total, how many different ways are there for John, Stephen, and Nick to make piles that sum up to exactly that amount?

The Input:

The first line of input will be a positive number, *c*, the number of checks for which they want an answer. Following this will be *c* lines each describing one check. Each of these lines will contain an integer, *t* ($1 \leq t \leq 100$), representing the total amount of this check. There will be at least one way to pay each check using only the bills that John, Stephen, and Nick possess.

The Output:

For each check, output a line of the form "`Dinner #i: s`" where *i* is the number of the check in the input (starting with 1) and *s* is the number of different ways to sum up to exactly the check total using the method described above. *Note that this total could be quite large.*

Sample Input:

```
3
2
7
100
```

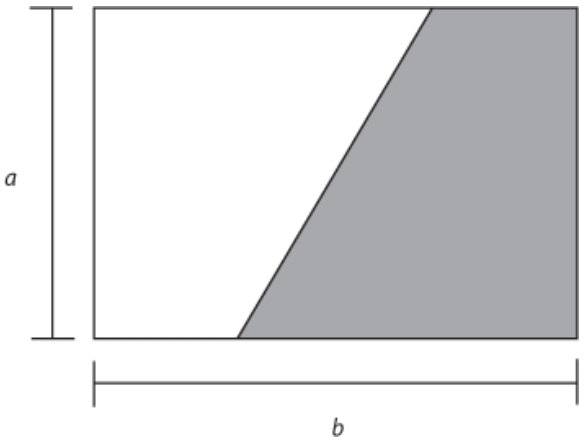
Sample Output:

```
Dinner #1: 1
Dinner #2: 2
Dinner #3: 9898980705
```

The Diagonal Crease

Filename: `crease`

One of Tyler's favorite ways of entertaining himself, other than making fractals, is folding paper into isosceles triangles, and then trying to make some funky isosceles triangle fractal creations. The first step in this process is always folding two opposite corners together. One day while stuck orbiting Earth in a shuttle with nothing other than food, water, and 5,006 sheets of paper with various side-lengths, Tyler decided to plan out a super cool 3D paper fractal design. However, in order to do this, he needs to know the length of the crease made when folding a piece of paper corner to corner. In the diagram below, if you folded the rectangle from its bottom-right corner to its top-left corner, the shaded portion would be laying over the non-shaded portion and the paper would be creased along the diagonal line in the middle.



The Problem:

Given the dimensions of a rectangular piece of paper, find the length of the crease formed when it is folded corner to the opposite corner.

The Input:

The input will begin with the number of sheets of paper to fold, *n*. Each of the next *n* lines will describe one sheet of paper and will consist of two integers, *a* and *b* ($1 \le a \le 100$; $1 \le b \le 100$), representing the height and width of the piece of paper, respectively, separated by a single space.

The Output:

For each sheet of paper, output a separate line containing "`Crease #`*`i`*`: `*`d`*" where *i* is the number of the sheet of paper in the input (starting with 1) and *d* is the length of the diagonal crease for that sheet rounded to two decimal places (where 0.034 rounds down to 0.03 and 0.035 rounds up to 0.04).

Sample Input:

```
2
8 11
13 5
```

Sample Output:

```
Crease #1: 9.89
Crease #2: 5.36
```

Anti-absolute Values

Filename: `absolute`

The absolute value of *x* is the non-negative value of *x* without regard to its sign. For example, the absolute value of 5 is 5, and the absolute value of -3 is 3. Calvin is really interested in a new kind of value that he has invented: the anti-absolute value! He has defined the anti-absolute value as the *negative* absolute value of *x*. For example, the anti-absolute value of 5 is -5, and the anti-absolute value of -3 is -3. Calvin wants to generate a new table of these values so he needs your help! Obviously, zero is ambiguous as an anti-absolute value, which Calvin dislikes, so he ignores it completely and will not include it at all.

The Problem:

Given an integer, output its anti-absolute value.

