Twenty-sixth Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
Zombie Spellcheck	zombie
Mack and Zack Run on the Track	track
Snail Shell	shell
France is Bacon	bacon
Reverse Batarang	batarang
May the 4th Be With You	force
Lagrange Street	census
Price Wars	price
Allowance or Aquarium?	aquarium
Jimmy's Perfect Vacation	vacation
It's Good to be the Chief!	chief

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

For example, if you are solving Mack and Zack Run on the Track:
Call your program file: track.c, track.cpp or track.java
Call your input file: track.in
Call your Java class: track

Zombie Spellcheck

Filename: `zombie`

It was a dark and stormy night (very much like this one), when a terrible incident more than mortified Mortimer the mortician. The dead had come back to life! Mortimer found himself barricaded within his office, listening to the shuffling and moaning of the living dead.

"BRAAAAAIIIIINNNSSSSS... BRAAAAAAAAAAIIIIIIIIIINNNNNNSSSSSS...!"
"BRAAAIIINNNSSSS! BRAAAAIIIIINNNNSSSSS!!"
"NNNNNAAAABRRRIIIISSSSS!!!"

Wait a minute, what was that last one? Did that zombie just say "nabris?" Mortimer frowns in disappointment as he reaches for his dictionary. Looks like these zombies are going to need a spellcheck.

The Problem:

Given a dictionary of words and the moans of the flesh-craving mob outside of Mortimer's office, output a list of possible words the scatterbrained zombies could have intended to utter. Scholars long ago discovered that zombies tend to drag out each letter of a word and that two words are equivalent if they have the same *number* of maximal "runs" of each letter. A run is a sequence of consecutive identical characters in a word. A maximal run is simply a run whose adjacent characters differ from its contents. For example, the word "CAREER" is composed of 5 maximal runs ("C", "A", "R", "EE", "R"). "AAARRRCCCCCEEEERRR" also breaks into 5 maximal runs ("AAA", "RRR", "CCCCC", "EEEE", "RRR"). Since both words have an equal number of runs of each letter (1 "A", 1 "C", 1 "E", 2 "R" and 0 for all other letters), they are a match.

The Input:

Mortimer would like this spellcheck to be reusable by future morticians, in case this sort of thing ever happens again. As such, input will begin with a single positive integer, t, the number of zombie spelling scenarios. A scenario begins with a line containing a single positive integer, d ($d \leq 1000$), the size of the dictionary, which is in turn followed by d distinct words, each on its own line. A word consists of anywhere from 1 to 30 uppercase alphabetic characters. Following the dictionary is a line containing a single positive integer, q ($q \leq 100$), the number of zombie utterances. Following this are q lines, each containing a single utterance. An utterance consists of anywhere from 1 to 100 uppercase alphabetic characters.

The Output:

For each scenario, print the header "`Scenario #`*i*`:`" on its own line, where i is the scenario number (starting at 1). For each utterance in this scenario, if it could not be unscrambled using the dictionary print "`No matches found.`" Otherwise, print "`Did you mean:`" on its own line, followed by a list of possible matchings sorted in alphabetical order, each followed by a single "`?`". Print each matching on its own line and in upper case. Print a single blank line after the output for each utterance.

Sample Input:

```
2
10
BRAINS
BRIANS
CAREER
CARER
DEER
DREER
RACE
RACECAR
RACER
RED
4
RRRRAAAABBBBSSSSIIIINNNN
AAAARRRCCCCEEEERRRR
PSYCHOLINGUISTICS
DER
3
DEER
DREER
RED
1
DER
```

