# Twenty-second Annual
# University of Central Florida

# High School Programming Tournament

## *Problems*

| Problem Name | Filename |
|---|---|
| Ali's Ice Cold Conundrum | ice |
| Balance That Equation | balance |
| Sorting Student Presentations | sorting |
| This is SPARTA! | sparta |
| The Non-Orthogonal Structure of Tuscany | tower |
| FaceSpace Friends | friends |
| Ali's Analog Clock | hands |
| Double Double: Scoops with Trouble | scoop |
| Pepe's Perfect Pizza Palace | pizza |
| Montgomery Anaconda and the Sacred Chalice | chalice |
| Lucas' Letters | letters |

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

For example, if you are solving Lucas' Letters:
Call your program file: letters.c, letters.cpp or letters.java
Call your input file: letters.in

# Ali's Ice Cold Conundrum

*Filename:* `ice`

Ali is throwing a party at one of his houses, and he's laid out a big selection of drinks for everyone. It's pretty hot outside, so the ice maker is cranked up to the max. However, as Ali is dispensing ice, he notices that the ice maker tends to release ice cubes in small bursts of two to ten cubes each. Being the consummate host, Ali knows that seven cubes is the perfect number of cubes for the glasses he's using, but he's having trouble hitting that mark with the dispenser.

**The Problem:**

Given the number of ice cubes dispensed in each burst in order, tell Ali whether or not each cup has the perfect number (7 cubes). If not, let Ali know how many cubes he would have to remove to get the perfect number. Since his guests are thirsty, he won't waste time actually correcting the number of ice cubes in each cup (he just wants to keep track of how he's doing). Once each cup is full (7 or more cubes), Ali will automatically move on to the next cup. Ali will keep filling cups with ice as long as the ice maker dispenses cubes (once it runs out, Ali will wait for it to fill up again). Note that once the ice maker runs out of ice, the last cup might not have enough ice cubes in it. If so, let Ali know how many cubes that cup is missing. After each session, he will immediately serve all of the cups he has filled (whether they have 7 cubes or not).

**The Input:**

The ice maker will be filled and dispensed several times today. The input will begin with a single positive integer, $s$, on a line by itself, indicating the number of ice dispensing sessions. Each session will start with an integer, $b$ ($1 \le b \le 50$), on a line by itself, indicating the number of ice dispensing bursts in this session. Following this, on each of the next $b$ lines, there will be a single integer, $c$ ($2 \le c \le 10$), indicating the number of ice cubes in this burst.

**The Output:**

At the start of each session, output a line of the form "`Session #`*n*`:`", where $n$ is the number of the ice dispensing session (beginning at 1). For each filled cup in this session output a line indicating the state of that cup:

- If the cup has exactly 7 cubes, print "`Cup #`*i*`: Perfect!`"
- If the cup has too many cubes, print "`Cup #`*i*`:` *k* `cubes too many!`"
- If the cup has too few cubes, print "`Cup #`*i*`: Need` *k* `more cubes!`"

In all cases, $i$ is the number of the filled cup in that session (beginning at 1), and $k$ is the absolute value of the difference between the number of ice cubes dispensed into that cup and 7. The output line for each cup should be indented three spaces and there should be one space after the colon. Leave a blank line after the output for each dispensing session.

**Sample Input:**

```
2
3
10
2
5
4
2
2
3
2
```

**Sample Output:**

```
Session #1:
   Cup #1: 3 cubes too many!
   Cup #2: Perfect!

Session #2:
   Cup #1: Perfect!
   Cup #2: Need 5 more cubes!
```

# Balance That Equation

*Filename:* `balance`

You've just entered Chemistry I and are anxious to do well in the course. You make it through the first unit and score a 96% on the test. Great job! But now things aren't looking so great. You have to start dealing with chemistry equations, which you've always hated. In lieu of having to balance an equation every time you get it, you decide to write a program to do the balancing for you. An equation is balanced if each element contains the same number of atoms on the reactants side (the left side) as it does on the products side (the right side).