The Input:

The first line will contain a single, positive integer, *n*, representing the number of non-zero integers that Calvin needs processed. Following this, each of the next *n* lines will contain a single non-zero integer, *p* ($-1{,}000 \leq p \leq 1{,}000$; $p \neq 0$).

The Output:

For each integer Calvin needs processed, output "`Integer #`*`i`*`: `*`a`*" where *i* is the number of the integer in the input (starting with 1) and *a* is the anti-absolute value of the integer.

Sample Input:

```
2
5
-3
```

Sample Output:

```
Integer #1: -5
Integer #2: -3
```

My Spidey Sense is Tingling!

Filename: `tingling`

The Amazing Spider-Man was enjoying a nice evening with Gwen when it happened. That's right, his spidey sense was tingling! Of course, that can only mean one thing: trouble is afoot! The Lizard, the Rhino, Electro, or another villain in New York City must be up to some misdeed. It is the Amazing Spider-Man's noble duty to protect the city from these beastly baddies. Using his spidey sense, help Spider-Man locate and capture the notorious ne'er-do-wells.

The Problem:

The spidey sense is a very well tuned mechanism, providing Spider-Man with a great deal of information about impending trouble. When Spider-Man's spidey sense activates, he intuitively feels two things: an ID and a signal power. The ID is the name of the troublemaker triggering the tingle of the spidey sense. The signal power is the strength of the sense, which aids Spider-Man in finding the distance between himself and the malevolent meanie. If the sense activates with a signal power *p*, it means that the villain lies on the boundary of a circle with area p^2 square feet and centered at Spider-Man. Armed with this knowledge, determine the distance between Spider-Man and the savage scoundrel. If needed, use 3.141592653589793 for the value of π.

The Input:

Input will begin with a single line containing a positive integer, *n*, representing the number of spidey sense activations of which Spider-Man needs help. Each of the following *n* lines will contain an ID, *s*, followed by a single space and a signal power, *p*. The string, *s*, is at least 1 and at most 50 upper and lower case alphabetic characters, and *p* is a decimal value between 5 and 1,000,000 (inclusive), with exactly two digits after the decimal point. No ID will be present more than once.

The Output:

For each spidey sense activation, print a single line of the format "`s is r feet away.`" where *s* is the ID of the spidey sense activation and *r* is the distance between Spider-Man and the corresponding villain, in feet. The value of *r* should be output to exactly 3 places (rounded) after the decimal point (for example, an *r* of 7.0284 should be output rounded to 7.028, and an *r* of 7.0285 should be output rounded to 7.029).

Sample Input:

```
3
TheLizard 5.02
TheRhino 2014.00
Electro 123.45
```

Sample Output:

```
TheLizard is 2.832 feet away.
TheRhino is 1136.278 feet away.
Electro is 69.649 feet away.
```

Dragon Fire

Filename: `fire`

Andrew was playing the newest turn-based RPG Apples and Dragons. After playing for some hours, he has reached a level full of obstacles (and, of course, dragons!) that has proven to be absurdly difficult. Every now and then, the dragons spew out a torrent of fire which instantly results in death. Frustrated after many attempts, Andrew has started to study the level in depth and has observed a few facts.

The first fact is that these dragons have certain cues in their animation so Andrew can determine when they will spew out their flames of instant death. When they do so, all the dragons will also fire at once. The second fact is that not *all* of the squares on the map will be covered by the fire, meaning it is possible to survive the turn as long as Andrew doesn't have his character standing on a square that will be covered by fire when the dragons let loose! Lastly, Andrew has uncovered the shooting mechanism of dragons and how fire spreads. He has stored the mechanism as well as crucial game rules in the notes below:

1. The map is a two-dimensional rectangular grid composed of squares.

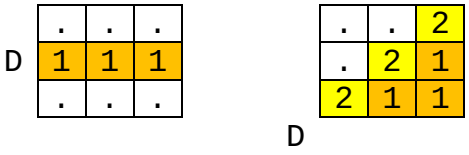
2. Each dragon occupies one square on the map. No two dragons may be on the same square.

