Qualification Round 2014 Problem C. **Minesweeper Master**

Problem

Minesweeper is a computer game that became popular in the 1980s, and is still included in some versions of the *Microsoft Windows* operating system. This problem has a similar idea, but it does not assume you have played *Minesweeper*.

In this problem, you are playing a game on a grid of identical cells. The content of each cell is initially hidden. There are **M** mines hidden in **M** different cells of the grid. No other cells contain mines. You may click on any cell to reveal it. If the revealed cell contains a mine, then the game is over, and you lose. Otherwise, the revealed cell will contain a digit between 0 and 8, inclusive, which corresponds to the number of neighboring cells that contain mines. Two cells are neighbors if they share a corner or an edge. Additionally, if the revealed cell contains a 0, then all of the neighbors of the revealed cell are automatically revealed as well, recursively. When all the cells that don't contain mines have been revealed, the game ends, and you win.

For example, an initial configuration of the board may look like this ('*' denotes a mine, and 'c' is the first clicked cell):

```
*..*...**.

....*.....

..c..*....

........*.

..........
```

There are no mines adjacent to the clicked cell, so when it is revealed, it becomes a 0, and its 8 adjacent cells are revealed as well. This process continues, resulting in the following board:

```
*..*...**.
```

```
1112*.....

00012*....

00001111*.

00000001..
```

At this point, there are still un-revealed cells that do not contain mines (denoted by '.' characters), so the player has to click again in order to continue the game.

You want to win the game as quickly as possible. There is nothing quicker than winning in one click. Given the size of the board (**R** x **C**) and the number of hidden mines **M**, is it possible (however unlikely) to win in one click? You may choose where you click. If it is possible, then print any valid mine configuration and the coordinates of your click, following the specifications in the *Output* section. Otherwise, print "Impossible".

Input

The first line of the input gives the number of test cases, **T**. **T** lines follow. Each line contains three space-separated integers: **R**, **C**, and **M**.

Output

For each test case, output a line containing "Case #x:", where x is the test case number (starting from 1). On the following **R** lines, output the board configuration with **C**characters per line, using '.' to represent an empty cell, '*' to represent a cell that contains a mine, and 'c' to represent the clicked cell.

If there is no possible configuration, then instead of the grid, output a line with"Impossible" instead. If there are multiple possible configurations, output any one of them.

Limits

0 ≤ **M** < **R** * **C**.

Small dataset

1 ≤ **T** ≤ 230.
1 ≤ **R**, **C** ≤ 5.

Large dataset

1 ≤ **T** ≤ 140.
1 ≤ **R**, **C** ≤ 50.

Sample

Input

```
5
5 5 23
3 1 1
2 2 1
4 7 3
10 10 82
```

Output

```
Case #1:
Impossible
Case #2:
c
.
*
Case #3:
Impossible
Case #4:
......*
.c....*
.......
..*....
Case #5:
**********
**********
**********
****....**
***.....**
***.c...**
***....***
**********
**********
**********
```

Round 1A 2014 Problem C. **Proper Shuffle**

Problem

A *permutation* of size **N** is a sequence of **N** numbers, each between 0 and **N**-1, where each number appears exactly once. They may appear in any order.

There are many (**N** *factorial*, to be precise, but it doesn't matter in this problem) permutations of size **N**. Sometimes we just want to pick one at random, and of course we want to pick one at random *uniformly*: each permutation of size **N** should have the same probability of being chosen.

Here's the pseudocode for one of the possible algorithms to achieve that goal (we'll call it the *good* algorithm below):

```
for k in 0 .. N-1:
  a[k] = k
for k in 0 .. N-1:
  p = randint(k .. N-1)
  swap(a[k], a[p])
```

In the above code, `randint(a .. b)` returns a uniform random integer between **a** and **b**, inclusive.

Here's the same algorithm in words. We start with the *identity* permutation: all numbers from 0 to **N**-1 written in increasing order. Then, for each **k** between 0 and **N**-1, inclusive, we pick an independent uniform random integer $\mathbf{p_k}$ between **k** and **N**-1, inclusive, and swap the element at position **k** (0-based) in our permutation with the element at position $\mathbf{p_k}$.

