Thirty-third Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
Blurry Sunshiny Day	blurry
Nate's Diverse Numbers	diverse
Charles and the Corgi Conundrum	corgi
Life Decisions	life
Huge Super Perfect Track	track
Hating on Fractions	fractions
The Invertomancer	invert
David and the Deli	deli
♪ Catchy Song ♪	song
Doubling Money	doubling
Perfectly Balanced Sentences	sentence
Unique Code Names	codenames

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

For example, if you are solving Huge Super Perfect Track:
Call your program file: track.c, track.cpp, track.java, or track.py
Call your Java class: track

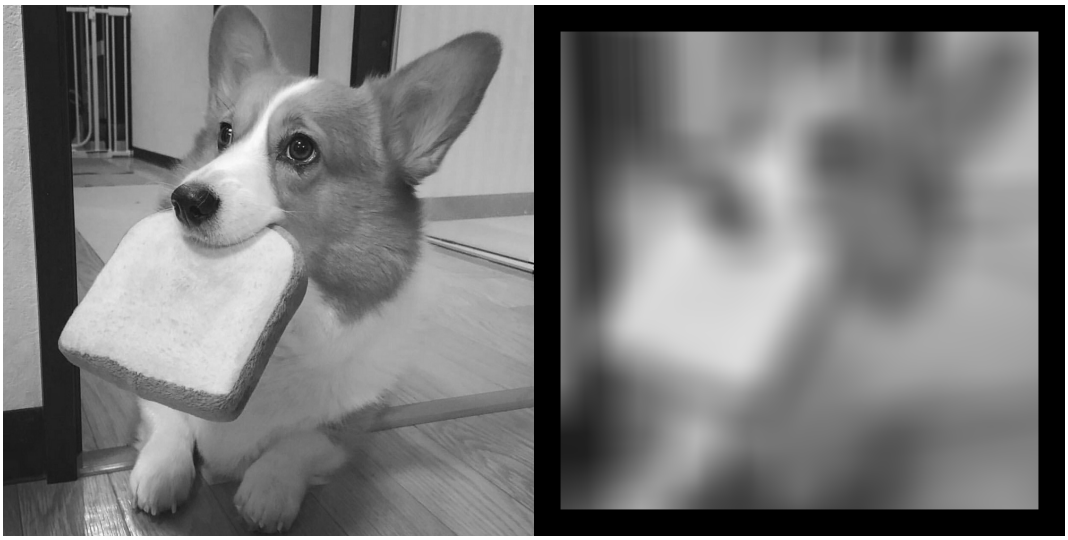
Blurry Sunshiny Day

Filename: `blurry`

Jimmy can see clearly now; the rain is gone! This is really quite unfortunate, though, because Jimmy prefers when it's blurry. To simulate poorly prescribed glasses which make everything blurry and block some of his peripheral vision, Jimmy has built an augmented reality (AR) set that will take in an n by n picture in which we will represent as a 2D matrix of brightness values, and output a more blurry (and slightly blocked) version of that picture. Jimmy also has a certain positive value, k (k is always odd), which describes how blurry the picture will be. The AR set should alter the picture in the following way:

- Any pixels with fewer than (k-1)/2 pixels separating them and any edge of the picture should be set to brightness 0.
- All other pixels, if there are any, should be set to the mean (average) brightness of the k by k square of pixels on the original image centered on that pixel.

For example, an example of a 1,000 by 1,000 pixel image, blurred with $k = 101$ is shown below. As you can see, the blurred version displays a much cuter Corgi enjoying a much yummier piece of toast!



Jimmy tried the naïve approach (using many loops), but it was too slow so he needs your expert help! Help Jimmy determine the final brightness of each pixel in his images (some of which are quite large) after he has blurred the image to his liking. All brightness levels should be rounded to the nearest integer. Round up if the first digit after the decimal point is a 5 or greater, or down otherwise. For example, brightness values of 2.5, 2.501, 2.9, and 3.495 should all be rounded to 3.

The Problem:

Given an n by n matrix of integers, as well as an integer, k, print a blurred version of the image with each pixel being the average of the k by k square centered on that pixel, or 0 if the square centered around that pixel leaves the image.