3. Each dragon is facing either left, right, up or down at any point in the game.

4. There are two kinds of fire. Both kinds move quickly and spread to all affected squares instantly, up to the edge of the map.

5. The **first** kind of fire will spread only in the direction in which it is fired.

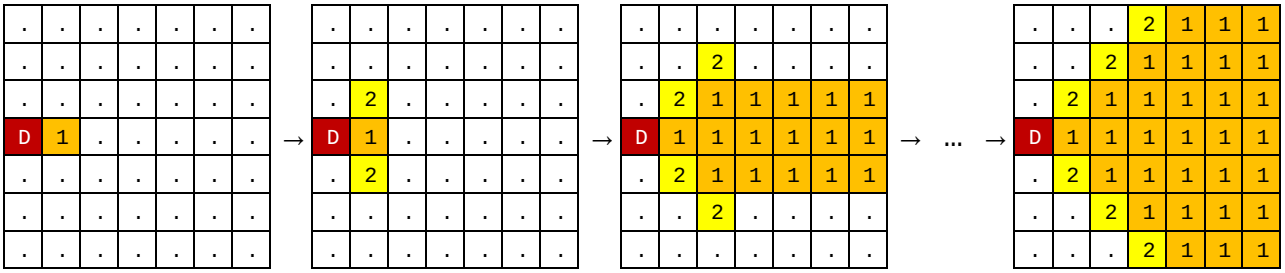
6. The **second** kind of fire will spread in two directions: a *primary* and a *secondary* direction. For each square of fire of the second kind, it will spawn another square of fire of the second kind in the *primary* direction. In addition, it will also spawn a square of fire of the **first** kind in the *secondary* direction.



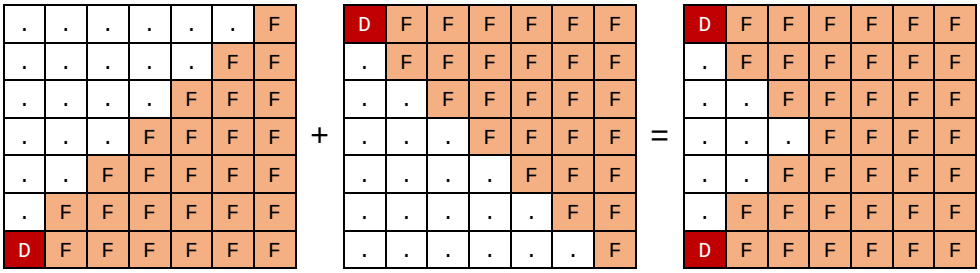
Left Diagram: Fire of the first kind, spreading to the right starting from the "D"

Right Diagram: Fire of the second kind starting at the D in the bottom left, spreading with a primary direction of upper right and secondary direction of right.

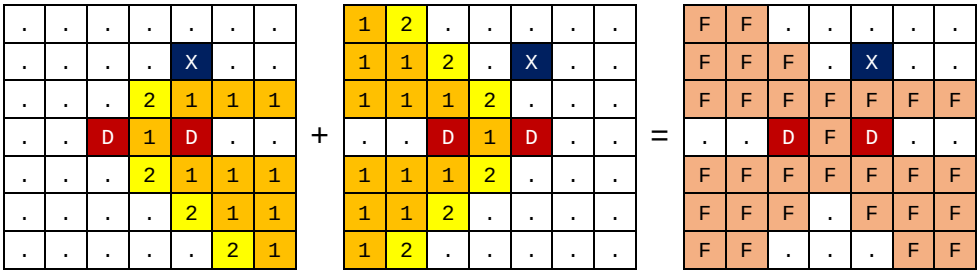
7. When a dragon spews out a stream of fire, it starts by spawning fire of the first kind in the square in front of it. To its relative left and right of that square, the dragon will also attempt to spawn fire squares of the second kind with the primary direction going from the dragon to that square and a secondary direction of the direction the dragon is facing. A dragon will never spawn fire of the second kind with primary and secondary directions other than these. For example, if a dragon is facing right on an empty map, it will spawn fire as such (where `D` represents a dragon, a `1` for a square of fire of the first type, and a `2` for a square of fire of the second type):