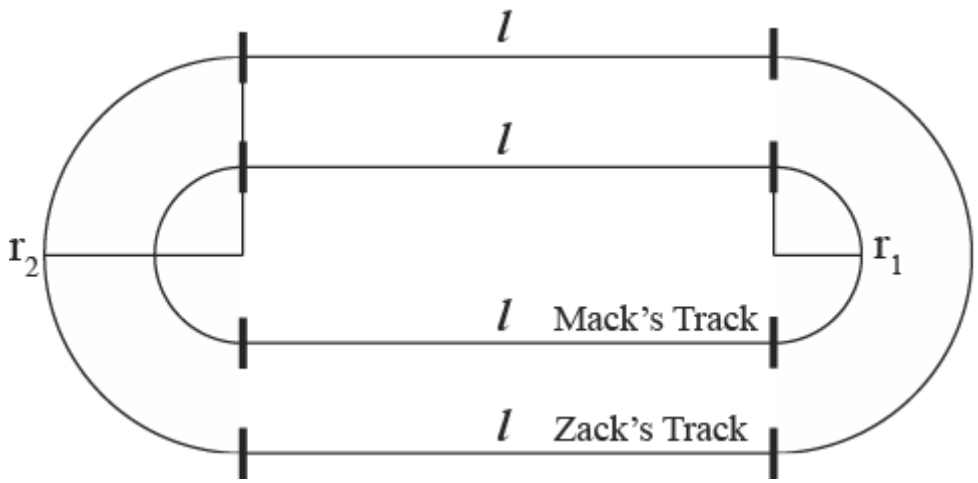
Sample Output:

```
Scenario #1:
Did you mean:
BRAINS?
BRIANS?

Did you mean:
CAREER?
CARER?
RACER?

No matches found.

Did you mean:
DEER?
RED?

Scenario #2:
Did you mean:
DEER?
RED?
```

Mack and Zack Run on the Track

Filename: `track`

Mack and Zack have competed in their share of sports. Their most recent obsession has been running. Both Mack and Zack enjoy running around the track at the local high school. Each of them wants to brag to their dad about their performance every day. The trouble with this is that they run in different lanes on the track, so it's difficult to determine who has run a longer distance. In particular, each lane of a track has two straightaways of equal length and two semicircles, one at each end. Mack always runs in the inner lane of the track, while Zack chooses the outer lane:



When Mack brags to his dad, he always mentions how many laps he's run. Unfortunately, Zack never runs more laps than Mack (because his lap is longer!), but sometimes he does run farther than Mack. Your goal will be to determine if Zack ran farther than Mack, and if so, create a boast Zack can use to retort to Mack mentioning how many laps he ran.

The Problem:

Given the dimensions of the lane where both Mack and Zack run, as well as the number of laps each of them has run, determine if Zack has run farther than Mack. If he hasn't, no retort can be created, since Mack has run more laps and farther than Zack. But, if Zack has run farther, create a retort exclaiming how many more meters Zack has run than Mack. If you need to use π, use a value of 3.141592653589793.

The Input:

The first line of input will contain a single positive integer, *n*, representing the number of tracks that Mack and Zack run on. The following *n* lines will contain descriptions of each track and how many laps Mack and Zack ran, one line per description.

The first positive integer on each of these lines, l ($l \leq 200$), will designate the length of the straightaways of the track (in meters). The second positive integer on each of these lines, r_1 ($r_1 \leq 100$), will be the radius of the inner lane of the track where Mack runs (in meters). The third positive integer on each of these lines, r_2 ($r_1 < r_2 \leq 200$), will be the radius of the outer lane of the track where Zack runs (in meters). The fourth positive integer on each of these lines, m ($m < 1000$), represents the number of laps Mack runs. The final positive integer on each of these lines, z ($z < m$), represents the number of laps Zack runs. Each number on each of these lines will be separated by a single space.

The Output:

For each track, output a header with the following format:

`Track #`*k*`:`

where *k* represents the number of the track considered (starting with 1).

If Mack has run farther than Zack, follow this with the exclamation, "`Drats!`".

But, if Zack has run farther than Mack, follow this with a statement of the following form:

`I've run `*x*` more meters than Mack!!!`

where *x* represents the number of meters Zack has run in excess of Mack, rounded to the nearest integer. It is guaranteed that for each track this excess distance does not have a fractional part in between .49 and .51 (one of the brothers runs at least .51 meters more than the other), so that there will be no close rounding cases.