Here's an example for **N**=4. We start with the identity permutation:

0 1 2 3

Now **k**=0, and we pick a random $\mathbf{p_0}$ between 0 and 3, inclusive. Let's say we picked 2. We swap the 0th and 2nd elements, and our permutation becomes:

2 1 0 3

Now **k**=1, and we pick a random $\mathbf{p}_1$ between 1 and 3, inclusive. Let's say we picked 2 again. We swap the 1st and 2nd elements, and our permutation becomes:

2 0 1 3

Now **k**=2, and we pick a random $\mathbf{p}_2$ between 2 and 3, inclusive. Let's say we picked 3. We swap the 2nd and 3rd elements, and our permutation becomes:

2 0 3 1

Now **k**=3, and we pick a random $\mathbf{p}_3$ between 3 and 3, inclusive. The only choice is 3. We swap the 3rd and 3rd elements, which means that the permutation doesn't change:

2 0 3 1

The process ends now, and this is our random permutation.

There are many other algorithms that produce a random permutation uniformly. However, there are also many algorithms to generate a random permutation that look very similar to this algorithm, but are not uniform — some permutations are more likely to be produced by those algorithms than others.

Here's one bad algorithm of this type. Take the *good* algorithm above, but at each step, instead of picking $\mathbf{p}_\mathbf{k}$ randomly between **k** and **N**-1, inclusive, let's pick it randomly between 0 and **N**-1, inclusive. This is such a small change, but now some permutations are more likely to appear than others!

Here's the pseudocode for this algorithm (we'll call it the *bad* algorithm below):

```
for k in 0 .. N-1:

  a[k] = k

for k in 0 .. N-1:

  p = randint(0 .. N-1)
```

```
swap(a[k], a[p])
```

In each test case, you will be given a permutation that was generated in the following way: first, we choose either the good or the bad algorithm described above, each with probability 50%. Then, we generate a permutation using the chosen algorithm. Can you guess which algorithm was chosen just by looking at the permutation?

Solving this problem

This problem is a bit unusual for Code Jam. You will be given **T** = 120 permutations of**N** = 1000 numbers each, and should print an answer for each permutation – this part is as usual. However, you don't need to get all of the answers correct! Your solution will be considered correct if your answers for at least **G** = 109 cases are correct. However, you must follow the output format, even for cases in which your answer doesn't turn out to be correct.
The *only* thing that can be wrong on any case, yet still allow you to be judged correct, is swapping GOOD for BAD or vice versa; but you should still print either GOOD or BAD for each case.

It is guaranteed that the permutations given to you were generated according to the method above, and that they were generated independently of each other.

This problem involves randomness, and thus it might happen that even the best possible solution doesn't make 109 correct guesses for a certain input, as both the good and the bad algorithms can generate any permutation. Because of that, this problem doesn't have a Large input, and has just the Small input which you can try again if you think you got unlucky. Note that there is the usual 4-minute penalty for incorrect submissions if you later solve that input, even if the only reason you got it wrong was chance.

In our experience with this problem, that *did happen* (getting wrong answer just because of chance); so if you are confident that your solution should be working, but it failed, it might be a reasonable strategy to try again with the same solution which failed.

Good luck!

Input

The first line of the input gives the number of test cases, **T** (which will always be 120). Each test case contains two lines: the first line contains the single integer **N** (which will always be 1000), and the next line contains **N** space-separated integers - the permutation that was generated using one of the two algorithms.

Output

For each test case, output one line containing "Case #**x**: **y**", where **x** is the test case number (starting from 1) and **y** is either "GOOD" or "BAD" (without the quotes). You should output "GOOD" if you guess that the permutation was generated by the first algorithm described in the problem statement, and "BAD" if you guess that the permutation was generated by the second algorithm described in the problem statement.

Limits

T = 120
G = 109
N = 1000
Each number in the permutation will be between 0 and **N**-1 (inclusive), and each number from 0 to **N**-1 will appear exactly once in the permutation.

Sample

Input

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

Output

```
Case #1: BAD
Case #2: GOOD
```

Note

The sample input doesn't follow the limitations from the problem statement - the real input will be much bigger.

Round 1B 2014 Problem C. **The Bored Traveling Salesman**

Problem

Your boss is sending you out on an international sales trip. What joy!