The Input:

The first line of input contains a single, positive integer, i, representing the number of images Jimmy would like you to process. The descriptions of i images follow. Each image description starts with a line containing two integers, n ($1 \leq n \leq 2{,}000$) and k ($1 \leq k < 2{,}000$; k is odd), representing the width and height of the image, and the constant describing the blurriness value, respectively. This is followed by n rows (lines), each containing of n integers, each representing the brightness of a pixel of the image that needs to be blurred. All n by n brightness values will be between 0 and 255 inclusive.

The Output:

For each image, first print "`Image #`x`:`" where x is the image number in the input (starting with 1). Then, starting on the next line, output n lines each containing n space-separated numbers representing the brightness of the corresponding pixel in the blurred image. Output all numbers rounded to the nearest integer (as explained earlier). Output a blank line after the output for each image.

Sample Input:

```
2
3 1
1 2 3
4 5 6
7 8 9
3 3
10 10 10
1 0 0
5 5 3
```

Sample Output:

```
Image #1:
1 2 3
4 5 6
7 8 9

Image #2:
0 0 0
0 5 0
0 0 0
```

Nate's Diverse Numbers

Filename: `diverse`

Nate works for the United States Census Bureau and is looking forward to the 2020 census. He's in charge of reporting total populations of individual cities and towns across the country. He's recently become dismayed to see that many cities have populations that do not properly represent their true diversity. For example, according to the last census, Detroit had a population of 713,777, a number which contains four 7's and just one 1 and one 3. Nate wanted to show how diverse the United States is, so he came up with a plan! For every city that he reports, he will select a number, k, and form a k-diverse number that is greater than or equal to the original population of the city. This number will be the new population that Nate reports.

A k-diverse number is a number in which no digit occurs more than k times. For example, the number 224,564 is 2-diverse, while 83,848 is not (but it is 3-diverse).

Nate knows that populations reported by the Census Bureau tend to be underestimates, since not everyone participates in the census. So he decided that his new reported population would be the closest k-diverse number greater than or equal to the original population. Sometimes it may not be possible to find an appropriate number that satisfies these conditions, however. In this case, Nate should choose a different value for k.

Since Nate isn't a programmer and he has a lot of cities to report, he's asked for your help solving this problem.

The Problem:

Help Nate find the smallest k-diverse number (if one exists) greater than or equal to each given number.

The Input:

The first line of the input contains a single, positive integer, c, representing the number of cities to consider. Each city starts on a new line and contains two integers, n and k ($1 \leq n \leq 10^{18}$; $1 \leq k \leq 10$), representing the original population reported for that city and the value Nate chose for k.

The Output:

For each city's original population, output on its own line the smallest k-diverse number greater than or equal to it. If there is no such number, output the line "`Find a different k`" instead.

(Sample Input and Sample Output follow on next page)

Sample Input:

```
5
713777 2
9000 1
123456789 1
9876543211 1
1 6
```

Sample Output:

```
713780
9012
123456789
Find a different k
1
```

Charles and the Corgi Conundrum

Filename: `corgi`

Charles, the Corgi-obsessed programmer, has just found Corgi Heaven! He visits a pond with exactly n dogs in a nearby field, all of which are Corgis. He wishes to take home exactly one Corgi, and as he is not partial to any particular doggo, he selects one at random. However, we all know that Corgis are amazingly energetic and friendly creatures who share inseparable bonds with others of their kind. Thus, upon choosing an initial Corgi, Charles also has to take its friends, and then its friends' friends, and so on…

Now given the friendships that all of the Corgis share, Charles wishes to know what the expected number of Corgis that he will take home is if he chooses an initial Corgi at random.

The Problem:

Given n Corgis and m friendship bonds between them, determine the number of Corgis that Charles can expect to take home, if he picks one Corgi at random.

The Input:

The first line contains a single, positive integer, s, which is the number of distinct scenarios to consider. For each scenario, the first line contains a two integers, n and m ($1 \leq n \leq 1{,}000$; $0 \leq m \leq 1{,}000$), representing the number of Corgis that exist in the pond and the number of friendship bonds, respectively. The following m lines of the scenario will contain friendship bonds, if any, and consist of two integers, u and v ($1 \leq u \leq n$; $1 \leq v \leq n$), meaning that Corgi number u and Corgi number v are friends.

The Output:

For each scenario, output "`Pond #`*`i`*`:` *`x`*" where i is the number of the pond (in the order of the input, starting with 1) and x is the expected number of Corgis that Charles can expect to take with him for this pond to 3 decimal places and rounded. For example, 12.1713 rounds to 12.171, 12.1715 rounds to 12.172, and 12.1718 rounds to 12.172.