To calculate the number of atoms an element has, you multiply the coefficient in front of the compound in which that element appears times the subscript of the element. For simplicity, you may assume that a compound is a grouping of *one* or more elements. Note that a coefficient that leads a compound applies to all elements in that compound (e.g. $2\ HCl_2$ has $2 \times 1 = 2$ *H* atoms and $2 \times 2 = 4$ *Cl* atoms). Consider the following examples:

Unbalanced Equations

1) $Zn + HCl \rightarrow ZnCl_2 + H_2$ ← *H* and *Cl* are unbalanced
2) $KClO_3 \rightarrow KCl + O_2$ ← *O* is unbalanced
3) $Fe + O_2 \rightarrow Fe_2O_3$ ← *Fe* and *O* are unbalanced

Balanced Equations

1) $Zn + \mathbf{2}\ HCl \rightarrow ZnCl_2 + H_2$
2) $\mathbf{2}\ KClO_3 \rightarrow \mathbf{2}\ KCl + \mathbf{3}\ O_2$
3) $\mathbf{4}\ Fe + \mathbf{3}\ O_2 \rightarrow \mathbf{2}\ Fe_2O_3$

The only valid change we can make is to change the coefficient in front of each compound—changing any element's subscript is strictly forbidden, as this would change the actual compound.

For equation 1, there is 1 atom of *Zn* in the reactants and 1 atom of *Zn* in the products, so no balancing is necessary for *Zn*. Both *H* and *Cl* have 1 atom in the reactants, but have 2 atoms in the products. We can balance this by placing a coefficient of 2 in front of *HCl* to give 2 *H* atoms and 2 *Cl* atoms on the reactants side. For equation 2, *K* and *Cl* are already balanced, but *O* is not. After balancing *O* by placing a 2 in front of $KClO_3$ and a 3 in front of $O_2$, we effectively unbalance *K* and *Cl*, so we must revisit these elements. We do this by placing a 2 in front of the *KCl* on the products side, resulting in a completely balanced equation. Equation 3 is balanced similarly.

When writing a final balanced equation, it is important to give all coefficients *in lowest terms* to ensure that all elements are in their simplest quantities possible. This means that you should divide all coefficients by the greatest common divisor (GCD) of all of the coefficients before reporting the final equation. For example, imagine you balanced an equation that resulted in the chemical equation $\mathbf{4}\ KClO_3 \rightarrow \mathbf{4}\ KCl + \mathbf{6}\ O_2$. Because all coefficients share a common divisor of 2, then all coefficients should be divided by 2. When the GCD of all coefficients is 1, then the solution is said to be in lowest terms.

**The Problem:**

Your task is to write a program which, given a set of chemical equations, balances the equations (if necessary) and gives the correct coefficients *in lowest terms*.

**The Input:**

The input will begin with a single positive integer, $n$, on a line by itself representing the number of equations you will be balancing. Each of the $n$ equations will begin with a line containing two integers, $i$ and $j$ ($1 \le i \le 10$; $1 \le j \le 10$), where $i$ represents the number of compounds on the reactants side, and $j$ represents the number of compounds on the products side. The following line will contain the chemical equation, which will have the form:

$$RC_1 + RC_2 + \ldots + RC_i \text{ -> } PC_1 + PC_2 + \ldots + PC_j$$

where $RC_x$ is Reactant Compound x and $PC_x$ is Product Compound x. Each compound will have up to five element-subscript pairs, and will be of the form:

$$E_1\ S_1\ E_2\ S_2\ \ldots\ E_k\ S_k \qquad (1 \le k \le 5)$$

where $E_x$ is Element x and $S_x$ is the Subscript of Element x. Each element will be 1 uppercase letter optionally followed by 1 lowercase letter. There will be exactly 1 space between each element, subscript, `->` and `+`. All subscripts will be between 1 and 100 inclusive. Note that not all compounds will contain the same number of elements. You are guaranteed, however, that no element will occur more than once on the reactants or products side and that every element on the reactants side will appear exactly once on the products side. You are also guaranteed that each equation can be balanced.