You have **N** cities (numbered from 1 to **N**) to visit and can get to them using a set of bidirectional flights that go between the cities.

All of the cities must be visited at least once. To do this you can book any number of tickets, subject to the following conditions:

- Each ticket consists of 2 flights, one from a specific city **X** to another specific city**Y** (this is called the **outbound** flight), and the other one from city **Y** to city **X** (this is called the **return** flight).
- You must use the outbound flight before the corresponding return flight (you can use other flights in between).
- At most 1 outbound flight going to each city, although there is no limit on the return flights (multiple return flights can go to the same city).
- You must use all flights which belong to the tickets you booked.
- You can otherwise visit the cities in any order you like.
- You can start your trip from any city you choose. You may not take an outbound flight to your starting city.

Now you could try to minimize the total distance travelled, but you did that last time, so that would be boring. Instead you noticed that each city has a distinct 5 digit ZIP (postal) code. When you visit a city for the first time (this includes the city which you start from) you write down the zip code and concatenate these into one large number (concatenate them in the order which you visited each city for the first time). What is the smallest number you can achieve?

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow.

Each test case starts with a single line containing two integers: the number of cities **N**and the number of possible bidirectional flights **M**.

N lines then follow, with the i-th line containing the 5-digit zip code of the i-th city. No ZIP code will have leading zeros and all ZIP codes in each test case will be distinct.

M lines then follow, each containing two integers **i** and **j** (1 ≤ **i** < **j** ≤ **N**) indicating that a bidirectional flight exists between the **i**-th city and the **j**-th city. All flights will be distinct within each test case.

It is guaranteed that you can visit every city following the rules above.

Output

For each test case, output one line containing "Case #x: y", where x is the test case number (starting from 1) and y is the smallest number you can achieve by concatenating the ZIP codes along your trip.

Limits

1 ≤ **T** ≤ 100.
0 ≤ **M** ≤ **N** * (**N** - 1) / 2.

Small dataset

1 ≤ **N** ≤ 8.

Large dataset

1 ≤ **N** ≤ 50.

Sample

Input

```
4
3 2
10001
20000
10000
1 2
2 3
5 4
36642
28444
50012
29651
10953
1 4
2 3
```

Output

```
Case #1: 100002000010001
Case #2: 1095328444500122965136642
Case #3: 1095328444366422965150012
Case #4: 100011000210003100041000510006
```

```
2 5
4 5
5 5
36642
28444
50012
29651
10953
1 2
1 4
2 3
2 5
4 5
6 6
10001
10002
10003
10004
10005
10006
1 2
1 6
2 3
2 4
3 5
4 5
```

Round 1C 2014 Problem C. **Enclosure**

Problem

Your task in this problem is to find out the minimum number of stones needed to place on an N-by-M rectangular grid (N horizontal line segments and M vertical line segments) to enclose at least K intersection points. An intersection point is enclosed if either of the following conditions is true:

1. A stone is placed at the point.
2. Starting from the point, we cannot trace a path along grid lines to reach an empty point on the grid border through empty intersection points only.

For example, to enclose 8 points on a 4x5 grid, we need at least 6 stones. One of many valid stone layouts is shown below. Enclosed points are marked with an "x".

Input

The first line of the input gives the number of test cases, **T**. **T** lines follow. Each test case is a line of three integers: **N M K**.

Output

For each test case, output one line containing "Case #x: y", where x is the test case number (starting from 1) and y is the minimum number of stones needed.

Limits

1 ≤ **T** ≤ 100.
1 ≤ **N**.
1 ≤ **M**.
1 ≤ **K** ≤ **N** × **M**.

Small dataset

N × **M** ≤ 20.

Large dataset

N × **M** ≤ 1000.

Sample

Input

```
2
4 5 8
3 5 11
```

Output

```
Case #1: 6
Case #2: 8
```

Round 2 2014 Problem C. **Don't Break The Nile**

Problem

Aliens have landed. These aliens find our Earth's rivers intriguing because their home planet has no flowing water at all, and now they want to construct their alien buildings in some of Earth's rivers. You have been tasked with making sure their buildings do not obstruct the flow of these rivers too much, which would cause serious problems. In particular, you need to determine what the maximum flow that the river can sustain is, given the placement of buildings.

