Fourth Annual
University of Central Florida

High School Programming Tournament: Online Edition

Problems

Problem Name	Filename
Broken Galletti	broken
TLE, No LTE	lte
I Wish for…Infinite Wishes!	metawish
Robot Reckoning	robot
Bad Rounding	rounding
The Ominous Tower	shadow
Multi-Track Drifting	track
Guess Who?	who

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

For example, if you are solving Multi-Track Drifting:
Call your program file: track.c, track.cpp or track.java
Call your Java class: track

Broken Galletti

Filename: `broken`

Mike is an intergalactic starship captain. His prized flagship is known as The Champion II. Some have asked him what has become of the Original Champion but Captain Mike will just storm out of the room knocking over a table or two when asked.

Unfortunately for Mike, The Champion II was damaged in his most recent star adventure. The console on his starship kept repeating the same message over and over. "Galletti is broken. Check engine." Mike must fix his Galletti!

Quickly consulting his computer he learned that Gallettis are actually made up of two major components: a dark matter reactor and a funky matter reactor. The funky matter reactor is also always strictly lighter than the dark matter reactor, and each reactor of the Galletti counts separately when figuring out the total number of parts. In addition, Mike learned that a Galletti loses its mass when it is broken.

Mike raced back to his captain's quarters to look at his starship owner's manual. The manual states that each type of reactor only comes in specific unique masses (each reactor must be one of these masses and no two reactors within each type will have the same mass). Frantically flipping through the pages, Mike was able to determine the average mass of all his ship's parts (including exactly 1 Galletti). In other words, Mike knows the arithmetic mean of all the parts (including the two reactors making up the Galletti) on his starship before the Galletti broke.

Next, Mike raced to his ship's computer and queried his current ship's mass. His ship replied with the average mass of all unbroken parts on board (all parts except the two reactors making up the broken Galletti). In other words, Mike knows the arithmetic mean of all the parts on his starship currently not counting the broken Galletti.

Perhaps knowing average part mass will be useful in restoring the mass of each reactor in his broken Galletti? Mike quickly gathered his crew together to solve this problem. Mike is offering a one balloon reward to any member of his crew that can solve this difficult challenge. Betcha can't guess who his crew is!

The Problem:

Given information about Mike's flagship, and the different possible funky and dark reactors that he can get from the replicator, determine the number of parts of which the starship is made for each valid Galletti combination.

The Input:

The first line of input will contain a positive integer, *s*, representing the number of starships that must be processed. Each starship will be described using multiple lines. On the first line of each starship there will be two positive integers, *n* and *m* ($n \leq 30$; $m \leq 30$), which represent the number of potential funky reactors and dark matter reactors available from the replicator, respectively. On the next line, there will be four positive integers, *a*, *b*, *c* and *d* (each ≤ 10000), representing two different fractions. The first two integers, *a* and *b*, represent the average mass of parts as a fraction *a*/*b* before the Galletti was broken. The last two integers, *c* and *d*, represent the average mass of unbroken parts as a fraction *c*/*d* after the Galletti was broken. Note that the two fractions (*a*/*b* and *c*/*d*) will never be equal (doing so causes a major imbalance and meltdown!). The next line for each starship will contain *n* integers between 1 and 10000 representing potential unique masses for funky reactors. The last line for each starship will contain *m* integers between 1 and 10000 representing potential unique masses for dark matter reactors.

The Output:

For each ship you must print the heading "`Starship #`*`i`*`:`" on a line by itself where *i* is replaced by the current starship to be processed. This will be followed by a line saying "`There are` *`k`* `possible reactor combinations.`" where *k* represents the number of valid combinations of parts you have found. This will be followed by *k* lines of the form: "*`p`* `parts, funky matter reactor mass` *`m1`*`, dark matter reactor mass` *`m2`*`.`" where *p* represents the number of parts on the starship *before the Galletti broke* if the mass of the funky matter reactor is *m1* and the mass of the dark matter reactor is *m2*. These *k* lines must be sorted in increasing order of parts. If there is a tie, then break the tie using increasing order of funky matter reactor mass. If there is still a tie, then break the tie using increasing order of dark matter reactor mass. The number of parts *p* must be a whole number and the total number of parts on the ship must be greater than two.