(Sample Input and Sample Output follow on next page)

Sample Input:

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

Sample Output:

```
Pond #1: 2.000
Pond #2: 3.000
```

Life Decisions

Filename: `life`

Josh, an upcoming programming prodigy, wants to be a happy person in life … What more could a person ask for?

Of course, being human, he wishes to do as many things as possible in his lifetime, or at least until he graduates and has to start working. Thus, Josh wishes to make as many life decisions as possible and make his happiness strictly increase. But wait, can he do both?

Josh, being a programmer, naturally visualizes his decision pathway as a directed graph. A directed graph is a set of nodes that are connected via directed edges (one way roads). Josh starts at node number 1 and can make a decision corresponding to traveling from that node to some other node.

Also being an epic gamer, he represents his happiness in terms of a point system, in which he wishes to only increase the total amount of happiness in successive decisions he takes. If he is currently at a node A and makes a decision D to visit node B, he will assign this decision some total number of happiness points. In order to remain as happy as possible, he wishes to always make decisions that only *increase* the number of happiness points. For example, if he makes a decision D_1 and then a decision D_2, then the happiness points of D_2 should be strictly greater than the happiness points of D_1. However, note that it is possible for Josh to return to the same node, provided that his total happiness continues to increase.

We all know that in order to be successful in life, one can't stay good forever … Sometimes you have to make bad decisions that still give you happiness. Josh represents good decisions with positive values of happiness points, and bad decisions with negative values of happiness points (in this latter case, his happiness level would be the absolute value of the negative value).

And once you go bad, you can never go back. At any one point in time, including the first decision he makes, Josh can switch from making good decisions to making bad decisions (if possible). In such a case, he would also have to make sure that going from a good decision D_1 to a bad decision D_2, the absolute value of the happiness points of D_1 should be less than the absolute value of the happiness points of D_2 because, again, he always increases his happiness. He would also have to make sure that for any two consecutive bad decisions, D_1 and D_2, the absolute value of the happiness points of D_1 must be less than the absolute value of happiness points of D_2.

So, given the graph of his life decisions, what is the maximum number of decisions that Josh can make, given that his happiness points strictly increase? Josh always starts at node number 1.

The Problem:

Given a scenario of life decisions (represented as nodes and directed connections representing potential decisions), determine the maximum number of decisions that Josh can make while making his happiness strictly increase.

The Input:

The first line of the input contains a single, positive integer, *s*, representing the number of scenarios to consider. Each scenario is defined by multiple lines. For each scenario, the first line contains a two integers, *n* and *m* ($1 \le n \le 10^5$; $1 \le m \le 10^5$), representing the number of nodes and directed connections (decisions), respectively. The following *m* lines will consist of three integers, *u*, *v* and *w*, ($1 \le u \le n$; $1 \le v \le n$; $|w| \le 10^9$, $w \ne 0$), meaning that making this decision at node *u* in life will make Josh travel to node *v* in life and result in *w* total happiness (again, if this is a good decision, then $w > 0$; otherwise $w < 0$).

The Output:

For each scenario, output "`Scenario #`*i*`:` *x*" where *i* is the scenario number (in the order of the input, starting with 1) and *x* is a single integer that represents the maximum number of decisions Josh can make while making his happiness strictly increase.

Sample Input:

```
2
4 5
1 2 5
1 3 10
3 2 -13
2 4 -17
3 4 5
9 13
1 2 2
1 3 3
2 3 4
3 2 4
2 4 7
2 5 -4
3 7 5
4 6 3
5 6 17
7 9 7
7 8 -8
9 8 -11
6 9 3
```

Sample Output:

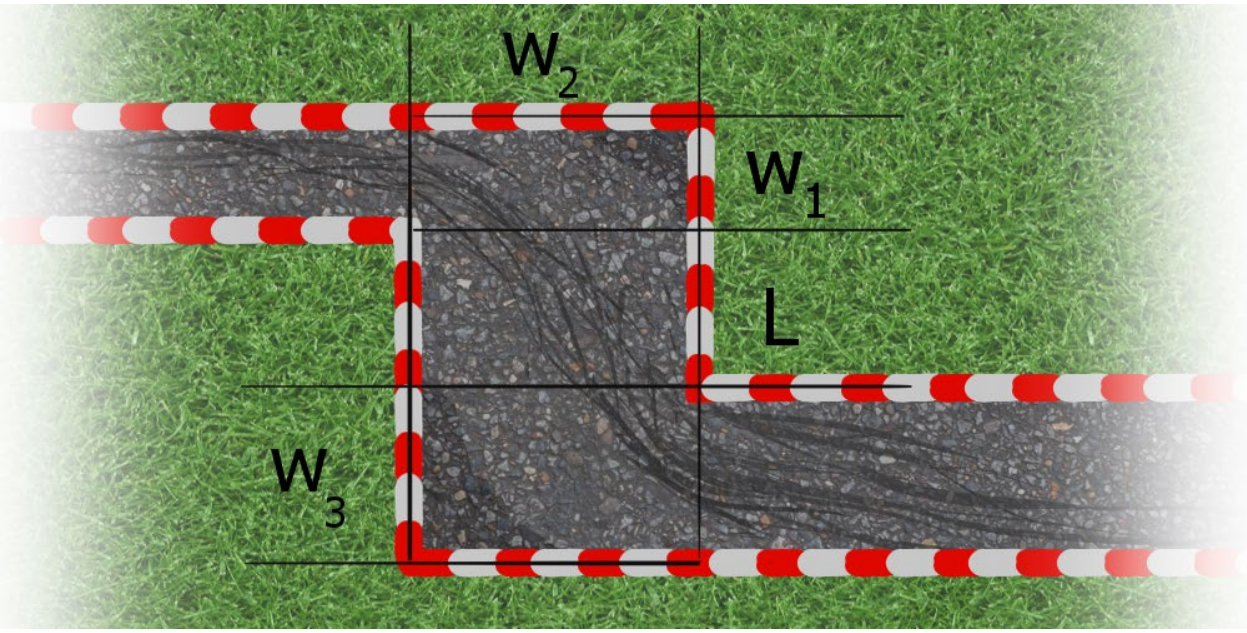
```
Scenario #1: 3
Scenario #2: 5
```

Huge Super Perfect Track

Filename: `track`

Every year, teams of high school students work together to build a go kart to race on the Huge Super Perfect Track (HSPT). This year, your team plans to build an extremely aerodynamic, needle-shaped car. Your car is almost infinitely thin (it will be autonomous, so you don't have to worry about fitting inside), but you want to make it as long as possible such that it can still finish. For simplicity, you can model your car as a line segment that can move and rotate however you would like as long as it stays on the track.

The Huge Super Perfect Track consists of three roads of widths, w_1, w_2, and w_3 meters, respectively, that meet at right angles as shown below. The length of road 2 measured from the bottom of road 1 to the top of road 3 is L, and roads 1 and 3 are infinitely long. Your car will need to be able to get from the start line on road 1 to the finish on road 3 without any part of it going off the track. Help your team find the maximum length of the car that can make it through the turns, assuming perfect driving/drifting. Your answer will be accepted if it is within either 10^{-6} meters or 0.0001% of the right answer.



Since there seems to be some confusion online as to what the real dimensions of the HSPT are, your program should handle several possible track dimensions, just to be safe.

The Problem:

Given the potential dimensions of the HSPT, determine the longest possible car your team can build so that it is still able to make it through each track without going off the road.

The Input:

The first line of the input contains a single, positive integer, *d*, representing the number of potential sets of track dimensions of the HSPT that you have found online. Each potential track configuration follows, each on a single line. The line contains four positive integers, w_1, w_2, w_3, and *L*, describing the dimensions of the track according to the track blueprint. Each dimension is at most length 10^6.

The Output:

For each potential set of track dimensions, output "`Track Blueprint #`*i*`: `*c*" on its own line where *i* is the number of set of track dimensions (in the order of the input, starting with 1) and *c* is the length of the longest car your team could possibly build that could still make it through the track. Be sure to print this length to at least 6 decimal points. Again, your answer will be accepted if it is within either 10^{-6} meters or 0.0001% of the right answer.

Sample Input:

```
2
10 8 30 80
10 8 30 1
```

Sample Output:

```
Track Blueprint #1: 25.403315272
Track Blueprint #2: 88.684835231
```

Hating on Fractions

Filename: `fractions`

Sally hates fractions with a burning passion! In fact, she hates them so much that she began to hate arrays as well, if they contain a pair of integers such that one of them is not evenly divisible by the other. Her parents are taking her to an expert to help her overcome her hating on fractions!