The aliens prefer to construct their buildings on stretches of river that are straight and have uniform width. Thus you decide to model the river as a rectangular grid, where each cell has integer coordinates (**X**, **Y**; 0 ≤ **X** < **W** and 0 ≤ **Y** < **H**). Each cell can sustain a flow of 1 unit through it, and the water can flow between edge-adjacent cells. All the cells on the south side of the river (that is with y-coordinate equal to 0) have an implicit incoming flow of 1. All buildings are rectangular and are grid-aligned. The cells that lie under a building cannot sustain any flow. Given these constraints, determine the maximum amount of flow that can reach the cells on the north side of the river (that is with y-coordinate equal to **H**-1).

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each test case will begin with a single line containing three integers, **W**, the width of the river,**H**, the height of the river, and **B**, the number of buildings being placed in the river. The next **B** lines will each contain four integers, **X0**, **Y0**, **X1**, and **Y1**. **X0**, **Y0** are the coordinates of the lower-left corner of the building, and **X1**, **Y1** are the coordinates of the upper-right corner of the building. Buildings will not overlap, although two buildings can share an edge.

Output

For each test case, output one line containing "Case #x: m", where x is the test case number (starting from 1) and m is the maximum flow that can pass through the river.

Limits

1 ≤ **T** ≤ 100.
0 ≤ **X0** ≤ **X1** < **W**.
0 ≤ **Y0** ≤ **Y1** < **H**.

Small dataset

$3 \leq \mathbf{W} \leq 100$.
$3 \leq \mathbf{H} \leq 500$.
$0 \leq \mathbf{B} \leq 10$.

Large dataset

$3 \leq \mathbf{W} \leq 1000$.
$3 \leq \mathbf{H} \leq 10^8$.
$0 \leq \mathbf{B} \leq 1000$.

Sample

Input

```
2
3 3 2
2 0 2 0
0 2 0 2
5 6 4
1 0 1 0
3 1 3 3
0 2 1 3
1 5 2 5
```

Output

```
Case #1: 1
Case #2: 2
```

Round 2 2014 Problem D. **Trie Sharding**

Problem

A set of strings **S** can be stored efficiently in a *trie*. A trie is a rooted tree that has one node for every prefix of every string in **S**, without duplicates.

For example, if **S** were "AAA", "AAB", "AB", "B", the corresponding trie would contain 7 nodes corresponding to the prefixes "", "A", "AA", AAA", "AAB", "AB", and "B".

I have a server that contains **S** in one big trie. Unfortunately, **S** has become very large, and I am having trouble fitting everything in memory on one server. To solve this problem, I want to switch to storing **S** across **N** separate servers. Specifically, **S** will be divided up into disjoint, non-empty subsets $\mathbf{T_1}$, $\mathbf{T_2}$, ..., $\mathbf{T_N}$, and on each server i, I will build a trie containing just the strings in $\mathbf{T_i}$. The downside of this approach is the total number of nodes across all **N** tries may go up. To make things worse, I can't control how the set of strings is divided up!

For example, suppose "AAA", "AAB", "AB", "B" are split into two servers, one containing "AAA" and "B", and the other containing "AAB", "AB". Then the trie on the first server would need 5 nodes ("", "A", "AA", "AAA", "B"), and the trie on the second server would also need 5 nodes ("", "A", "AA", "AAB", "AB"). In this case, I will need 10 nodes altogether across the two servers, as opposed to the 7 nodes I would need if I could put everything on just one server.

Given an assignment of strings to **N** servers, I want to compute the worst-case total number of nodes across all servers, and how likely it is to happen. I can then decide if my plan is good or too risky.

Given **S** and **N**, what is the largest number of nodes that I might end up with? Additionally, how many ways are there of choosing $\mathbf{T_1}$, $\mathbf{T_2}$, ..., $\mathbf{T_N}$ for which the number of nodes is maximum? Note that the **N** servers are different -- if a string appears in $\mathbf{T_i}$ in one arrangement and in $\mathbf{T_j}$ (**i** != **j**) in another arrangement, then the two arrangements are considered different. Print the remainder of the number of possible arrangements after division by 1,000,000,007.

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each test case starts with a line containing two space-separated integers: **M** and **N**. **M** lines follow, each containing one string in **S**.