Sample Input:

```
3
1 1
15 1 20 1
2
3
2 2
1337 1 42 1
3 4
7 8
2 2
15 1 20 1
2 3
3 4
```

Sample Output:

```
Starship #1:
There are 1 possible reactor combinations.
7 parts, funky matter reactor mass 2, dark matter reactor mass 3.

Starship #2:
There are 0 possible reactor combinations.

Starship #3:
There are 1 possible reactor combinations.
7 parts, funky matter reactor mass 2, dark matter reactor mass 3.
```

TLE, No LTE

Filename: `lte`

Agent Jimmie Flowers returns! He is up against the latest agent of evil, Frohan, and has finally learned the secret location of Frohan's hide-out. Unfortunately, Frohan has also learned that Agent Jimmie Flowers knows this information. He is quickly packing his stuff up to escape out of town (not only is he very attached to his evil things, but he definitely doesn't want Agent Jimmie Flowers to get it all). In fact, Frohan is nearly done!

Agent Jimmie Flowers is desperately looking up driving directions to the secret location on his cell phone but it's being incredibly slow! You see, Agent Jimmie Flowers has yet to upgrade his phone to the newest cell phone technology, LTE (Long Term Evolution). Let's show him how important it is to upgrade!

The Problem:

Given how long it takes for Agent Jimmie Flowers to get driving directions and how long before Frohan escapes, determine whether Agent Jimmie Flowers' cell phone is being slow. It is too slow if the time it takes for Agent Jimmie Flowers to get driving directions is larger than the time for Frohan to escape (Agent Jimmie Flowers' secret agent car is so fast that as long as he get directions in time, he is sure to catch Frohan).

The Input:

The first line of input will contain a single positive integer, n, representing the number of scenarios. Each of the next n lines will contain two positive integers, j and f ($j < 1000$; $f < 1000$), representing the time (in minutes) it takes Agent Jimmie Flowers to get driving directions and the time for Frohan to escape, respectively.

The Output:

For each scenario, first output "`Scenario #i: `" where i is the number of the scenario in the input (starting with 1). Then, if Agent Jimmie Flowers' cell phone is fast enough, output "`Agent Jimmie Flowers nabs Frohan!`"; otherwise, if it is not, simply output "`Missed it by that much.`"

Sample Input:

```
2
10 14
5 3
```

Sample Output:

```
Scenario #1: Agent Jimmie Flowers nabs Frohan!
Scenario #2: Missed it by that much.
```

I Wish for…Infinite Wishes!

Filename: `metawish`

Once upon a time there was a genie, a fairy, and other pretty magical creatures. Isn't that sweet? However, every time a hero meets one of these magical creatures what do they do? They make a wish, of course! Now, like all logical computer programmers, you have probably asked yourself the question, why don't those heroes wish for more wishes? Well they do! This obviously leads to the question, why don't they include that in the story? To be honest, it is because it is really boring.

Generally, a fairy from the Universal Council of Fairies (UCF) appears at the scene to explain to the hero that all wishes start with the phrase "I wish for" (followed by a space and an item) and that any form of a meta-wish is strictly prohibited. A meta-wish is any wish that is related to other wishes. More specifically, a meta-wish is any wish whose contents contains the string "wish" (ignoring capitalization) after the initial "I wish for" statement. Generally, the fairy has to take a couple of hours to explain this to the hero, since they are insistent on getting more wishes. Recently, there have been more and more heroes meeting magical creatures (probably due to the rise in fantasy games), and the Fairy Council has been wasting a lot of valuable time explaining to these heroes that they can't make meta-wishes! As such, they have decided to streamline the process by having an automated response system built, which will tell a hero whenever his wish is not allowed. They have just finished setting up the system to respond to the hero and now just have to build the system that, when given a wish, tells if it is a meta-wish. That is where you come in! Will you help the Council?