8. Fire from a dragon does not interfere with fire from another dragon. Here, two dragons are at the top-left and bottom-left corners, both facing right. In this diagram, all squares covered by *either type* of fire are denoted as `F` and the contribution from each dragon is shown, plus the resulting total fire coverage:



9. Fire of either type cannot spawn at a square that is occupied by another dragon or by an obstacle. This diagram has two dragons facing each other. Each dragon will be in the way of some of the fire of the other dragon. There is also one obstacle (labeled `X`) that will block fire from spreading further (again `F` represents a square of fire of *either* type):



Andrew can determine the moment in time when dragons are about to fire and must find safe squares that will not be touched by fire at all. However, following the notes he has about the map he is playing gives him a headache so he has come to you to write a program to determine which squares on the map are covered by fire.

The Problem:

Given a map, print out the same map updated with the fire spewed out by any and all dragons that appear on it.

The Input:

The input will begin with a positive integer, *m*, representing the number of maps. Each map will start with a line with two positive integers, *h* and *w* ($h \leq 20$; $w \leq 20$), representing the height and the width of the map, respectively. Following this, there will be *h* lines of *w* characters representing the map. Each character will be one of: `X`, `.`, `V`, `^`, `<` or `>`. The characters `<`, `>`, `^` and `V` represents a dragon facing left, right, up and down, respectively; the character `X` is an obstacle and the period (`.`) is an empty square.

The Output:

For each map, first print out "`Map #`*`i`*`:`" on a separate line where *i* is the number of the map in the input (starting at 1). After this, print *h* lines of *w* characters each, representing the map after the dragons spew out fire. If an originally empty square is now covered by any sort of fire from any dragon, print an `F` instead of the original period (`.`). Leave a blank line after the output for each map.

Sample Input:

```
2
4 4
.V..
....
....
...X
3 3
...
..^
...
```

Sample Output:

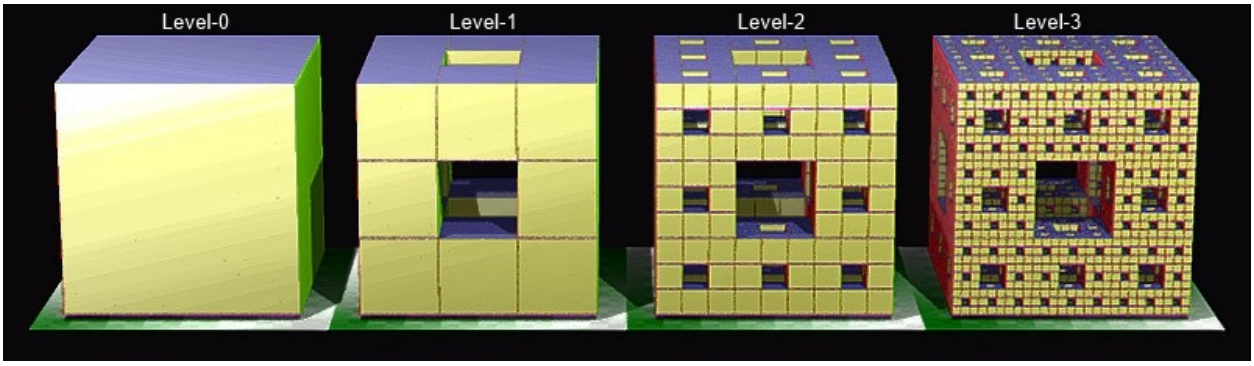
```
Map #1:
.V..
FFF.
FFFF
FFFX

Map #2:
.FF
..^
...
```