**The Output:**

For each equation, output a single line of the form:

$$\texttt{Equation } e\texttt{: } RCo_1\ RC_1 + \ldots + RCo_i\ RC_i \text{ -> } PCo_1\ PC_1 + \ldots + PCo_j\ PC_j$$

where $e$ is the equation number being balanced (beginning with 1), $RCo_x$ is the Coefficient of Reactant Compound x, and $PCo_x$ is the Coefficient of Product Compound x. For simplicity, your program should output all coefficients and subscripts, even when these values are 1. Use the same spacing as in the Input (no more than 1 space separating items on the output line).

**(Sample Input and Sample Output are on following page)**

**Sample Input:**

```
3
2 2
Zn 1 + H 1 Cl 1 -> Zn 1 Cl 2 + H 2
1 2
K 1 Cl 1 O 3 -> K 1 Cl 1 + O 2
2 1
Fe 1 + O 2 -> Fe 2 O 3
```

**Sample Output:**

```
Equation 1: 1 Zn 1 + 2 H 1 Cl 1 -> 1 Zn 1 Cl 2 + 1 H 2
Equation 2: 2 K 1 Cl 1 O 3 -> 2 K 1 Cl 1 + 3 O 2
Equation 3: 4 Fe 1 + 3 O 2 -> 2 Fe 2 O 3
```

# Sorting Student Presentations

*Filename:* `sorting`

Your teacher is tired of sorting the students in the class in regular alphabetical order, since this means that certain students always go first for presentations. She has an idea to mix up the order of the students by sorting them in a different manner. In particular, she will not care about the order of the letters in the last name of a student. Instead, she will put the student who has the highest number of A's in his or her last name first. If there is a tie between two students based on this value, she'll compare the number of B's in their last names. If this is tied also, she will go on to C's, then D's, etc. The second student will be found using the same method, and so on until all of the students have presented.

**The Problem:**

Your job is to sort a list of student names based on this criterion. It is guaranteed that no class has two students who have names that are anagrams of one another; thus, there will not be any ties between two students (an anagram is a different arrangement of the same exact letters).

**The Input:**

The first line of the input file consists of a single positive integer, $n$, representing the number of classes in the file. The first line of each class contains a single positive integer, $m$ ($1 < m < 100$), which represents the number of students in that class. Each of the following $m$ lines will contain the last name of one student in the class. It is guaranteed that each name will only contain uppercase letters and will be between 1 and 20 characters long, inclusive.

**The Output:**

For the $k^{th}$ ($1 \leq k \leq n$) class, output a single header line with the following format:

`Class #k ordering`

Then, output each name in the order given by the new sorting scheme, with one name per line. Put one blank line at the end of the output for each class.

**(Sample Input and Sample Output follow on next page)**

**Sample Input:**

```
2
2
JONES
LINDT
3
WALLACE
DAVIS
MADISON
```

**Sample Output:**

```
Class #1 ordering
LINDT
JONES

Class #2 ordering
WALLACE
MADISON
DAVIS
```

# This is SPARTA!

*Filename:* `sparta`

As the Persian Empire slowly expands outward due to the crowning of its new emperor, very few oppose its might. So, in a brilliant strategy, the Persians send an ambassador to the city-state of Sparta to convince the Spartans to join the Persian Empire. However, the Spartans refuse to give up their homeland, so the ambassador and the Spartan king get into a dispute and try to discover the size of their enemy's forces, even though we all know the Spartan force will always have 300 troops. This is where you come in. Because neither side has any intention of giving away the number of their forces, you have to determine, based on the tone of voice of the ambassador, how many troops the Persian side will have. The louder the ambassador yells, the larger the Persian army is. Each time the ambassador yells, he gives away more troops. As this starts, the ambassador will say "MADNESS" one or more times, each time with an integer given as the tone of voice. The tone of voice should be added to the sum of the troops thus far in the Persian army.

**The Problem:**

Given the ambassador's side of the dispute, output a summary showing the number of troops on both sides, and the Spartan king's answer.

**The Input:**

Input will begin with an integer, $n$ ($1 \leq n \leq 100$), representing the number of disputes between the ambassador and the Spartan king. This will be followed by $n$ negotiation sessions representing the multiple disputes. Each negotiation session will start with an integer, $m$ ($0 \leq m \leq 1000$), denoting the number of times the ambassador will shout at the Spartan king. The next $m$ lines will each have the word "`MADNESS`" followed by a single space and then a single positive integer, $x$ ($0 < x \leq 10000$), denoting the tone of voice.