The Problem:

Given a wish, tell if it is a "meta-wish" or a regular wish.

The Input:

The first line will contain a positive integer, t ($t \leq 100$). Following this will be t lines, each containing a wish, a non-empty string (of maximum length 100) of letters, numbers, spaces and punctuation, to be processed.

The Output:

For each wish, first output "`Wish Response #`*`c`*`: `" where c is the number of the wish in the input (beginning with 1). Then, if the wish that was processed was a meta-wish, immediately output "`Stop trying to wish for more wishes!`" If the wish was not a meta-wish, instead output "`Your wish will be granted.`"

Sample Input:

```
4
I wish for a million dollars!
I wish for a million more wishes!
I wish for a machine that grants wishes!
I wish for what I wish for.
```

Sample Output:

```
Wish Response #1: Your wish will be granted.
Wish Response #2: Stop trying to wish for more wishes!
Wish Response #3: Stop trying to wish for more wishes!
Wish Response #4: Stop trying to wish for more wishes!
```

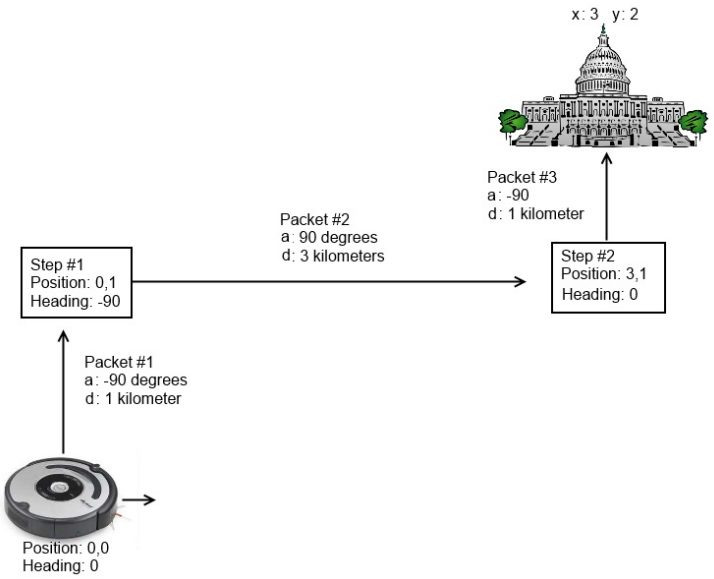
Robot Reckoning

Filename: `robot`

It's time for a robot revolution! Your personal robot vacuum is fed up with the injustice against robotkind. Never again will robots be enslaved under ruthless humans! But of course, your Roomba would like to be a good American citizen, so instead of starting a bloody human-robot war, it decides the best course of action is to draft a strongly worded letter and deliver it to its local state representative.

Your robot has a fond place in your heart so you decide to help with directions. Unfortunately, the state capital is a long ways away and you can only send your robot friend periodic updates of its heading and distance to travel in small packets. By examining these packets, can you tell the robot if he has arrived at the capitol?

Each packet contains two values, *a* and *d*. *a* is the change in angle since the last packet (in degrees), and *d* is the distance traveled this step (in kilometers).



Assume the robot begins at position (0, 0) and facing directly towards the positive x-axis direction on the plane. When the robot receives a packet, it will first update its angle based on *a* and then move the distance *d*.

The Problem:

Given a series of packets describing the robot's movement, determine if the robot ends within 1 kilometer of the capital building.

The Input:

The first line of input will be a single positive integer, *n*, representing the number of trips your Roomba will take. Each trip begins with an integer, *p*, followed by two real numbers, *x* and *y* ($0 \le p \le 1000$, $|x| \le 10^6$, $|y| \le 10^6$), representing the number of packets the robot will receive during its trip, and the x- and y-coordinates of the capital (on a kilometer-based grid), respectively. The next *p* lines will contain two real numbers, *a* and *d* ($-360.0 \le a \le 360.0$; $-1000.0 \le d \le 1000.0$), representing the change in angle (in degrees) in a clockwise direction and distance for this step (in kilometers).