To help treat Sally's phobia, she will now be exposed to various different arrays. Write a program to predict Sally's reaction to each array.

The Problem:

For each array, determine if Sally hates it or not.

The Input:

The first line contains a single, positive integer, t, representing the number of arrays. For each array, there are two lines of input, the first of which contains a single integer, $1 \leq n \leq 1000$, representing the length of the array. The second line contains n integers, $1 \leq a_i \leq 10^9$, representing the elements of the array itself.

The Output:

For each array, output a single line containing "`Array #`*`i`*`:`" where *`i`* is the number of the array (in the order of the input, starting with 1), followed by a space and then Sally's response. Sally's response is very predictable: if she hates the array she will say "`Go away!`" and if she is okay with the array (does not hate it), she will say "`This array is bae!`".

Sample Input:

```
3
4
2 3 5 7
3
4 8 16
2
1 1000
```

Sample Output:

```
Array #1: Go away!
Array #2: This array is bae!
Array #3: This array is bae!
```

The Invertomancer

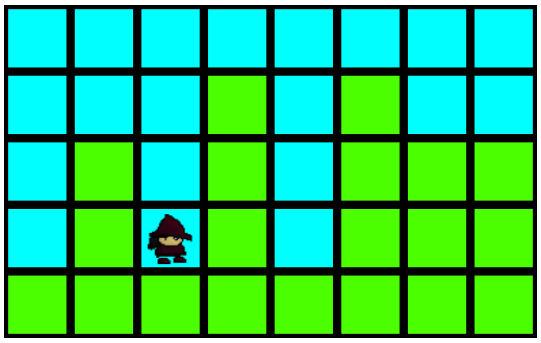
Filename: `invert`

The Invertomancer lives in a 2D screen (represented as a grid of r rows and c columns), and he wants to walk from the left-most column of the screen to the right-most column. The grid is composed out of earth blocks (`#`) and empty spaces (`.`). The invertomancer begins on top of the top-most earth block on the left-most column of the screen.

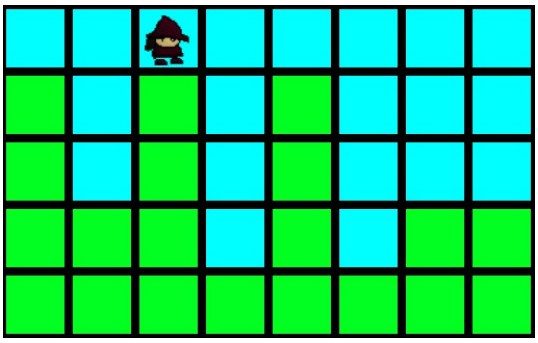
The invertomancer can climb up at most k blocks. Therefore, he can move to the column to the right of him only if the height of the highest earth block in that column is at most k blocks above his current elevation. When he does move to the right, he always moves on top of the highest block. If the highest block is below his current elevation, he moves down due to gravity.

Of course, the Invertomancer has a special ability: he can invert the world! When he does so, the world flips vertically, and all earth blocks become empty space, and vice versa (as seen below). When this happens, he is forced to move on top of the (newly flipped) block he was formerly occupying as empty space.

Normal



Inverted



Using this ability is very tiring, so the Invertomancer would like to limit its use as much as possible. He would like to know the minimum number of times he must invert the world in order to travel from the left-most side of the world to the right-most side of the world.

The Problem:

Determine the minimum number of times the invertomancer must invert the world to reach the right-most column of the screen.

The Input:

The first line of input contains a single, positive integer, *w*, representing the number of worlds. Each world's description will begin with a line containing two positive integers, *r* and *c* ($2 \leq r \leq 100$; $2 \leq c \leq 100$), representing the number of rows and columns of the grid, respectively, followed by a single, positive integer, *k* ($1 \leq k \leq 100$), representing the height that the invertomancer can climb. The following *r* lines will each contain *c* characters, either "`#`" if the cell is an earth block, or "`.`" if it is empty space.

Within each world, it is guaranteed that the top row is composed entirely out of empty space, and the bottom row is composed entirely out of earth blocks. Furthermore, each empty space cell that is not in the top row has empty space above it, and each earth block cell that is not in the bottom row has an earth block below it.

