Twenty-ninth Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
Hoo's Counting?	hoo
Infinite Fractal Fractions	fractal
Fishy Business 2: Fish Nuggets	fishytoo
Maximum Flowers	flowers
Circular Sums	sums
Nurikabe Checker	nurikabe
Troll Crossing	troll
Can I Have a Dollar?	dollar
The Little Bird	bird
Roughly the Size of a Barge	eggs
Swap Lobster	lobster

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

For example, if you are solving Hoo's Counting?:
Call your program file: hoo.c, hoo.cpp or hoo.java
Call your Java class: hoo

Hoo's Counting?

Filename: hoo

Mr. Owl has seen some extreme disciplinary action from the Tootsie Roll Company's board of lolly-licking logistics for greatly underselling their product. It turns out that the average consumer is not interested in buying a lollipop that only lasts three licks! If Mr. Owl can't move the latest batch of Tootsie Pops, he'll be one dead duck! I mean owl! Can you help Mr. Owl determine the maximum number of licks it'll take to get to the Tootsie Roll center of a Tootsie Pop? Mr. Owl is very aware he is asking you for help when owls have a reputation for being both wise and intelligent, but reminds you that this is a common misconception and that they are in fact some of the dumbest birds on the planet; almost certainly they should never be placed as head of an entire product division.

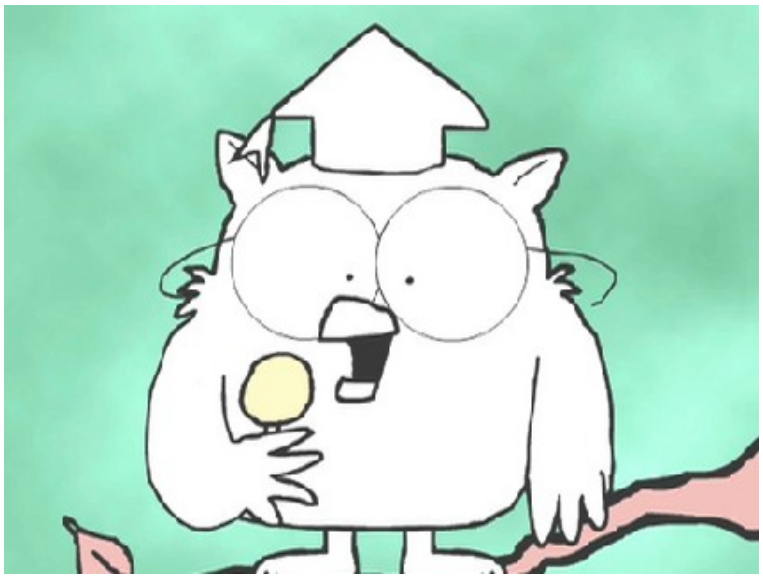


Figure 1: A dumb bird

The Problem:

As everyone knows, a lollipop of radius r must be licked in such a way that its radius is reduced to an integer divisor of r, excluding r. For example, a lollipop of radius 6 may be reduced to the sizes 3, 2, or 1 after a single lick, but not 6 as this would mean the lollipop was not licked at all. The Tootsie Roll center of a Tootsie Pop begins at radius 1, so once a Tootsie Pop has been reduced to size 1, its center has been reached. There are 3 sequences of licks that will reach the center of a size 6 Tootsie Pop:

$$6 \rightarrow 3 \rightarrow 1$$
$$6 \rightarrow 2 \rightarrow 1$$
$$6 \rightarrow 1$$

Only two of these are maximal lickings with length 2. Mr. Owl only needs you to find the length of any maximal licking for a given radius r, which will let him know the maximum amount of enjoyment a customer could derive from that particular pop. Remember, if a Tootsie Pop takes 3 or less licks to reach the center, the customer won't buy it and Mr. Owl needs to fly the coop!

The Input:

The first line of the input will contain a single, positive integer, *p*, representing the number of Tootsie Pops to process. Each of the next *p* lines will contain a single, positive integer, *r* ($1 \leq r \leq 1{,}000$), representing the radius of the given Tootsie Pop.