**The Output:**

The output for each dispute should begin with "`Dispute d:`" where $d$ is the current dispute number (beginning at 1). On the next line, output "`Persian p, Spartan s`" where $p$ and $s$ are the number of troops in the Persian and Spartan army, respectively. Finally, for each dispute there should be a concluding line stating the Spartan king's retort, "`This is SPARTA!`" After this line a single blank line should be printed.

**(Sample Input and Sample Output are on following page)**

**Sample Input:**

```
3
2
MADNESS 1200
MADNESS 25
4
MADNESS 15
MADNESS 5
MADNESS 6
MADNESS 100
1
MADNESS 10
```

**Sample Output:**

```
Dispute 1:
Persian 1225, Spartan 300
This is SPARTA!

Dispute 2:
Persian 126, Spartan 300
This is SPARTA!

Dispute 3:
Persian 10, Spartan 300
This is SPARTA!
```

# The Non-Orthogonal Structure of Tuscany*

*Filename:* `tower`

There is a bit of mystery in the history of a certain tall, tilting structure located in the western Italian peninsula. No one actually knows who designed this tower, but its tilted orientation has made it an international tourist destination. Recent studies, however, have revealed the remarkable truth behind this tower. It seems that the original idea came from a church official who was a bit of a prankster. He liked to climb up to the top of the local cathedral's bell tower and try to drop eggs onto the heads of his fellow clergymen. Trouble was, the typical bell tower had vertical walls, so it was difficult to accurately aim the egg.

The clergyman pondered a bell tower that could actually tilt and swivel, allowing him to precisely line up a target below. Unfortunately, the clergyman never lived to see his dream realized, because it took over 200 years to build the tower. Even after it was built, it wasn't until the 15th century, when another Tuscan appeared (this one famous for his engineering prowess, along with a painting of a certain smiling lady) that the full tilting and swiveling functionality of the tower was realized. Of course, the machinery didn't last, and the tower is now stuck forever in the famous leaning position. It is thought that an experiment conducted by yet another Tuscan (this one a noted astronomer and physicist) contributed to the failure of the tower's machinery. Apparently, this experiment involved the dropping of cannonballs from the top of the tower (of course, the tower was only designed for the dropping of eggs).

**The Problem:**

For the sake of posterity, the Tuscan Tower Trustees want you to simulate the ideal operation of the tower, as if the clergyman had lived to see his dream realized. The clergyman can easily swivel the tower to line up a potential egg recipient, but he needs help with the precise tilt angle. Given the distance of a target from the base of the tower, calculate the tilt angle needed to vertically align the top of the tower with the target. When standing straight up at 90 degrees, the tower is 56 meters tall. Since the clergyman always drops the egg from the side of the tower nearest the ground, you should neglect the width of the tower for simulation purposes.

**The Input:**

There will be multiple targets for the prankster clergyman's eggs. The input will begin with a single integer, $t$, on a line by itself. On each of the next $t$ lines will be a single floating-point number, $d$ ($1.0 \leq d \leq 50.0$), indicating the distance of that target from the base of the tower (in meters). The precision of $d$ will not exceed five significant digits.

* Author's Note: This problem statement contains a mixture of factual information, historical conjecture, and complete fabrication. The task of distinguishing fact from fiction is left as an exercise for the reader.

**The Output:**

For each target, print a line of the form "`Target #`*p*`: `*a*` degrees`", where *p* is the number of the target (starting at 1), and *a* is the tilt angle (relative to the ground) needed to hit that target, in degrees (use a value of 3.14159265 for $\pi$). Round the angle to the nearest tenth of a degree (for example, 5.449 rounds to 5.4 and 5.450 rounds to 5.5). Leave one space between the colon and the angle.