The Output:

For each world, output "`World #`*i*`: `*c*" where *i* is the number of the world (in the order given in the input, starting from 1), and *c* is the number of times the invertomancer must invert the world.

Sample Input:

```
2
5 8 2
........
...#.#..
.#.#.###
.#.#.###
########
4 4 1
....
#..#
#.##
####
```

Sample Output:

```
World #1: 3
World #2: 0
```

David and the Deli

Filename: `deli`

David really likes the combination of turkey and Swiss cheese, but recently he was urged to try new things. When David wants to try a new type of sandwich he goes to the local deli, Ellie's Deli, to get a sandwich.

Ellie has the widest selection of meats and cheeses in the entire continental United States. Her deli is set up such that there are n toppings labelled from *1* to n from left to right. Each topping has an associated price and David always has a budget. When David picks toppings he starts from the left. He picks every topping he can buy and skips only the toppings that cost more than his remaining budget. Following these rules, David buys at most one of each topping.

Ellie knows David's method and sometimes changes the prices of items so that David gets a different set of items on his sandwich. David goes to Ellie's Deli nearly every day, so he knows that Ellie only uses non-negative powers of two for prices.

The Problem:

Your task is to respond to two types of queries. The first type is "given David's budget and with the current menu prices, how many toppings will David get?" The other type of query is to update the price of the i^{th} topping (ordered from *1* to n from left to right).

The Input:

The first line of input contains a single, positive integer, s, representing the number of scenarios of David's visits to Ellie's Deli. Each scenario is given on multiple lines. The first line of each scenario has two positive integers, n and m ($n \leq 500{,}000$; $m \leq 500{,}000$), representing the number of toppings and queries, respectively. The next line has n integers, representing the initial prices of the toppings in pennies.

Each scenario ends with m lines. Each line describes a query. The first type of query is described as "`1 b`" where b is David's budget ($1 \leq b \leq 10^{18}$). The second type is "`2 i x`" where i is the index of the topping whose price should now become x. The prices will always be non-negative powers of two of at most 2^{30}.

The Output:

For each scenario, output on a new line "`Scenario #a:`" where `a` is the number of the day (in the order given in the input, starting from 1). Then, for each query of the first type, output on its own line "`David can get t toppings`" where t is a single integer that is the number of toppings David can get on his sandwich with the current set of prices. Output a blank line after each scenario.

Sample Input:

```
2
3 3
2 1 4
1 3
2 1 16
1 17
6 1
1 1 1 1 1 1
1 5
```

Sample Output:

```
Scenario #1:
David can get 2 toppings
David can get 2 toppings

Scenario #2:
David can get 5 toppings
```

♪ Catchy Song ♪

Filename: `song`

Emmet and Lucy are back! And they return with a new catchy song – literally, entitled "Catchy Song" – that will be stuck in your head for this entire competition! Especially the repetitive parts … time to sing along! Everybody!

This song's gonna get stuck inside yo'
This song's gonna get stuck inside yo'
This song's gonna get stuck inside yo' head

'Cause it's so catchy, catchy
It's such a catchy song
Gonna make you happy, happy
I'm tryna fight and sing along

The Problem:

Given how many times to repeat "`This song's gonna get stuck inside yo'`", sing along by outputting it that many times, followed by "`head`" on a final line.

The Input:

The first line will contain a single, positive integer, *n*, representing the number of stanzas to process. Following this, each of the next *n* lines will contain a single, positive integer, *r* ($r \leq 10$), representing how many times to repeat the lyric.

The Output:

For each stanza, output on a new line "`Stanza #`*i*`:`" for stanza *i* in the input (in the order given in the input, starting from 1). Then output the lyrics as given for the stanza. Follow the output for each stanza by a single blank line.

Sample Input:

```
2
2
3
```

Sample Output:

```
Stanza #1:
This song's gonna get stuck inside yo'
This song's gonna get stuck inside yo'
head

Stanza #2:
This song's gonna get stuck inside yo'
This song's gonna get stuck inside yo'
This song's gonna get stuck inside yo'
head
```

Doubling Money

Filename: `doubling`

As you may know, Ali loves circles! He has just begun playing a new Massive Multiplayer Online Role-playing Game (MMORPG), and he is very excited! You see, it uses a coin-based money system (and of course, coins are circular). After spending a long time grinding for money, Ali was fortunate enough to meet a very generous user, EpicJohnny42. EpicJohnny42 made an offer: if you give me all your money, I will give you double the amount.