The Output:

For each Tootsie Pop, output "`Pop #`*`i`*`:` " where *i* is the lollipop's number (starting from 1), followed by either "*`x`*` licks? Your goose is cooked!`" if it takes 3 or less licks to reach the center of the Tootsie Pop, or "`A-one... A-two-HOO... A-`*`x`*`.`" where *x* is the maximum number of licks it takes to reach the center, otherwise.

Sample Input:

```
2
6
32
```

Sample Output:

```
Pop #1: 2 licks? Your goose is cooked!
Pop #2: A-one... A-two-HOO... A-5.
```

Infinite Fractal Fractions

Filename: `fractal`

Benoit decided to take a break from fractals for a few days and turn his attention to fractions. However, being Benoit, his fraction soon turned into what he described as a *fractal fraction*. He first chose two integers *a* and *b*, then arranged them into a fraction like this:

$$x = \cfrac{a}{\cfrac{a}{\cfrac{a}{\frac{a}{a}+\frac{b}{a}}+\cfrac{b}{\frac{a}{a}+\frac{b}{a}}}+\cfrac{b}{\cfrac{a}{\frac{a}{a}+\frac{b}{a}}+\cfrac{b}{\frac{a}{a}+\frac{b}{a}}}}+\cfrac{b}{\cfrac{a}{\cfrac{a}{\frac{a}{a}+\frac{b}{a}}+\cfrac{b}{\frac{a}{a}+\frac{b}{a}}}+\cfrac{b}{\cfrac{a}{\frac{a}{a}+\frac{b}{a}}+\cfrac{b}{\frac{a}{a}+\frac{b}{a}}}}$$

The fraction, like his fractals, continues infinitely.

The Problem:

Given two integers, *a* and *b*, help Benoit determine the absolute value of *x*.

The Input:

The input begins with a line containing a single, positive integer, *t*, representing the number of fractal fractions. This line is followed by *t* fraction descriptions. Each fraction description is on a new line and contains two integers, *a* and *b* (-1,000 < *a* < 1,000; -1,000 < *b* < 1,000).

The Output:

The output for each fraction should start with "`Fraction #i: `" where *i* is the fraction number (starting with 1), followed by the absolute value of *x* rounded to 2 decimal places (for example, 0.034 rounds down to 0.03 and 0.035 rounds up to 0.04). If *x* isn't a real number output "`DNE`" instead.

Sample Input:

```
3
17 8
3 -4
3 4
```

Sample Output:

```
Fraction #1: 5.00
Fraction #2: DNE
Fraction #3: 2.65
```

Fishy Business 2: Fish Nuggets

Filename: `fishytoo`

Your good friend Nikolai is back and has expanded his fish business, and he needs your help again to calculate his profits![1] Nikolai's best seller these days are his Fish Nuggets. How are they made? We probably don't want to know…but we can calculate how many Fish Nuggets can be made by the items caught in Nikolai's fishing net! One square inch of useful ingredients can be used to create one fish nugget. Nikolai's net is *h* by *w* inches and each square inch is represented by an ASCII character. An example of a catch in Nikolai's net is:

```
~~0
<><
0~#
```

The only items that can appear in Nikolai's net are seaweed (~)[2], fish (<>< or ><>)[3], rocks (0), and empty net (#)[4]. Remember that Nikolai is a master fisherman and that he only catches whole fish and those fish will not overlap.

We can calculate the amount of Fish Nuggets that can be made by how many useful characters there are in this ASCII representation. Nikolai can use both fish and seaweed to create his Fish Nuggets - but not rocks, Nikolai has other uses for those.[5] Here we can see that 6 of these characters are useful because the rocks and empty net cannot be made into Fish Nuggets, but the seaweed and fish parts can. Thus, 6 Fish Nuggets can be made from this catch.

The Problem:

Given an ASCII representation of Nikolai's net, calculate how many Fish Nuggets can be made using the net's contents as ingredients.

The Input:

The first line of the input will begin with a single, positive integer, *n*, representing the number of fishing nets. Each fishing net will begin with a line containing 2 positive integers, *h* and *w*, the height and width of the net, both no greater than 500. The next *h* lines will contain *w* characters each and correspond to the contents of Nikolai's net (limited to <, >, ~, 0, #).