Sample Input:

```
2
100 50 100 3 2
150 20 100 50 2
```

Sample Output:

```
Track #1: I've run 114 more meters than Mack!!!
Track #2: Drats!
```

Snail Shell

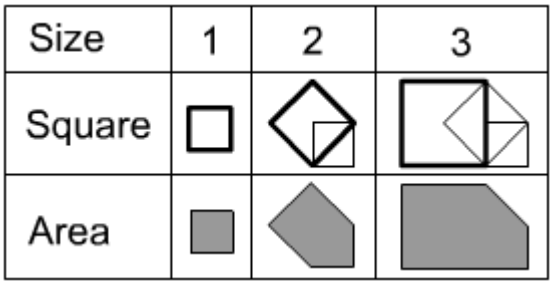
Filename: `shell`

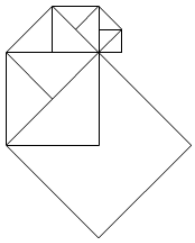
Speedy the Snail is one of the fiercest competitors in the underground molluscan deathrace scene. The main event is fast approaching and his last shell was absolutely totaled in the previous race. The tabloids had gone wild about the maneuver in which he recklessly jumped a twig to cut off a threatening advance from rival Banana Slug Joe. As you can imagine, it can get pretty hectic out there on the track, and so Speedy needs his new shell to be able to take a few hits. You hear the squeal of burning mucus as Speedy pulls up outside your shop, and know he can only be here for one reason.

The Problem:

Speedy isn't sure which size shell he'd like to drive in the big race tomorrow, so he needs you to write a program that will tell him the durability of a specific size shell. As a revered shellsmith, you're quite aware that a shell's durability is roughly equivalent to its area. Additionally, you know that a snail shell of a particular size can be generated in the following manner:

1. A snail shell of size 1 is made by placing the lower-left corner of a square with side length 1 on the origin.
2. A snail shell of size n ($n > 1$) is made by building up from a snail shell of size n-1. A new square is placed along the diagonal of the last square placed in the shell of size n-1, with side length equal to the length of the diagonal, overlapping exactly one half of the previous square. All shells are built out from the previous shell in a counter-clockwise direction. The first 3 shells and their actual areas as well as the size 6 shell are shown in the following chart.





Size 6 shell

A shell's area is defined as the area of the region enclosed by its overlapping squares. Since Speedy wants to know the durability of several shells, you think it may be best if you wrote a program that can output the area of any shell.

The Input:

The first line of the input file will contain a single positive integer, *s*, the number of shell specifications in which Speedy is interested. Following this will be *s* lines, each with a single positive integer, *n* ($1 \leq n \leq 50$), representing the size of a snail shell.

The Output:

For each snail shell, output the line "`Shell #`*i*`: `*a*", where *i* is the number of the shell specification (starting at 1), and *a* is the surface area of that shell, rounded to three decimal places (as an example of rounding, an area of 1.5455 would be rounded up to 1.546, whereas an area of 1.5454 would be rounded down to 1.545).

Sample Input:

```
5
1
2
7
8
15
```

Sample Output:

```
Shell #1: 1.000
Shell #2: 2.500
Shell #3: 95.500
Shell #4: 191.000
Shell #5: 24448.000
```

France is Bacon

Filename: `bacon`

My story[1] goes:

When I was young my father said to me:

"Knowledge is Power….Francis Bacon"

I understood it as "Knowledge is power, France is Bacon."

For more than a decade I wondered over the meaning of the second part and what was the surreal linkage between the two? If I said the quote to someone, "Knowledge is power, France is Bacon" they nodded knowingly. Or someone might say, "Knowledge is power" and I'd finish the quote "France is Bacon" and they wouldn't look at me like I'd said something very odd but thoughtfully agree. I did ask a teacher what "Knowledge is power, France is Bacon" meant and got a full 10 minute explanation of the "Knowledge is power" bit, but nothing on "France is Bacon." When I prompted further explanation by saying "France is Bacon?" in a questioning tone I just got a "yes." At 12 years old, I didn't have the confidence to press it further. I just accepted it as something I'd never understand.

From then on, while I now understood the meaning of "Knowledge is power" I continued to be puzzled over "France is Bacon" and used them interchangeably for a long while, thinking they were equivalent.