Menger Sponges

Filename: sponge

Tyler not only likes to fold pieces of paper, but he really likes fractals! He just recently acquired a 3D printer and wants to print out his very own Menger Sponge. The Menger Sponge is a 3D fractal that is constructed using the following steps:

1. Begin with a cube. This is called a level-0 Menger Sponge.
2. Divide every face of the cube into 9 squares. This will divide the cube into 27 smaller cubes.
3. Remove the smaller cube in the middle of each face and remove the smaller cube in the very center, leaving 20 smaller cubes. This is a level-1 Menger Sponge.
4. Repeat steps 2 and 3 for each of the remaining smaller cubes to create higher level sponges.



However, Tyler is not sure how much material he'll need to make his creation; so he's decided to request your help.

The Problem:

Given the initial side length and the level of a Menger Sponge to be created, determine its volume.

The Input:

The input will start with a line containing a single positive integer, *t*, representing the number of sponges to process. This line will be followed by *t* sponges, each on a separate line. Each sponge will contain 2 integers, *n* ($0 \leq n \leq 50$) and *m* ($1 \leq m \leq 200$), separated by a single space representing the level of the sponge Tyler wants to print and the side length (in millimeters) of the initial cube, respectively.

The Output:

For each sponge, output "`Sponge #`*`i`*`: `*`v`*" where *i* is the sponge number in the input (starting with 1) and *v* is the volume (in mm^3) of material used to create the corresponding sponge rounded to 3 decimal places (for example, 0.0014 rounds down to 0.001 and 0.0015 rounds up to 0.002).

Sample Input:

```
3
1 10
3 100
0 1
```

Sample Output:

```
Sponge #1: 740.741
Sponge #2: 406442.107
Sponge #3: 1.000
```

Ant Tunneling

Filename: `ant`

The evil ants of UCF have taken your teacher hostage in order to force you to help them with a problem. They are trying to interconnect their ant hills, so that there is some way to get from any ant hill to any other ant hill using only tunnels they have dug. They are quite smart ants, so they have already mapped out all the possible tunnels they can dig to connect their ant hills and assigned each a cost. However, before they get to digging, they demand your help to determine which set of tunnels they should choose to dig to minimize the total cost while still connecting all of their ant hills (directly or indirectly). A pair of ant hills is considered connected if there exists a series of tunnels connected end-to-end that form a path between each of the hills. The ants can start digging from any ant hill, but in the end all pairs of hills must have a path of fully dug tunnels connecting them.

The Problem:

Given the cost of all possible tunnels to connect pairs of ant hills, determine the minimum total cost to dig a tunnel network that connects all ant hills.

The Input:

The first line will contain a positive integer, *n*, the number of campuses for which the ants want you to solve this problem (UCF has many satellite campuses). Following will be descriptions of *n* campuses. Each campus description starts with a line containing two integers, *h* ($1 \leq h \leq 100$) and *t* ($0 \leq t \leq h * (h - 1) / 2$), representing the number of ant hills and the number of tunnels the ants can possibly dig, respectively. Each ant hill is assigned a unique number from 1 to *h*. The following *t* lines will describe the possible tunnels. Each tunnel is described by three integers, *x*, *y* ($1 \leq x < y \leq h$) and *d* ($1 \leq d \leq 1{,}000$), representing the two ant hills this tunnel connects and the cost of digging this tunnel to connect them, respectively. For each campus, each possible pair of ant hills will have at most one tunnel between them.

The Output:

For each campus, the output should start with "`Campus #`*c*`:`" where *c* is the number of the campus in the input (starting at 1). Follow this with either a single integer which is the minimum sum of difficulty costs to dig tunnels in a matter that connects all ant hills (directly or indirectly) or with "`I'm a programmer, not a miracle worker!`" if there is no way to dig tunnels to connect them all.

Sample Input:

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

Sample Output:

```
Campus #1: 5
Campus #2: I'm a programmer, not a miracle worker!
Campus #3: 6
```