**Sample Input:**

```
2
3.5
30.4
```

**Sample Output:**

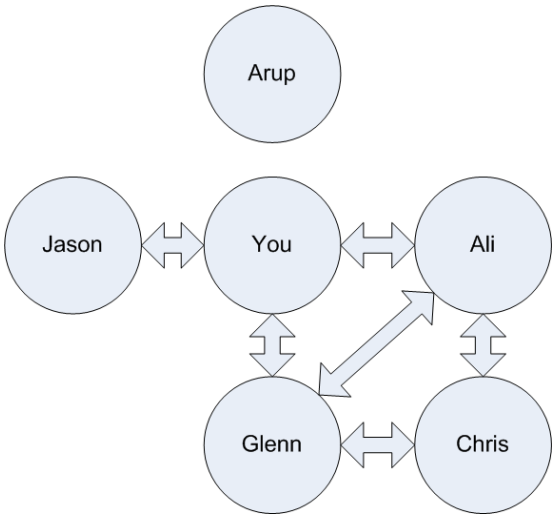
```
Target #1: 86.4 degrees
Target #2: 57.1 degrees
```

# FaceSpace Friends

*Filename:* `friends`

You've been surfing around the Internet a lot lately and have stumbled upon a variety of websites that allow you to create social networks with your friends. You've also noticed a few of your rivals have been going to these sites as well, and you are determined to show everyone how much cooler you are than them. You decide that you want a method to determine just how much cooler (or less cool) you are than your rivals.

Coolness is measured by summing the quantity of your immediate friends and the number of friends in your *extended network*. The number of friends in your extended network is calculated by adding up all of your friends, all of your friends' friends, all of your friends' friends' friends, and so on, taking care not to add any friend multiple times and not to add yourself. For example, if you are friends with Ali and Glenn, and Ali and Glenn are friends, you do not get extra coolness points for Ali and Glenn being friends since you are already friends with both. Similarly, you only get one point for any friends they share. Consider the following example, with six people registered in the FaceSpace social network:




Your friends : 3 (Jason, Ali, Glenn)
Friends in your extended network : 4 (Jason, Ali, Glenn, Chris)

Your coolness : 7

*Notice that Arup has no friends, and does not give coolness to anyone.*

**The Problem:**

Your task is to write a program which takes in an entire FaceSpace social network and a list of your rivals and determines how much cooler (or less cool) you are than each of your rivals. Note that because the social network of friends changes from week to week, you want to be able to process multiple social networks.

**The Input:**

The input will begin with a positive integer, *n*, on a line by itself representing the number of social networks which will follow. Following this will be *n* social networks, each of which includes the list of people currently registered on FaceSpace, and all of their current friendships. This will be followed by a list of your current rivals. Each social network will be separated into multiple lines as follows:

- The first line contains only a positive integer, $f$ ($1 \leq f \leq 50$), representing the number of people registered in the social network.
- The second line contains exactly $f$ names where each name is:
  - Unique within the social network under consideration.
  - guaranteed to consist of only upper and lowercase letters.
  - between 1 and 70 characters, inclusive.
  - case sensitive ("Arup" ≠ "arup").
  - separated by one space from the next/previous name.
  - one of the names in the list will always be "You" (quotes for clarity).
- The third line contains only a nonnegative integer, $c$, where $c \leq (f \times (f - 1)) / 2$, representing the number of friend connections in the social network.
- Starting at the 4th line, there will be $c$ lines containing friend connections of the form *friend1 friend2*, where:
  - *friend1* ≠ *friend2*.
  - friend connections always go both ways (that is, *friend1* is friends with *friend2* and *friend2* is friends with *friend1*).
  - *friend1* and *friend2* will be separated by one space.
  - *friend1* and *friend2* are both guaranteed to be registered in the current social network.
- On the next line will be a nonnegative integer, $r$ ($r \leq 10$), representing the number of rivals whom you want to compare your coolness with for the current social network.
- This will be immediately followed by $r$ lines, each containing only the name of a rival. Note that the only guarantee on rivals is that you will not be your own rival. There is no guarantee that your rivals will be cool enough to know about FaceSpace, and so they may not be registered at all in the social network being reviewed. In that case their coolness is, obviously, 0.

**The Output:**

For each FaceSpace social network, begin the output with the line "`Social Network i:`" where *i* is the social network being processed (beginning with 1). Follow this with $r$ lines, one for each of your rivals. Each of these lines should begin with three spaces followed by the phrase "*m*`: Difference of p point(s).`" where *m* is the name of the rival who you are currently comparing your coolness with and *p* is your coolness points minus your rival's coolness points. Each social network should be followed by a blank line.