Output

For each test case, output one line containing "Case #**i**: **X** **Y**", where **i** is the case number (starting from 1), **X** is the worst-case number of nodes in all the tries combined, and **Y** is the number of ways (modulo 1,000,000,007) to assign strings to servers such that the number of nodes in all **N** servers are **X**.

Limits

1 ≤ **T** ≤ 100.
Strings in **S** will contain only upper case English letters.
The strings in **S** will all be distinct.
N ≤ **M**

Small dataset

1 ≤ **M** ≤ 8
1 ≤ **N** ≤ 4
Each string in **S** will have between 1 and 10 characters, inclusive.

Large dataset

1 ≤ **M** ≤ 1000
1 ≤ **N** ≤ 100
Each string in **S** will have between 1 and 100 characters, inclusive.

Sample

Input

```
2
4 2
AAA
AAB
AB
B
5 2
A
B
C
D
E
```

Output

```
Case #1: 10 8
Case #2: 7 30
```

Round 3 2014 Problem D. **Willow**

Problem

Hanaa and Sherine are playing Willow, a game that is played on a board containing **N**cities. The i^{th} city contains $\mathbf{C_i}$ coins, and there are **N - 1** bidirectional roads running between the cities. All cities are reachable from one another. The game is played as follows:

First Hanaa chooses one of the cities as her starting location, then Sherine chooses one of the cities (possibly the same one Hanaa chose) as her starting location. Afterwards, they take turns playing the game, with Hanaa going first.

On a player's turn, that player *must* take all the coins on the city where she currently is, if there are any; there might be none if the city starts with no coins, or if one of the players has already started a turn in that city. Then, if possible, the player must travel to an adjacent city on a road. It might not be possible, because each road can be used at most once. This means that after one player has used a road, neither player is allowed to use the same road later. The game ends when neither Hanaa nor Sherine can make a move.

After the game ends, each player's score is equal to the difference between the number of coins she has and the number of coins her opponent has. If her opponent has more coins, this means that her score will be negative. Both players are trying to maximize their scores. Assuming that they are both using the best possible strategy to maximize their scores, what is the highest score that Hanaa can obtain?

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each test case starts with a line containing one integer **N**, the number of cities on the board. **N**lines then follow, with the i^{th} line containing an integer $\mathbf{C_i}$, the number of coins in city i.

Finally there will be another **N - 1** lines, with the i^{th} line (i starts from 1) containing a single integer j ($1 \le i < j \le \mathbf{N}$) indicating that there is a road between city i and city j. All cities are guaranteed to be reachable from one another at the start of the game.

Output

For each test case, output one line containing "Case #x: y", where x is the case number (starting from 1) and y is the highest score that Hanaa can obtain.

Limits

1 ≤ **T** ≤ 50.
0 ≤ $\mathbf{C_i}$ ≤ 10000.

Small dataset

2 ≤ **N** ≤ 80.

Large dataset

For 10 test cases, 2 ≤ **N** ≤ 4000.
For the remaining test cases, 2 ≤ **N** ≤ 500.

Sample

Input

```
3
3
1000
200
1000
2
3
8
8
0
8
0
0
0
0
10
2
5
4
5
6
7
```

Output

```
Case #1: 200
Case #2: -2
Case #3: 5100
```

```
8
10
150
200
0
5000
0
100
0
0
0
10000
10
3
8
5
8
7
8
9
10
```

World Finals 2014 Problem E. **Allergy Testing**

Problem

Kelly is allergic to exactly one of **N** foods, but she isn't sure which one. So she decides to undergo some experiments to find out.

In each experiment, Kelly picks several foods and eats them all. She waits **A** days to see if she gets any allergic reactions. If she doesn't, she knows she isn't allergic to any of the foods she ate. If she does get a reaction, she has to wait for it to go away: this takes a total of **B** days (measured from the moment when she ate the foods).

To simplify her experimentation, Kelly decides to wait until each experiment is finished (after **A** or **B** days) before starting the next one. At the start of each experiment, she can choose the set of foods she wants to eat based on the results of previous experiments.

Kelly chooses what foods to eat for each experiment to minimize the worst-case number of days before she knows which of the **N** foods she is allergic to. How long does it take her in the worst case?

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each test case on a single line, containing three space-separated integers: **N**, **A** and **B**.