What Ali doesn't know is that EpicJohnny42 is a devious scammer. When they trade, he will give Ali double the amount, except he will "forget" to append the last digit. For example, if Ali had 9,874 gold, he will receive 1,974 gold in return instead of 19,748. Unfortunately, while you were reading this problem Ali had already accepted the trade, failing to notice the error. Now your task is to calculate how much money Ali had lost!

The Problem:

Determine the amount of money Ali lost in the trade.

The Input:

The input begins with a single, positive integer, t, representing the number of trades. Then, t lines follow, each with a single integer, m ($5 \leq m \leq 100{,}000$), representing how much money Ali had before the trade.

The Output:

For each trade, output a line containing "`Trade #`*i*`:` *x*" where *i* is the trade number (in the order given in the input, starting from 1) and *x* is the amount that Ali lost in the trade.

Sample Input:

```
3
3502
10000
6
```

Sample Output:

```
Trade #1: 2802
Trade #2: 8000
Trade #3: 5
```

Perfectly Balanced Sentences

Filename: `sentence`

After watching his favorite non-anime movie, Atharva came to believe that all things should be perfectly balanced, especially capitalization! Atharva thinks a word is "balanced" if it has the same number of uppercase letters and lowercase letters. But what about sentences? Atharva considers a whole sentence to be balanced if the concatenation of all the words into one large word is balanced. However, he needs your help to verify whether sentences are balanced!

The Problem:

Given a sentence, check if it is perfectly balanced.

The Input:

The first line of input contains a single, positive integer, *s*, representing the number of sentences that follow. The first line of each sentence contains a single, positive integer, *n* ($n \leq 100$), representing the number of words in the sentence. On the next line are *n* words separated by single spaces. Each word is no more than 100 characters long and only consists of upper- and lowercase English letters.

The Output:

For each sentence, output a single line containing "`Sentence #`*`i`*`:`" where *i* is the number of the sentence (in the order given in the input, starting from 1), followed by a single space and then either "`Yes`" if the sentence is balanced, or "`No`" if it is not.

Sample Input:

```
2
3
CpLuSPluS iS gReAt
5
i am a sad string
```

Sample Output:

```
Sentence #1: Yes
Sentence #2: No
```

Unique Code Names

Filename: `codenames`

Your city is putting on an event to promote local businesses. There will be dozens of restaurants, music schools, and other small businesses there to give away samples and coupons and hand out brochures. The event organizers would like to assign each business a unique three-letter code in order to easily refer to them, and they've asked for your help to come up with the codes.

They have given you a few restrictions on creating the codes because they want to be able to recognize the business based on the code. Firstly, the code must start with the first letter of the business name. Secondly, the code must be a subsequence of the business name (ignoring case). This means that the code can be created by removing 0 or more letters from the business name without rearranging the letters. Finally, the code must be exactly 3 uppercase letters (no spaces) and each code must be unique.

The Problem:

Determine if it's possible to create a code for every business, such that no two businesses have the same code.

The Input:

The first line will contain an integer, v, denoting the number of events. Each event will start with an integer, n ($1 \leq n \leq 100$), representing the number of businesses at the event. The following n lines will each contain the name of a business. Each name will consist only of uppercase and lowercase letters and spaces. The name will not start or end with a space. It will contain at least 3 letters (of any case) and the total length of the name will not exceed 50. For each event, it is guaranteed that there will be no more than 8 businesses that start with the same letter.

The Output:

For each event, first output a line in the form "`Event #e:`" where e is the event number in the input (starting from 1). Following this, if it is possible to give every business a unique code, output n lines, one for each abbreviation (in all uppercase letters). If it is impossible to uniquely assign a code to each business, output a single line: "`Not Possible`". If there are multiple solutions, you may print any. Each event should be followed by a blank line.

(Sample Input and Sample Output follow on next page)

Sample Input:

```
2
5
Hair Cuttery
Chick fil a
Longhorn
Starbucks
Subway
4
Abc
Abd
Abcd
Acd
```

Sample Output:

```
Event #1:
HCU
CHA
LON
SBK
SAY

Event #2:
Not Possible
```