It wasn't until years later I saw it written down that the penny dropped.

The Problem:

Given a sentence, determine how many times "France" would be replaced by "Bacon" within it. Note that "France" is a proper noun so any instances with differing capitalization should be ignored (for example, "france" should not be replaced by "bacon"). However, we do not care if "France" is a part of another word (so words such as "Frances" would be replaced with "Bacons").

[1] Origin unknown.

The Input:

Input will begin with a single, positive integer, *n,* on a line by itself, representing the number of sentences. On each of the next *n* lines, a single sentence will be given (each sentence will be a maximum of 80 characters). The only punctuation marks that might be a part of the input are the period, question mark, exclamation point, comma, quotation mark and apostrophe. Spaces at the front and end of a line as well as multiple consecutive spaces will not occur.

The Output:

For each line of input, output the line "`Sentence #`*i*`: `*r*" where *i* is the number of the sentence (starting with 1) and *r* is the number of times that "France" would be replaced with "Bacon" in that sentence (if we were to do the replacement).

Sample Input:

```
11
For more than a decade I wondered over the second part.
France is Bacon or Bacon is France?
What was the surreal linkage between the two?
If I said "Knowledge is power, France is Bacon" they nodded.
Or I'd finish the quote "France is Bacon" and they would agree.
I did get an explanation of the Knowledge is power bit.
But nothing on "France is Bacon."
When I prompted further explanation by saying "France is Bacon?"
I just got a "yes."
At 12 years old, I didn't have the confidence to press it.
I just accepted it as something I'd never understand.
```

Sample Output:

```
Sentence #1: 0
Sentence #2: 2
Sentence #3: 0
Sentence #4: 1
Sentence #5: 1
Sentence #6: 0
Sentence #7: 1
Sentence #8: 1
Sentence #9: 0
Sentence #10: 0
Sentence #11: 0
```

Reverse Batarang

Filename: `batarang`

The Joker's at it again! This time he's taken over the Gotham City playing card factory, threatening to leave all of Gotham one card short of a full deck! Once again, it's up to Batman to bring the Prince of Chaos to justice. Fortunately, the dark knight just so happens to have an ace up his sleeve.

Batman first has to clear the facility of Joker's thugs. Normally this would be a trivial matter thanks to the Bat's trusty arsenal of batarangs. Unfortunately, for this particular outing Alfred only packed the dubious reverse batarang, a tool Batman hardly ever uses. He'd feel a lot better if he knew the minimum number of reverse batarangs he has to throw to clear each room.

The Problem:

Batman can throw a batarang to any location, after which it will return to him in a straight line and knock out every thug in its path. The initial throw is harmless, as the reverse batarang is only lethal on the return path. Given the locations of each thug in the room relative to Batman, determine the minimum number of reverse batarangs he will have to throw to knock out each thug. Note that Batman must catch the batarang once it returns, as letting it continue its flight would be irresponsible and may hurt someone's parents.

The Input:

The first line of the input file will contain a single positive integer, r, the number of rooms in the playing card factory. Following this will be r room descriptions. A room description will begin with a single positive integer, n ($1 \leq n \leq 100$), on a line by itself, representing the number of thugs in this room. Following this will be n lines in the format x y ($-1000 \leq x, y \leq 1000$, x and y will never both be 0), representing the x and y coordinates of a thug relative to Batman.

The Output:

For each room, output "`Room #`*i*`: `", where *i* is the room number (starting at 1), followed by one of two reports. If Batman will need more than one batarang to clear the room, output "`Batman will only need` *x* `batarangs!`" where *x* is the smallest number of batarangs necessary to knock out every thug. If, however, Batman can clear the room with a single batarang, instead output "`Leaping lizards,` *n* `birds with one stone!`", where *n* is the number of thugs in the room. Print a blank line after each room report.

Sample Input:

```
3
4
1 1
2 2
1 0
5 0
3
1 1
-3 -3
8 -4
4
-1 1
-7 7
-49 49
-343 343
```

Sample Output:

```
Room #1: Batman will only need 2 batarangs!

Room #2: Batman will only need 3 batarangs!

Room #3: Leaping lizards, 4 birds with one stone!
```

May the 4th Be With You

Filename: `force`

Sergio is just ecstatic! He is a huge Star Wars fan that looks forward to today, Star Wars Day, more than anything! He loves to dress up as Chewbecca and play Holochess. Obviously, dressed up as a Wookie, he has quite a good winning record.