**Sample Input:**

```
2
6
You Arup Jason Ali Glenn Chris
6
You Jason
You Ali
You Glenn
Glenn Ali
Ali Chris
Glenn Chris
2
Arup
Chris
5
You Ross Dan Amy Casey
5
You Dan
Casey Dan
Ross Casey
Amy You
Amy Casey
2
Ross
Casey
```

**Sample Output:**

```
Social Network 1:
   Arup: Difference of 7 point(s).
   Chris: Difference of 1 point(s).

Social Network 2:
   Ross: Difference of 1 point(s).
   Casey: Difference of -1 point(s).
```

# Ali's Analog Clock

*Filename:* hands

Ali has decided to build a giant, novelty analog clock, using large pieces of scrap metal that he found in the attic of his new house. He has several perfect triangular pieces that could be used as hands for the clock, but depending on the hands he selects, he's not sure he has a circular piece big enough to use for the clock face.


For each triangular metal piece used as a clock hand, he plans to attach a small hinge at the midpoint of the triangle's shortest side. The hinge will attach the hand to the exact center of the clock face, which is a perfect circle.

The point of a hand is the corner of the triangle piece that is opposite its hinged side. The clock face needs to be large enough that the points of both hands will fit entirely within the circle.

Ali wants to be absolutely certain that the sharp points of the hands won't protrude beyond the clock face, in spite of any slight errors in his original measurements or floating-point calculations. So, any point inside the circle that is calculated to be less than 0.01 meters from the circle's edge should be considered to be outside the clock face.

Note: If a triangular metal piece has more than one shortest side, any of the shortest sides can be used for the hinge.

**The Problem:**

Given the lengths of the sides of two triangular metal pieces to be used as clock hands, determine the minimum integer diameter required for the circular clock face.

**The Input:**

The first line of the input file will contain only a positive integer, *p*, indicating the number of prospective pairs of clock hands to be evaluated. The next *p* lines will each contain a description of two triangular metal pieces. Each pair of metal pieces will be represented by six positive integers less than 1,000,000, with each number separated from the next by exactly one space. The first three integers are the lengths, in meters, of the three sides of the first metal piece, and the last three integers are the lengths, also in meters, of the three sides of the second metal piece. Both pieces are guaranteed to be valid triangles.

**The Output:**

For each pair of triangular metal pieces in the input, output a line of this form:

```
Pair #n of hands requires a clock face at least d meters wide.
```

where *n* is the number of the triangle pair from the input (starting at 1) and *d* is the smallest integer that could be the diameter, in meters, of the circular clock face.

**Sample Input:**

```
2
8 8 4 7 8 14
3 4 5 9 9 9
```

**Sample Output:**

```
Pair #1 of hands requires a clock face at least 22 meters wide.
Pair #2 of hands requires a clock face at least 16 meters wide.
```

# Double Double: Scoops with Trouble

*Filename:* `scoop`

After working hard competing in a programming event, Ula, Chad, and Fred decide to reward themselves with a tasty treat and visit the Universal Confection Factory (UCF) to get their favorite ice cream. There, they decide to order the largest ice cream cones their wallets can muster but are having trouble figuring out exactly how many scoops of ice cream a particular order would contain. Unfortunately, it seems like UCF has a peculiar method for ordering ice cream. Instead of specifying exactly how many scoops you want, you start with an ice cream cone containing 1 scoop and list qualifiers to increase the number of scoops by multiples. UCF offers the following qualifiers to an order:

| Qualifier | Multiplier |
|---|---|
| Double | ×2 |
| Triple | ×3 |
| Quadruple | ×4 |

For example, a Double Scoop ice cream will contain 2 scoops ($1 \times 2 = 2$) while a Triple Quadruple Scoop ice cream will contain 12 scoops ($1 \times 3 \times 4 = 12$). Each order will contain at least one of these qualifiers and each qualifier can be repeated as many times as needed. Unable to calculate the number of scoops involved in a large order in their heads, they ask you to write a program to do it for them.