---

[1] Where do fish keep their money?... In a riverbank!

[2] What does seaweed say when it's stuck at the bottom of the sea?... 'Kelp! Kelp!'

[3] Why is the ocean so blue?... Because fish say "Blu Blu Blue"

[4] Why don't fish play basketball?... Because they're afraid of the net!

[5] Nikolai's friends are "professional" rock-throwers

The Output:

For each net, output "`Net #`*`i`*`: `*`x`*` Fish Nuggets`" where *i* corresponds to the index of the fishing net (starting at 1), and *x* corresponds to how many Fish Nuggets can be created by the contents of that net.

Sample Input:

```
3
3 3
~~0
<><
0~#
4 5
~0~0~
~~~~~
~000~
~0~0~
3 3
<><
><>
###
```

Sample Output:

```
Net #1: 6 Fish Nuggets
Net #2: 13 Fish Nuggets
Net #3: 6 Fish Nuggets
```

Maximum Flowers

Filename: `flowers`

Briar Rose, known to a select few as Princess Aurora, loves collecting flowers, and her sixteenth birthday is coming up. The creatures of the forest are throwing her a surprise party and would like to get her as many flowers as they can. They have asked some brilliant computer programmers to help them, and the programmers recommended that they set up an assembly line. The assembly line operates as follows:

There are n animals in the line. The first animal is called the "source", and is responsible for picking the flowers. The n^{th} animal is called the "sink", and is responsible for arranging the flowers in the basket. The animals form a line and each animal (besides the last animal in line), upon obtaining a flower, throws the flower to the next animal in the line. Every animal, depending on his or her physical capabilities, has a maximum rate at which they can throw flowers to the next animal. However, the travel time of a flower (when thrown) is instantaneous.

Given this system for flower collection, the animals would like to figure out the "maximum flower flow" of the system. The maximum flower flow is simply the maximum rate at which flowers can be collected (passed from the source to the sink). It is assumed, for the purposes of calculating the maximum flower flow, that both the flower picking at the beginning of the line and the flower arranging at the end of the line are instant (take no time). Can you help the forest creatures figure out how quickly they can collect flowers?

The Problem:

Given the flower-throwing speed of each animal in the assembly line, calculate the maximum flower flow of the system.

The Input:

The first line contains a single, positive integer, t, representing the number of possible assembly lines the animals would like you to consider. Each assembly line is given on multiple lines. The first line contains a positive integer, n ($2 \leq n \leq 100$), representing the number of animals in the line. The next n lines each contain a string, s, with at least 1 and no more than 10 alphanumeric characters, followed by a single space and an integer, x ($1 \leq x \leq 100$), representing the name of the animal and how quickly that animal can throw flowers (in flowers/minute), respectively. The animals are listed in the input in the same order in which they line up.

The Output:

For each assembly line, output a single line containing "`Assembly #a: f`" where a represents the number of the assembly line (starting with 1) and f represents the maximum flower flow (in flowers/minute) of that assembly line as an integer.

Sample Input:

```
2
2
Deer 15
Rabbit 15
3
Bluebird 20
Chipmunk 11
Skunk 8
```

Sample Output:

```
Assembly #1: 15
Assembly #2: 11
```

Circular Sums

Filename: `sums`

The people of Oddville often play math games. Today, they are playing a game called "Circular Sums."

To play Circular Sums, they arrange themselves in a circle, and each person writes down a nonnegative integer on a piece of paper. Then, the players take turns in clockwise order around the circle. During their turn, a player erases the number on his or her paper and writes down a new number, the sum of the numbers on all the other players' papers. Play continues in the same way around the circle indefinitely until they decide to stop.

Since they are rather odd people, they get really excited and give a cheer whenever someone writes down an odd number in Circular Sums. In addition, their cheers have drawn a crowd that watches the game and cheers along with them whenever an odd number is reached. Each member of the crowd watches the game for some consecutive range of turns and measures the amount of fun they have as the number of cheers they are a part of.

The mayor of Oddville likes to keep track of the happiness of his citizens. He wants you to help him by writing a program to determine how much fun each member of the crowd has.

The Problem:

Given the initial state of Circular Sums and the range of turns in which each crowd member watches the game, determine how much fun each crowd member has.

The Input:

The first line of the input contains a single integer, t, representing the number of games of Circular Sums to process. Following this line are t descriptions of Circular Sums. Each game description begins with a line consisting of two integers, n and m ($3 \le n \le 15$; $1 \le m \le 10^4$), representing the number of players in the game and the number of people in the crowd, respectively.

The next line contains n space-separated integers where the i^{th} number, a_i ($0 \le a_i \le 10^6$), represents the number that the i^{th} player initially writes on his or her paper. The players are given in clockwise order around the circle, and the player that starts the game is the first player given in the input.

The next m lines of a game description each contain two integers, s_j and e_j ($1 \le s_j \le e_j \le 10^9$), meaning that the j^{th} crowd member starts watching the game just before turn s_j and stops watching the game just after turn e_j.

The Output:

For each game of Circular Sums, output "`Game #i:`" where i is the game number, starting at 1. Then, output m lines where the j^{th} line contains the amount of fun that the j^{th} crowd member has, in the same order as they were given in the input. Finally, output a blank line after each game.

Sample Input:

```
2
3 3
0 0 1
1 2
3 7
100 100
5 1
10 5 17 14 23
20 30
```

Sample Output:

```
Game #1:
1
4
1

Game #2:
9
```

Nurikabe Checker

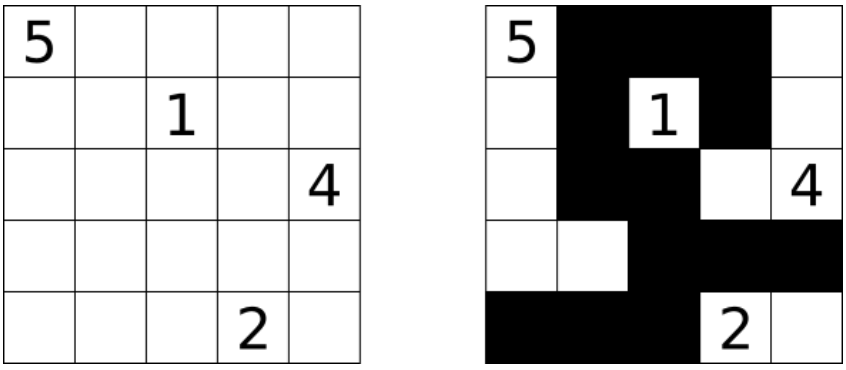
Filename: `nurikabe`

Evan loves solving logic puzzles. One type of puzzle that he finds particularly interesting is called Nurikabe.

In a Nurikabe puzzle, you are initially given a grid where the cells are either empty or contain a positive integer. To solve the puzzle, you must fill in some of the cells. Cells that are filled in are considered black, while cells that aren't filled in are considered white. Two cells of the same color are connected if they are directly or indirectly connected, that is, if they share a side horizontally or vertically, or if there is a path of directly connected cells from one to the other. Each group of connected white cells is called an *island*, and each group of connected black cells is a *stream*. Furthermore, a correctly solved puzzle must satisfy the following constraints:

- Cells with numbers cannot be filled in.
- Each island must contain exactly one cell with a number, and that number must be equal to the number of cells in the island.
- Every black cell must be connected directly or indirectly with every other black cell (i.e. there can only be one stream).
- There can be no 2x2 squares of black cells (called *pools*) in the stream.

The figure below shows the Nurikabe puzzle given in the first sample case (left) and its correct solution (right).



Evan worked on lots of Nurikabe puzzles before he realized that he didn't completely understand the rules! He needs help deciding if his completed puzzles are correct, or if he needs to try again.

The Problem:

Given a completed Nurikabe puzzle, determine if it is solved correctly.

The Input:

The first line of input contains a single, positive integer, p, representing the number of puzzles. Following this line are p descriptions of completed puzzles. Each puzzle description begins with a line with three integers, n, m and k ($1 \leq n \leq 300$; $1 \leq m \leq 300$; $1 \leq k \leq n * m$), representing the number of rows, the number of columns, and the number of numbered cells, respectively. Rows are numbered beginning from top to bottom, and columns are numbered from left to right. Following this are n lines, each containing m characters. These lines represent the completed puzzle, where the characters are one of the following:

- "`.`": A white cell
- "#": A black cell

The following k lines describe the white cells with numbers. Each of these lines contains three integers, r_i, c_i and v_i ($1 \leq r_i \leq n$; $1 \leq c_i \leq m$; $1 \leq v_i \leq n * m$), representing the row number, column number, and value of the i^{th} numbered cell, respectively. No two numbered cells will be given at the same position in the puzzle, and all numbered cells will be associated with a white cell. As a consequence of these constraints, the input will contain at least k white cells.

The Output:

For each puzzle, output "`Puzzle #`*i*`: `" where i is the puzzle number, starting at 1. Then output "`Correct`" if the puzzle is correctly solved, or "`Incorrect`" otherwise.

(Sample Input and Sample Output are on following page)

Sample Input:

```
2
5 5 4
.###.
.#.#.
.##..
..###
###..
2 3 1
1 1 5
3 5 4
5 4 2
5 6 4
#####.
.##...
.#.##.
.#..##
.#.##.
1 6 6
4 1 4
4 3 4
5 6 1
```

Sample Output:

```
Puzzle #1: Correct
Puzzle #2: Incorrect
```

Troll Crossing

Filename: `troll`

Travis the Troll lives under a bridge. Every day, Paul, an evil vacuum cleaner salesman (not really evil, Travis just hates vacuum cleaners), brings his shipment of vacuums across Travis's bridge. Angered by this, Travis fights back! On a good day, Travis manages to dump Paul's vacuum shipment into the river below, making the shipment a loss. On a bad day, however, Paul gets his shipment past Travis successfully.

Travis's good and bad days are determined by his strength. His strength starts at a number, *s*, but this number is cut in half (rounding down) after each day's shipment. Travis is having a good day if his strength is an odd number, and a bad day otherwise (he likes chaos, even is too balanced for him).

Paul transports *n* vacuums the first day, and then doubles his shipment size each following day to compensate for Travis's interference (regardless of the outcome of the day).

The Problem:

Travis knows he cannot stop them all, but he wants to know how many vacuums he can stop. Can you help him?

The Input:

There are many possibilities for Travis's situation, so the first line of input will have a single, positive integer, *t*, representing the number of situations to process.

Each of the following *t* lines will describe a situation, consisting of two non-negative integers, *s* ($s < 2^{31}$) and *n* ($n < 2^{31}$), representing Travis's initial strength and the size of Paul's first day shipment, respectively.

The Output:

For each situation, output one line of the form "`Situation #`*i*`: `*v*" where *i* is the situation number (starting at 1) and *v* is the total number of vacuums Travis can dump into the river in that situation. The value of *v* will always be less than 2^{31}.

Sample Input:

```
2
1 7
27 35
```

Sample Output:

```
Situation #1: 7
Situation #2: 945
```

Can I Have a Dollar?

Filename: `dollar`

Stephen is a pretty good guy — especially because it's REALLY easy to ask Stephen for a dollar. If you ask Stephen for a dollar multiple times, he always alternates between two responses:

"Uhh, I dunno…" which means he doesn't give you a dollar.
"Uhh, I guess…" which means he does give you a dollar.

For each person, he always starts with "Uhh, I dunno…" because he doesn't really like giving out money.

The Problem:

Given the number of times someone asks Stephen for a dollar, calculate how much money Stephen will give that person.

The Input:

The first line of the input will begin with a single, positive integer, *n*, representing the number of people asking Stephen for money. Each situation will contain one integer on a new line, *x* ($0 \leq x \leq 10{,}000$), corresponding to how many times that person will ask Stephen for a dollar.

The Output:

Output "`Person #`*`i`*`: $`*`m`*" for each person *i* in the input (starting with 1) where *m* corresponds to the amount of money Stephen gives that person.

Sample Input:

```
5
2
6
75
13
1
```

Sample Output:

```
Person #1: $1
Person #2: $3
Person #3: $37
Person #4: $6
Person #5: $0
```

The Little Bird

Filename: `bird`

One day, while playing in her backyard, a little girl came across an injured baby bird. Like any good kid would do, she picked it up and took it home and decided to take care of it until it got better. After it healed, she decided to give it flying lessons in her backyard. The girl's back yard is well-fenced, and the bird is safe inside the fence. However, she lives near the woods, which are filled with cats and foxes and thorny trees and many other things that might hurt the helpless little bird, and if the little bird were to land on or outside the fence, there's no telling what might happen to it! As a result of these dangers, the girl knows that until it is a strong flier, it will have to remain confined to the yard, so she wants to make sure that the yard is absolutely safe for the bird to practice in.

She knows that because the bird is still learning, it can fly at most a certain distance away from her before getting tired and landing. However, the bird may fly in any direction away from her when she releases it, and the yard is considered unsafe if flying in any direction could result in the bird landing on or outside the fence. The girl will be standing inside of her yard, at coordinates (0, 0) when she releases the bird. She also knows that her parents love geometry and have fenced their yard in such a way that it is a polygon whose internal angles are all less than 180 degrees.

The Problem:

Given the coordinates of the corners of the fence in clockwise order, and the maximum distance that the little bird can fly, determine whether or not a given yard is safe for the bird.

The Input:

The first line will contain a single, positive integer, t, representing the number of yards to be checked. Each yard will begin with a line containing two integers, r and n ($1 \leq r \leq 1{,}000$; $3 \leq n \leq 100$), representing the maximum distance that the little bird can fly and the number of corners in the fence, respectively. The next n lines will each contain two integers, x and y ($-1{,}000 \leq x \leq 1{,}000$; $1{,}000 \leq y \leq 1{,}000$), representing the coordinates of a corner of the fence.

The Output:

For each yard, output "`Yard #`*`i`*`: `*`m`*" where i is the number of the yard (starting with 1) and m is the message "`Fly away!`" if it is safe for the bird to fly in the given fencing layout or the message "`Better not risk it.`" if not.

Sample Input:

```
2
10 4
8 8
8 -8
-8 -8
-8 8
3 4
-6 7
7 8
9 -6
-7 -8
```

Sample Output:

```
Yard #1: Better not risk it.
Yard #2: Fly away!
```

Roughly the Size of a Barge

Filename: `eggs`

Gaston, a famous historical figure, was known among his peers for his unusually large neck and for his love of eggs. In fact, when he was a lad, he ate four dozen eggs every morning to help him get large. Gaston was a "lad" from ages 0 to 17, and became "grown" when he turned 18. However, once he was grown, he decided this was not good enough. His wish was to become roughly the size of a barge, approximately 350,000 ft^3. To accomplish this, he took advantage of his larger stomach and began eating five dozen eggs every morning. Once he achieved this goal, however, he stopped eating eggs for breakfast.

The Problem:

Given the age in years and volume in cubic feet of Gaston when he woke up on a particular morning, determine the number of dozens of eggs Gaston ate for breakfast that morning.

The Input:

The first line of the input contains a single, positive integer, *b*, representing the number of breakfasts to analyze. Each of the next *b* lines contains two integers, *a* ($0 \leq a \leq 30$) and *v* ($1 \leq v \leq 350{,}000$), representing the age of Gaston in years and the volume of Gaston in cubic feet, respectively. You may assume that if $a < 18$, then *v* will be less than 350,000.

The Output:

For each breakfast, output a line containing "`Breakfast #`*b*`:` *d*" where *b* is the number of the breakfast (starting with 1) and *d* is the number of dozens of eggs that Gaston should eat that morning.

Sample Input:

```
4
17 500
18 15000
28 350000
19 20000
```

Sample Output:

```
Breakfast #1: 4
Breakfast #2: 5
Breakfast #3: 0
Breakfast #4: 5
```

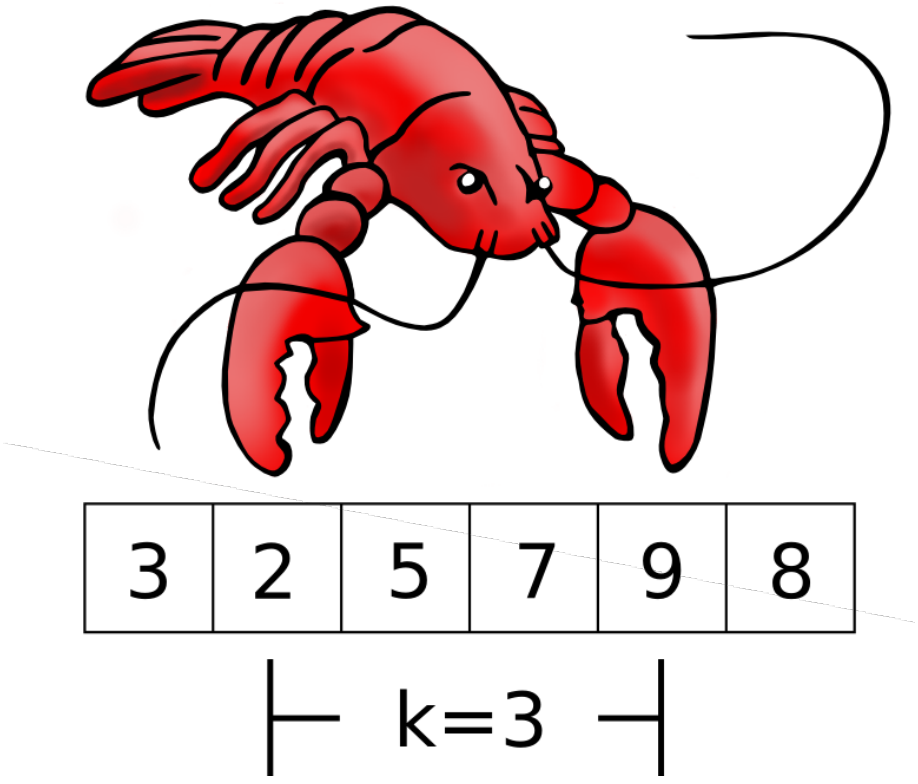
Swap Lobster

Filename: `lobster`

Lobsters comprise a group of large marine crustaceans. They typically have long bodies, muscular tails, and spend their lives in crevices and burrows on the sea floor. Many of their pairs of legs have claws, but they are well known for their front claws. While there are some well known varieties across the world, one kind you may never have heard of is the swap lobster!

A swap lobster is a lobster that likes sorting arrays. Swap lobsters come in many different sizes. A swap lobster of size *k* can swap two elements in an array that are exactly *k* distance apart. An array is considered sorted if for each pair of adjacent elements, the number on the left is less than or equal to the number on the right.

Despite the fervor with which swap lobsters approach sorting, it is an unfortunate truth that not every array can be sorted by every swap lobster. Consider, for example, a size 3 swap lobster attempting to sort the array [3, 2, 5, 7, 9, 8]. No amount of swapping elements a distance of exactly 3 apart will ever see this array sorted, yet the pitiable crustacean is none the wiser. Watching the poor creature in its futility, you decide it'd be best to write a program to tell whether an array is sortable for a particular lobster.



The Problem:

Given an array, determine if it can be sorted by a swap lobster of size *k*.

The Input:

The first line of the input contains a positive integer, *s*, representing the number of swap lobsters. Each swap lobster will begin with a line containing two integers, *n* and *k* ($1 \leq k < n \leq 50$), representing the size of the array and size of the swap lobster, respectively. On the following line will be *n* integers, $v_1, v_2, \ldots, v_n$, where v_i represents the i^{th} integer in the array ($1 \leq v_i \leq 500$).

The Output:

For each swap lobster, output "`Lobster #d: `" where *d* is the number of the lobster, starting from 1. Follow this by "`Sortable`" if the array can be sorted, or "`Unsortable`" if it cannot.

Sample Input:

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

Sample Output:

```
Lobster #1: Unsortable
Lobster #2: Sortable
Lobster #3: Sortable
Lobster #4: Unsortable
```