What makes Sergio even more excited about today is that he's also a computational physicist! He just loves to compute force! Of course, he does it on a massive scale with lots and lots of processors running in parallel. However, he wants to spread the love of his work to all so he is requiring you to compute force in a simple manner using simple forces on a single object in motion. He reminds you that Newton's Second Law of Motion is:

> "*The acceleration **a** of a body is parallel and directly proportional to the net force **F** and inversely proportional to the mass **m**.*"

Or in other words, **F** = **ma**.

The Problem:

Given the mass and acceleration of an object, compute the force.

The Input:

Input will begin with a single, positive integer, *n*, on a line by itself, representing the number of objects for which Sergio wants you to compute the resultant force. Following this there will be *n* lines, each representing one of those objects. Each object will include two positive integers, *m* and *a*, that represent the mass and acceleration of the object, respectively. You are guaranteed that the resulting force will also be an integer.

The Output:

For each object, output the header "`Force #i:` " (where *i* is the number of the force, starting with 1) followed by the force of the object.

Sample Input:

```
2
10 15
3 5
```

Sample Output:

```
Force #1: 150
Force #2: 15
```

Lagrange Street

Filename: `census`

It's census time. Unfortunately, many people don't like being part of a census. The worst street for taking a census is Lagrange Street. Why is Lagrange Street so bad you ask? That's because on this street live the families of the most prominent mathematicians in the world. Let's look at a case from a previous census. When asked about the ages of her children a mother replied, "I have three children, the product of their ages is 36, the sum of their ages is equal to the address of the house next door." The census taker walked over to the house next door and read the number 13 off the garage. Our volunteer walked back to the house asking for more information. However, the mother replied "I have to go. My oldest child is sleeping upstairs."

With this last piece of information the census taker knew the ages of the children have to be 9, 2, and 2. This is because the other possible assignment of ages before learning there was an oldest child was 6, 6, and 1. Knowing that there is an oldest child makes the solution to the problem unique.

To help make census taking on Lagrange Street easier you have been hired to write a computer program to take information for a house on Lagrange Street and print out one of three options. The information may make it so no assignment of ages fits the description, or there may be a unique assignment of ages, or there may be more than one possible assignment of ages. Each age assigned must be a positive integer and there can be any number of children as long as the assignment meets all the requirements.

In addition to the sum and product of the ages you will be given several more facts in the following form:

Fact	Meaning
`has Oldest Child`	The oldest child's age is unique.
`has Youngest Child`	The youngest child's age is unique.
`has Middle Child`	There exists a middle child and the middle child's age is unique.
`has Triplets`	There exists at least one age with exactly three children.
`has Twins`	There exists at least one age with exactly two children.
`has` *`n`* `Children`	The household has exactly *n* children total.

Note that in this problem middle child means a child who has an equal number of younger and older siblings.

The Problem:

Given a description of a family household on Lagrange Street, analyze whether the ages of the family's children can be uniquely determined.

The Input:

The first line of input will be an integer, h, representing the number of houses for which to gather census information. For each house there will be one line with three integers s, p, and f ($1 \le s \le 100$; $1 \le p \le 1{,}000{,}000$; $1 \le f \le 6$), which stand for the sum of the ages, the product of the ages and the number of facts, respectively. Following the line will be f lines where each line contains a fact from above. For the "`has n children`" fact the given n will be between 1 and 10, inclusive.

The Output:

For each house in the input, output a line with the message "`House #h:` " where h is the number of the house (starting from 1). On the next line, output either "`Not Enough Information`", "`Contradictory Information`" or a list of ages separated by a single space. Output "`Not Enough Information`" if there exists more than one age assignment that meets the requirements. Output "`Contradictory Information`" if no age assignment can satisfy all the facts. If the age assignment is unique then output the ages of the children in order of increasing age. Output a blank line after each house report.