**The Problem:**

For each order, output the total number of scoops of ice cream that the order contains. Because multiple people will be ordering ice cream, your program should work for multiple orders.

**The Input:**

The input consists of multiple ice cream orders. The input file begins with a positive integer, *n*, on a line by itself, which represents the number of orders that need to be processed. Each of the following *n* lines will contain 1 or more qualifiers followed by the word "`Scoop`" with each word separated by a single space. There will be at most 19 qualifiers in a particular order. The qualifiers can appear in any sequence, but "`Scoop`" will always be at the end of the line, signifying the end of the order. All of the input is case sensitive.

Despite the latest advancements in ice cream stacking technology, no ice cream cone will ever have more than 1,000,000 scoops.

**The Output:**

For each order, output "`Order #`*`i`*`:` *`x`*" where *i* is the current order number (starting at 1) and *x* is the total number of ice cream scoops in that order. Each order should be printed on its own line.

**Sample Input:**

```
3
Double Scoop
Triple Quadruple Scoop
Double Double Scoop
```

**Sample Output:**

```
Order #1: 2
Order #2: 12
Order #3: 4
```

# Pepe's Perfect Pizza Palace

*Filename:* `pizza`

Perry is on a diet and has given up pastries. However, he just cannot get himself to give up pepperjack, particularly if it is on pizza. He's a big fan of the pies at Pepe's Perfect Pizza Palace. However, pangs of guilt profusely pain him so he tries to limit how much he eats (he can't give it up completely, but he can pare down his portions). He also recently started doing pilates for exercise.

At Pepe's Perfect Pizza Palace the special is known as the Perfectly Popular Pizza. It's perfectly round and includes a plentiful portion of pepperoni, pepperjack and pepperoncinis. Luckily, it also comes with a pitcher of positively palatable pomegranate punch.

Perry wants to know how many calories he will consume if he eats the Perfectly Popular Pizza with its plentiful portion of pepperoni, pepperjack and pepperoncinis (the calorie count from the pitcher of positively palatable pomegranate punch is paltry so he will neglect that contribution to his intake). Pepe's Perfect Pizza Palace publishes the calorie count of the Perfectly Popular Pizza by the square inch to be 10 calories. Based on the perimeter and pieces (slices) of the Perfectly Popular Pizza and the number of pieces eaten by Perry, determine how many calories he partook (consumed).

**The Problem:**

Given the perimeter of a Perfectly Popular Pizza from Pepe's Perfect Pizza Palace (with its plentiful portion of pepperoni, pepperjack and pepperoncinis) and the number of pieces eaten by Perry, compute the number of calories that Perry partook. Use a value of 3.14159265 for $\pi$.

**The Input:**

Input will begin with a single, positive integer, *n,* on a line by itself, representing the number of Perfectly Popular Pizzas ordered by Perry. For each pizza, there will be three positive integers, $p$, $t$ and $c$, on a new line by themselves (each value is separated by a single space) where $p$ represents the perimeter of the Perfectly Popular Pizza in inches ($p \leq 300$), $t$ represents the total number of equal-sized pieces into which the Perfectly Popular Pizza is partitioned or sliced ($t \leq 50$), and $c$ is the number of pieces that Perry ate ($c \leq t$).

**The Output:**

For each Perfectly Popular Pizza, output the header "`Perfectly Popular Pizza i:` " where $i$ is the number of the pizza ordered by Perry (starting with 1). Following the header, output the phrase "`Perry consumed x calories.`" on the same line where $x$ is how many calories Perry partook. Round all values up to the next calorie (i.e. both 12.1 and 12.8 should be rounded up to 13). Each Perfectly Popular Pizza ordered by Perry is output on a new line.