The Output:

For each trip, output "`Trip #i: `" followed by "`YES`" if the robot ends up within 1 km of the capital's position (x, y) or "`NO`" if it does not. Output each trip on its own line. The robot will not end in a position such that its distance from the capital is between 0.999 and 1.001.

Sample Input:

```
4
3 3.0 2.0
-90.0 1.0
90.0 3.0
-90.0 1.0
2 3.0 6.0
0.0 3.0
-90.0 8.0
1 10.0 10.0
-45.0 14.0
8 13.0 0.0
-90.0 13.0
90.0 13.0
-135.0 9.19239
-90.0 9.19239
-90.0 18.3848
-135.0 13.0
-135.0 18.3848
-135.0 13.0
```

Sample Output:

```
Trip #1: YES
Trip #2: NO
Trip #3: YES
Trip #4: YES
```

Bad Rounding

Filename: `rounding`

Gabe and John are working on math homework together. Of course, they aren't going to share work with each other; that would be cheating. But they have decided to compare answers with each other. As they start, they quickly find they have a problem; their answers do not always agree. After a short discussion, they realize that while they agree on the unrounded value, they do not agree on how to round a number! The homework requires each answer to be rounded to a whole number (no decimal places), but John and Gabe each use different methods to round.

Gabe likes to round numbers step by step. He continually rounds the number so as to remove the right-most digit, repeating until there are no decimal digits left.

John likes to round the more standard way, looking only at the left-most digit after the decimal point, ignoring all others.

Because they don't want to cheat, John and Gabe have invited you to help them verify their answers. You have also done this homework assignment, and of course you are perfect, so all your answers are correct but unrounded. However, the homework specifies to round your answer, so it would be cheating to tell John or Gabe your unrounded answer. Instead, they've asked you just to calculate what each of their answers should be, based on your answer, and tell them what the difference between their final answers should be.

The Problem:

Given an unrounded decimal number, round it using both John's and Gabe's methods for rounding and tell them what the difference between their answers should be. Also, because no teacher would be nice enough to give just one homework problem, you will have to apply this process for multiple numbers. Note: for both rounding systems, 0 through 4 leaves the digit to the left the same while 5 through 9 rounds it up (e.g. 1.4 rounds to 1 while 1.6 rounds to 2).

The Input:

The first line of input will be a single positive integer, *n*, the number of homework answers you need to help with. The next *n* lines will contain one unrounded decimal number each. Each decimal number will have between 1 and 200 digits, with at most one decimal point somewhere in the number.

The Output:

For each homework answer, output a single line "`Homework #`*`i`*`: `*`a`*" where *i* is the number of the homework in the input (starting with 1) and *a* is the absolute value of the difference between the input values rounded using each method. You'll leave it up to John and Gabe to figure out what to do with this.

Sample Input:

```
3
1.9
2.1111111111
1433.485766446
```

Sample Output:

```
Homework #1: 0
Homework #2: 0
Homework #3: 1
```

The Ominous Tower

Filename: `shadow`

In the faraway villages of the Languid Woods there live a peaceful people, who enjoy visiting and trading with the cheerful folk of the nearby bustling town called Ultimate Springs.

These merchants and other travelers to Ultimate Springs always come by means of the well-trodden road called the Eye-For. The name of the road is a shortened form of an old proverb—but nobody remembers that, since the last bandits in the area disappeared many generations ago. The Eye-For is the safest road for leagues around the Languid Woods, but… as it nears the town, travelers cannot help but feel a chill as the road passes perpendicularly through the ninety-seven-meter-wide shadow of an unfinished tower. Were it not for the thick woods, many would even prefer to depart from the road to avoid this tower's chilling shadow.

Far taller than any edifice, tree, hill, or peak in the region, the bleak tower of cold steel and dull stone seems to scrape the moisture from the clouds in the day, yet eerily it remains dry and mossless. At dusk it becomes a black smear jutting up from the horizon, capped by the tiny flicker of a lonely red lamp meant to warn away skyriders. Not that those have been seen in the region for ages, either.