Output

For each test case, output one line containing "Case #x: y", where x is the test case number (starting from 1) and **y** is the number of days it will take for Kelly to find out which food she is allergic to, in the worst case.

Limits

1 ≤ **T** ≤ 200.

Small dataset

$1 \leq \mathbf{N} \leq 10^{15}$.
$1 \leq \mathbf{A} \leq \mathbf{B} \leq 100$.

Large dataset

1 ≤ **N** ≤ 10^{15}.
1 ≤ **A** ≤ **B** ≤ 10^{12}.

Sample

Input

```
3
4 5 7
8 1 1
1 23 32
```

Output

```
Case #1: 12
Case #2: 3
Case #3: 0
```

In the first sample case:

- First, Kelly eats foods #1 and #2.
- If she gets no reaction after 5 days, she eats food #3. 5 days after *that*, she will know whether she is allergic to food #3 or food #4.
- If she does get a reaction to the first experiment, then 7 days after the first experiment, she eats food #1. 5 days after that, she will know whether she is allergic to food #1 or food #2.

World Finals 2014 Problem F. ARAM

Problem

In the game League of Legends™, you can play a type of game called "ARAM", which is short for "All Random, All Mid". This problem uses a similar idea, but doesn't require you to have played League of Legends to understand it.

Every time you start playing an ARAM game, you're assigned one of **N** "champions", uniformly at random. You're more likely to win with some champions than with others, so if you get unlucky then you might wish you'd been given a different champion. Luckily for you, the game includes a "Reroll" function.

Rerolling randomly reassigns you a champion in a way that will be described below; but you can't reroll whenever you want to. The ability to reroll works like a kind of money. Before you play your first ARAM game, you begin with **R** RD ("reroll dollars"). You can only reroll if you have at least 1 RD, and you must spend 1 RD to reroll. After every game, you gain 1/**G** RD (where **G** is an integer), but you can never have more than **R** RD: if you have **R** RD and then play a game, you'll still have **R** RD after that game.

If you have at least 1RD, and you choose to reroll, you will spend 1RD and be re-assigned one of the **N** champions, uniformly at random. There's some chance you might get the same champion you had at first. If you don't like the champion you rerolled, and you still have at least 1RD left, you can reroll again. As long as you have at least 1RD left, you can keep rerolling.

For example, if **R**=2 and **G**=2, and you use a reroll in your first game, then after your first game you will have 1.5 RD. If you play another game, this time without using a reroll, you will have 2.0 RD. If you play another game without using a reroll, you will still have 2.0 RD (because you can never have more than **R**=2). If you use two rerolls in your next game, then after that game you will have 0.5 RD.

You will be given the list of champions, and how likely you are to win a game if you play each of them. If you play 10^{100} games and choose your strategy optimally, what fraction of the games do you expect to win?

Input

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each starts with a line containing three space-separated integers: **N**, **R** and **G**. The next line contains **N** space-separated, real-valued numbers $\mathbf{P}_i$, indicating the probability that you will win if you play champion `i`.

Output

For each test case, output one line containing "Case #x: y", where x is the test case number (starting from 1) and y is the proportion of games you will win if you play 10^{100}games.

y will be considered correct if it is within an absolute or relative error of 10^{-10} of the correct answer. See the FAQ for an explanation of what that means, and what formats of real numbers we accept.

Limits

$1 \le \mathbf{T} \le 100$.
$0.0 \le \mathbf{P}_i \le 1.0$.
$\mathbf{P}_i$ will be expressed as a single digit, followed by a decimal point, followed by 4 digits.

Small dataset

$1 \le \mathbf{N} \le 1000$.
$1 \le \mathbf{R} \le 2$.
$1 \le \mathbf{G} \le 3$.

Large dataset

$1 \le \mathbf{N} \le 1000$.
$1 \le \mathbf{R} \le 20$.
$1 \le \mathbf{G} \le 20$.

Sample

Input

```
3
2 1 1
1.0000 0.0000
3 1 1
1.0000 0.0000 0.5000
6 2 3
0.9000 0.6000 0.5000 0.1000 0.2000 0.8000
```

Output

```
Case #1: 0.750000000000
Case #2: 0.666666666667
Case #3: 0.618728522337
```