**Sample Input:**

```
2
63 8 4
100 12 10
```

**Sample Output:**

```
Perfectly Popular Pizza 1: Perry consumed 1580 calories.
Perfectly Popular Pizza 2: Perry consumed 6632 calories.
```

# Montgomery Anaconda and the Sacred Chalice

*Filename:* `chalice`

By divine proclamation, Arturia was named ruler of all of Albion when she was given the holy saber, Excelsior, by the Woman in the Water. Unfortunately, the ignorant peasants, who consider themselves an autonomous autocracy, feel that her claim to rule is not proper for supreme executive power. Blasted peasants! In an attempt to prove herself as true ruler of the Brettons, the queen has undertaken a quest by the Big Guy in the Sky to search for the Sacred Chalice. Thus, Queen Arturia and her Sires of the Pentagonal Ottoman have begun their quest to find the Sacred Chalice, so that Arturia may claim her rightful place as Queen of the Brettons.

While on her quest, she comes across a village where a crowd of peasants are waving torches and pitchforks around a woman dressed in robes and a carrot on her nose, with loud chants of "She's a witch! Burn her!" being heard. Arturia's most intelligent sire, Bolivar, stepped forward to assess the situation. According to the villagers, this woman is a witch, and must, therefore, be burned. Unconvinced, Sire Bolivar suggested that there should be some test to ensure that she is a witch. But what kind of test should there be? Bolivar reasoned that if she is a witch, she is able to be burned. Because she can be burned, she must also be made of wood since it also burns. And if she is made of wood, she must also float. With that, the villagers began to ponder tossing her into the water to see if she'd float, only to realize that there is no large body of water nearby.

"Well", Bolivar then asked, "If she can float, she must be very light, at least as light as what?" The peasants stood dumbfounded. Then, the queen responded "A flock of geese!" to which Bolivar agreed whole-heartedly. Thus, if the woman truly is a witch, she must be as light as or lighter than a flock of geese. The crowd shouted in earnest and glee to test the woman of witchcraft. Unfortunately, their scales are not large enough to measure the woman and the flock of geese; thus, each must be weighed individually. Even worse, none of them are able to tell whether the woman or the flock of geese is heavier. Therefore, Queen Arturia has turned to you, Sire Not-Named-In-This-Problem, to use your programming skills to solve this dilemma, because here in 12th century Albion, no one else knows what in the world a computer is.

**The Problem:**

Given the weight in pounds of a person accused of witchcraft and the weight (also in pounds) of each of the geese in the trial's flock, determine whether or not the person is a witch, according to Sire Bolivar's reasoning. Since many other potential witches may be found, Queen Arturia would like you to make sure the program will work for multiple trials.

**The Input:**

The first line shall list a single positive integer, *n*, on a line by itself, stating the number of trials to be held. For each trial, the first line shall contain a single positive integer, *w* ($w \leq 500$), stating the weight of the accused. The next line shall contain a single integer, *f* ($1 \leq f \leq 10$), stating the number of geese in the flock. The next line shall contain *f* positive integers, each separated by a single space, telling the weight of each individual goose. No individual goose shall exceed 50 pounds.

**The Output:**

For each trial, output a line "`Trial #x: j`" where *x* is the current trial number (starting at 1), and *j* is either the phrase "`SHE'S A WITCH! BURN HER!`" if the accused is a witch according to her trial or "`She's not a witch. BURN HER ANYWAY!`" if she isn't. There should be one space after the colon.

**Sample Input:**

```
2
120
5
30 24 35 50 45
200
3
40 24 60
```

**Sample Output:**

```
Trial #1: SHE'S A WITCH! BURN HER!
Trial #2: She's not a witch. BURN HER ANYWAY!
```

# Lucas' Letters

*Filename:* `letters`

As with many young children, Lucas is learning his letters and often slightly mispronounces some. In particular, Lucas will often replace an "l" with a "w" and that can make for some interesting words! His brother, Logan, is just starting to speak and wants to practice. A converter to such a language will help him.

**The Problem:**

Given a number of words, replace each "l" in each word with a "w" and output the new word. Each word is given in all lowercase.

**The Input:**

Input will begin with a single, positive integer, *n,* on a line by itself, representing the number of words to convert. Each word will begin on a new line without any whitespace. Each word will be between 1 and 10 lowercase letters, inclusive.

**The Output:**

For each word, change each "`l`" to a "`w`" and output the new word on a new line by itself.

**Sample Input:**

```
6
lucas
logan
lollipop
what
look
wow
```

**Sample Output:**

```
wucas
wogan
wowwipop
what
wook
wow
```