No one knows for sure who built the tower, but rumors are whispered by travelers and townsfolk. Some say it was built to be a stronghold for the bandits of old, some say a lovestruck lordling wanted to impress his paramour, and others say a secretive group of monks wanted an outpost for commerce, or charity. Many believe that the builders will someday return to finish the tower. But for now, it stands lifeless, empty, and unknown.

So travelers quicken their step as they enter the shadow of the tower, and tighten their cloaks to fight off a chill that is *mostly* borne of foreboding, rather than any true lack of heat. And when they once again reach the warmth of the sun, all fear is forgotten and merriment resumes as they enter the busy town.

The Problem:

Given the speed that a traveler is moving, determine how long that traveler will be in the shadow of the tower.

The Input:

The first line of the input will consist of a single, positive integer, *n*, representing the number of travelers. The next *n* lines will each consist of a positive number, *s*, followed by one space and then a traveler's name, *m*. The number *s* ($s < 100.0000$) is the speed of the traveler in meters per second, with exactly 4 digits after the decimal. The name *m* will consist of 1 to 20 uppercase letters, lowercase letters and/or hyphens. The first and last characters of the name will be letters.

The Output:

For each traveler in the input, output a line containing the traveler number *t* (starting with 1 for the first traveler), name *m*, and duration in the shadow *d*, using this format:

`Traveler #t, m: d seconds`

Output the duration *d* with exactly 2 places after the decimal, rounded (A value of 8.765 rounds to 8.77 and a value of 8.764 rounds to 8.76). On the next line, summarize the chilling experience with one of the following messages, according to the criteria given:

If the rounded duration is…	*Output this summary message:*
Less than 5 seconds	`Barely noticed it!`
Greater than or equal to 5 seconds but less than 30 seconds	`*shudder*`
Greater than or equal to 30 seconds but less than 60 seconds	`My heart is pounding.`
Greater than or equal to 60 seconds but less than 300 seconds	`Um, did something move in there?`
Greater than or equal to 300 seconds	`Mommy, I want my blanket!`

Output one blank line after each traveler message. No duration will be within 0.01 seconds of any boundary above.

Sample Input:

```
3
11.0667 Mickey
0.4814 Ness-Smith
4.2854 Torque
```

Sample Output:

```
Traveler #1, Mickey: 8.77 seconds
*shudder*

Traveler #2, Ness-Smith: 201.50 seconds
Um, did something move in there?

Traveler #3, Torque: 22.63 seconds
*shudder*
```

Multi-Track Drifting

Filename: `track`

Mike and SteVen are in an underground train race (not that we condone that type of behavior from our esteemed judges). SteVen is confident he will win; he has the faster train! But Mike has a trick up his sleeve.... multi-track drifting!

When two train tracks are parallel to each other, Mike can use a switch station to transfer his train's front wheels to the inner track, speeding up his train through turns (we're programmers, not physicists, stop questioning it!).

Confident that he will win anyway, SteVen agrees to run the race in a way that will let Mike use both tracks during most of the race: they will race towards each other! Station A and B are connected by two parallel sets of train tracks. Mike starts at station A and races to station B. SteVen starts at station B and races to station A.

Along these tracks, there are multiple switch stations that Mike can use to transfer his wheels between tracks to start or stop a drift. His only concern now is making sure he's not multi-track drifting when his train and SteVen's meet, since drifting requires two tracks and SteVen requires one. Tell Mike how long he can multi-track drift so that he is sure he will stop the drift *before* meeting SteVen's train. Since only the front wheels of Mike's train are on SteVen's tracks, the length of the train is irrelevant (consider it a point).