(Sample Input and Sample Output on following page)

Sample Input:

```
4
13 36 2
has Oldest Child
has 3 Children
13 36 2
has Youngest Child
has 3 Children
1 1 4
has Oldest Child
has Youngest Child
has 1 Children
has Twins
13 36 1
has 3 Children
```

Sample Output:

```
House #1:
2 2 9

House #2:
1 6 6

House #3:
Contradictory Information

House #4:
Not Enough Information
```

Price Wars

Filename: `price`

You've gotten a job at the local grocery store and your boss has asked you to get some pricing information for the nearest competitor. You are to go over to their store and track the price of several of their items on different days. From that data, you are supposed to figure out the price that occurs the most often (the *mode*) within the sample of days, so that your boss has a good idea what price to expect from her competitor.

The Problem:

Given prices of an item for several days, determine which price occurs most frequently.

The Input:

The first line of input will contain a single positive integer, n, representing the number of items to price. The data for each item follows on the next n lines. The first value on each of these lines will be a positive integer, k ($k \leq 100$), representing the number of days for which you have price data for that item. This is followed by k positive integers less than 1,000,000, representing the price of the item on each of the k sample days in cents. All of the numbers on each line will be separated by a single space.

The Output:

For each item, output a single line of the following format:

`Item #x: Most likely price is c cents.`

where x is the item number (starting at 1) and c is the most frequent price of that item in cents, according to the days the price was taken. It is guaranteed that the most likely price will be unique; namely, for each item, there will be one number that appears strictly more times than any other number.

Output a blank line after the output for each item.

Sample Input:

```
2
5 99 129 99 99 99
10 529 609 499 529 519 609 439 459 609 599
```

Sample Output:

```
Item #1: Most likely price is 99 cents.

Item #2: Most likely price is 609 cents.
```

Allowance or Aquarium?

Filename: `aquarium`

Now you've gone and done it! That shady character, Ila (Ali's arch-nemesis), convinced you to part with your whole week's allowance, and what do you have to show for it? A virtual aquarium with some assembly required? It's a good thing you're such a brilliant programmer and may be able to breathe some life into these goldfish; they must literally be worth their weight in gold!

You unfold the sheet of paper the masked man handed you and find a set of instructions. You skim the page to discover that the aquarium is a rectangular grid of blocks (Of course! Ali would have used a perfectly round fishbowl, but not Ila!), each of which contains either water or coral. The aquarium is home to several species of fish varying in size. Regardless of their physical size, however, each fish may occupy exactly one block of water. The aquarium manufacturers have provided a starting configuration for the fish tank, so you're not too worried about that. The instructions go on to describe fish behavior.

The aquarium updates in time steps, so the state of the aquarium changes instantaneously from one step to the next. There are, however, two phases to each time step. The first of these is movement. Each fish comes pre-equipped with a command sequence from which a single command is read in order at each time step. Fish read the first command in the sequence on the first time step, the next command on the next time step, and so on. Once a fish reaches the end of its command sequence, it will start over from the beginning of its sequence. A command may tell a fish to move in any of four directions: up, down, left, or right. If a command would move a fish into a block of coral or out of the tank, then that fish remains stationary instead (it will, however, still move to the next command in the next time step).

Once all fish have moved, the feeding phase begins. If two fish occupy the same space, one will eat the other. The larger of the two fish will be the predator, and the smaller the prey. The predator will increment its size by 1, and the prey will be removed from the aquarium. Note that there is at most one fish in each cell following movement and feeding.

If two or more fish of the same size occupy a cell, then a tie-breaker must be performed. In the event more than two fish collide in a cell, the largest set of fish will always move first. A tie may be broken based on how the fish moved in the previous phase, using the following list of priorities.

Down > Right > Left > Up > Stationary

A stationary fish is one whose movement command on this time step would have taken it out of the aquarium or into a block of coral.

The Problem:

Given a description of an aquarium, including fish and their movements, determine the state of the aquarium after a given number of timesteps.