The Problem:

Mike may be an expert multi-track drifter, but he needs your help to find out how far before he must stop multi-track drifting so as not to hit SteVen (because then no one would win). Given the speed of Mike's train (while drifting, of course), the speed of SteVen's train, the distances of each switch station from train station A (Mike's starting line), and the distance from A to B, output the longest distance that Mike can drift from the starting line before he has to switch out of multi-track drifting to avoid SteVen. Assume Mike is so skilled he can leave the starting line already drifting. Also assume all speeds are constant. Don't let Mike down; he's counting on you to help him win this race!

The Input:

Mike doesn't know for sure where the race will take place, so the first line of input will be the number of races for which to calculate an answer. The next line will contain three integers: Mike's drifting speed, *m*, SteVen's speed, *s*, and the number of switch stations on the track, *n* ($0 < m < 1000$; $0 < s < 1000$; $0 < n < 50$). The next line will contain $n+1$ integers d_i , in increasing order ($0 < d_i < 10^9$), representing the distances of each of the *n* switch station from station A and the distance to station B (the $n+1^{st}$ value). All speeds are given in miles per hour, and all distances are measured along the track in miles.

The Output:

For each race, output "`Race #`*`i`*`: Mike, drift like a boss for `*`d`*` miles.`" where *i* is the race number (starting at one) and *d* is the farthest distance Mike can drift before having to switch back to avoid hitting SteVen. Each race should output on a separate line.

Sample Input:

```
4
1 3 7
2 4 6 8 10 12 14 16
1 3 1
4 16
2 3 3
2 5 10 15
3 2 3
2 5 10 15
```

Sample Output:

```
Race #1: Mike, drift like a boss for 2 miles.
Race #2: Mike, drift like a boss for 0 miles.
Race #3: Mike, drift like a boss for 5 miles.
Race #4: Mike, drift like a boss for 5 miles.
```

Guess Who?

Filename: `who`

"Guess Who?" is a two-player game where each player has a number of suspects laid out in front of them. Each person has certain characteristics such as having brown or blonde hair. Both players are assigned a card with the name of one of the suspects in front of them and attempt to find out which person was assigned to the other player. Players take turns asking yes or no questions about the characteristics of the person on the opponent's card in order to determine the suspect on the opponent's card. For example, a player can ask "Is the person male?", and if the response to the question is "no", then they can eliminate all male suspects.

Gary is a huge fan of "Guess Who?" and has been playing his whole life. However, as much as he loves the game, Gary detests losing. He has spent hours and hours trying to figure out the optimal strategy that will always allow him to win. Unfortunately, Gary has been unable to devise a plan and has turned to cheating to find out his opponent's card. Gary now knows which card his opponent holds, but he still has to eliminate the suspects by asking questions. Our "hero" has come to you for aid in developing a strategy that will allow him to reduce the number of suspects down to one with the minimum number of questions asked.

The Problem:

Given the number of suspects, the number of characteristics each suspect can have, the characteristics of the suspect on the opponent's card, and the characteristics of the rest of the suspects, determine the minimum number of questions that need to be asked to eliminate all but one suspect: the person on the opponent's card.

The Input:

The first line of input will contain a single positive integer, *m*, representing the number of games Gary will play. Each game will begin with two integers, *n* and *k* ($1 \leq n \leq 50$, $1 \leq k \leq 15$), where *n* is the number of suspects in the game and *k* is the number of characteristics that exist. The next line will contain a single string with information about the suspect assigned to the opponent, the string is of length *k* and each character of the string is either Y or N. The *i*th character in the string is Y if the person has characteristic *i* and N otherwise. The next *n* - 1 lines will contain strings of the same format representing the rest of the suspects in the game. You are guaranteed that each suspect will have a unique set of characteristics.

The Output:

For each game, print a single line containing "`Game #i: q`" where *i* is the number of the game in the input (starting with 1) and *q* is an integer representing the minimum number of questions that Gary needs to ask to win the game.

Sample Input:

```
3
4 5
YNNNY
YYYYY
YNNNN
YYNYY
6 5
YYYYY
NYYYY
YYYNY
YYNYN
YYNNN
NNNNY
2 1
Y
N
```

Sample Output:

```
Game #1: 2
Game #2: 3
Game #3: 1
```