The Input:

The first line of the input file will contain a single positive integer, *a*, representing the number of aquariums. Each aquarium will begin with a line with four integers, *r, c, n* and *t* ($1 < r, c < 20$; $0 \le t \le 100$; $1 \le n \le r*c$), where *r* and *c* are the aquarium dimensions, *n* is the number of fish, and *t* is the number of time steps for which to run the aquarium, respectively. The aquarium grid follows on the next *r* lines, containing *c* characters each, either "`.`" or "`X`" to denote water or coral, respectively. The fish descriptions follow on the next *n* lines, each of the form "*`Name s x y Sequence`*", where *Name* is a string containing between 1 and 30 alphabetic characters, *s* is a positive integer representing the size of the fish, *x* and *y* are the fish's starting coordinates ($1 \le x \le r$, $1 \le y \le c$), and *Sequence* is a string containing between 1 and 100 characters. A command sequence will only contain the characters "`U`", "`D`", "`L`", and "`R`", specifying up, down, left, and right, respectively. The top-left cell of the aquarium has coordinates (1, 1) and the bottom-right cell has coordinates (*r*, *c*). It is guaranteed that no two fish will have the same starting coordinates, and no starting coordinates will contain a block of coral.

The Output:

Preface each aquarium with a line of the format "`Aquarium #`*`i`*`: `" where *i* is the aquarium number (starting at 1). For each fish in the order given in the input, print a line with "*`f`*`. `" where *f* is that fish's number (starting at 1). If that fish has been eaten output "` (Deceased)`". Then, regardless of whether the fish was eaten or not, output the word "`Big `" *n* times where *n* is the number of victims the fish ate during the *t* time steps, followed by the fish's name. Separate aquariums with a blank line.

(Sample Input and Sample Output on next page)

Sample Input:

```
2
5 5 3 10
....X
.....
XXX..
..X..
..X..
Isolated 1 5 1 URDL
Predator 10 1 1 RRRDRRDDUU
Prey 3 5 5 UUULRDDD
3 6 4 10
......
..XX..
......
Stationary 1 2 5 L
SmallPredator 2 3 4 RUULLLDDRR
HugePredator 3 1 1 URRR
Snack 1 1 6 RL
```

Sample Output:

```
Aquarium #1:
1. Isolated
2. Big Predator
3. (Deceased) Prey

Aquarium #2:
1. (Deceased) Stationary
2. (Deceased) Big SmallPredator
3. Big Big HugePredator
4. (Deceased) Snack
```

Jimmy's Perfect Vacation

Filename: vacation

Jimmy and his family are planning a vacation. They all love amusement parks, and they always try to go on every ride that the park has. Unfortunately, they only have a certain amount of time in the day. Not only does timing make things complicated, some of the paths between two rides have been blocked off for parades, construction, and mothers tending to crying babies. To make matters even worse, Jimmy is very sensitive to rides and throws up every time he gets off, so he never wants to go near a ride he has already ridden. They could write out by hand to figure out the fastest way to visit each ride, but Jimmy has a good friend (you)! Help Jimmy and his family find the fastest way to visit all of the rides. Of course, they will be going on a lot more vacations, so your program should be able to handle multiple vacations.

The Problem:

Given each ride the amusement park has, as well as a list of blocked paths between two rides, find the fastest time it will take for Jimmy and his family to visit each one. Assume that he and his family walk at 1 meter per second, and they can always travel to their destination in a straight line assuming that the path is not blocked. Rides always take a constant time of 120 seconds per ride (they always visit the parks during the off-season so the waiting time is negligible).

The Input:

Input will begin with a single integer, n, which is the number of different parks Jimmy and his family have planned to visit. For each park, there will be two non-negative integers, r and b, which are the number of rides and blocked paths respectively ($r < 11$, $b < r^2$). On the following r lines are two integers, x and y, denoting the Cartesian coordinates of the ride (in meters). The rides are numbered from 1 to r for simplicity, and no two rides will have the same coordinates. After these r lines are b lines, each giving two integers i and j stating that the path between ride i and ride j is blocked, and therefore, the path from j to i is blocked as well. All coordinates will be non-negative integers less than 1,000, and Jimmy and his family always start their visit at the entrance, which is at the origin, (0,0). There will never be a blocked path that includes the entrance, and you may never revisit the entrance (they consider that exiting the park and make you buy another ticket!). Assume there is a path between every pair of rides unless it was blocked.

The Output:

For each vacation, first output a header, on a line by itself, stating "`Vacation #`k`:`" where k is the number of the vacation, starting with 1. If Jimmy can and his family can ride every ride, output a new line "`Jimmy can finish all of the rides in `s` seconds.`" where s is the number of seconds rounded to the nearest thousandth of a second it takes to ride every ride (as an example of rounding, a time of 100.5455 would be rounded up to 100.546, whereas a time of 100.5454 would be rounded down to 100.545). If Jimmy and his family cannot ride every ride, output one line saying "`Jimmy should plan this vacation a different day. `" instead. Output a single blank line after the output for each vacation.

Sample Input:

```
3
4 0
0 2
2 2
2 4
4 4
5 4
10 10
12 35
64 60
3 7
100 857
1 2
1 3
1 4
1 5
5 2
0 5
0 10
0 20
0 50
0 25
3 4
1 2
```

Sample Output:

```
Vacation #1:
Jimmy can finish all of the rides in 488.000 seconds.

Vacation #2:
Jimmy should plan this vacation a different day.

Vacation #3:
Jimmy can finish all of the rides in 670.000 seconds.
```

It's Good to be the Chief!

Filename: `chief`

Hao! I'm the Chief Judge. Welcome to the UCF High School Programming Tournament!

As Chief Judge, I have many responsibilities. One of these is trying to form a sentence from the filenames in the problem set (there are never enough verbs!). Another is making sure that the problem set is well-balanced and that each problem is solvable (but not too easy!). These tasks can be tedious, so this year, I have decided to determine if the set is solvable by transforming each problem into objects in a game! In this game, you will be given a ball (a perfect sphere) of a certain radius. Each of the problems will become a block (a box with straight sides and all right angles) of some size. Harder problems become bigger blocks. Your task is to roll your ball around the playing area and try to collect all of the blocks. The catch is that you can only collect a problem if the center of mass of your ball is higher than the center of mass of the problem's block. Once you collect a problem, your ball expands in volume by the volume of the problem's block. A bigger ball allows you to collect bigger problems! If you can collect all of the blocks, then the problem set can be solved. Sounds like fun, right?

Remember, the volume of a sphere is $V = \frac{4}{3}\pi r^3$, where r is the radius, and the volume of a box is $V = lwh$, where l is length, w is width, and h is height. From the ground, the center of mass of a sphere is as high as the radius of the sphere, and the center of mass of a box is as high as one half the box's height. When using π, use a value of 3.141592653589793.

The Problem:

Given a number of different problem sets, modeled as a collection of blocks, determine if each problem set can or cannot be solved if the optimal order of collection of the blocks is used.

The Input:

The input will begin with a single, positive integer, *n*, on a line by itself. This represents the number of problems sets to examine. Each problem set begins with a line containing a positive integer, *p* ($p \leq 10{,}000$), and a positive real number, *r*, separated by a single space, representing the number of problems, and the initial radius of the ball, respectively. Following this will be *p* lines, each containing three real numbers, *l*, *w*, and *h*, separated by single spaces, representing the length, width, and height of the block for that problem.

The Output:

For each problem set, first output the header "`Problem Set #`*i*`: `" where *i* is the number of the problem set in the input (starting with 1). Following this (and on the same line) output the phrase, "`It's going to be a good set!`" if all of the problems can be "solved" by collecting their respective blocks with the ball. If the set cannot be "solved" in this way, output "`We need to rebuild this!`" instead.

Sample Input:

```
2
4 5.0
10.0 10.0 10.0
10.0 10.0 10.0
10.0 10.0 10.0
10.0 10.0 10.0
4 5.0
20.0 20.0 1.0
20.0 20.0 2.0
20.0 20.0 3.0
20.0 20.0 4.0
```

Sample Output:

```
Problem Set #1: We need to rebuild this!
Problem Set #2: It's going to be a good set!
```