Fifth Annual
University of Central Florida

High School Programming Tournament: Online Edition

Problems

Problem Name	Filename
Great Butterfly Wizards	butterfly
How'd We Do?	how
The Woe of Painting	painting
Professor Lateton and the Missed Appointment	professor
Red Light	redlight
BrokenSpace	space
Squirrel Squad	squirrel
Electronic Toilet Dilemma	toilet

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

For example, if you are solving Professor Lateton and the Missed Appointment:
Call your program file: professor.c, professor.cpp or professor.java
Call your Java class: professor

Great Butterfly Wizards

Filename: `butterfly`

Balthazar the adventurer is sad because his flight skill level is only 1 on a scale from 1 to 10^{18}. Fortunately, he recently arrived at the magical kingdom of Lepidoptera, which is the home of the great butterfly wizards (who are clearly flying experts). There are two types of butterfly wizards in Lepidoptera – queen butterflies and monarch butterflies. Currently there are *a* queen butterflies and *b* monarch butterflies that are willing to help. When Balthazar visits a queen butterfly, his flight skill level is increased by *x* (that is, his new flight skill level is the sum of his old skill level and *x*). When he visits a monarch butterfly, his flight skill level is multiplied by *x*. He is permitted by the laws of Lepidoptera to visit each butterfly exactly one time. Balthazar is smart and knows that the optimal strategy is to visit all the queen butterflies first, and then all the monarch butterflies. Unfortunately, queen and monarch butterflies look remarkably similar and Balthazar can't tell them apart, so he instead visits all of the butterfly wizards in a random order (that is, all $(a + b)!$ possible orders have equal probabilities). He wants to know what his expected flight skill level will be. The expected level can be calculated as follows: For each possible order of visiting the wizards, add the resulting level multiplied by the probability that Balthazar visits the wizards in that order.

The Problem:

Given *a* queen butterfly wizards that add *x* to Balthazar's level and *b* monarch butterfly wizards that multiply Balthazar's level by *x*, compute the expected value of Balthazar's level if he visits all of the butterfly wizards in a random order.

The Input:

The first line of input will contain a single, positive integer, *t*, representing the number of scenarios to process. Each of the next *t* lines will contain three integers, *x* ($2 \le x \le 10$), *a* ($0 \le a \le 15$) and *b* ($0 \le b \le 15$), representing the number the wizards use when changing Balthazar's level, the number of queen butterfly wizards and the number of monarch butterfly wizards, respectively.

The Output:

For each scenario, output a line containing "`Scenario #`*`i`*`: `*`e`*" where *i* is the number of the scenario in the input (starting with 1) and *e* is Balthazar's expected level after visiting the butterfly wizards. Output each answer to an absolute or relative error of 1e-9.

Sample Input:

```
3
10 0 5
3 2 2
5 12 12
```

Sample Output:

```
Scenario #1: 100000.000
Scenario #2: 35.000
Scenario #3: 1652644229.615
```

How'd We Do?

Filename: `how`

The UCF Programming Team often likes to practice using past problem sets from other contests. Since these sets were used in a real contest, this means that there is a scoreboard for that competition on this set. The UCF Programming Team wants to know how they did in comparison with all of the teams from the real contest. They would like your help to combine their practice scoreboard with the scoreboard from the real competition.

Just a reminder: ranks at programming contests are sorted based on the number of problems solved (highest first), breaking ties with teams having fewer penalty points placing better. It is guaranteed in this problem that no ties will remain unbroken after considering penalty points.

The Problem:

Given the scoreboard for a set, output a combined scoreboard in proper order with both sets of teams on it.

The Input:

The first line of input will be a single positive integer representing the number of problem sets to process. Each problem set will include the descriptions of the scoreboards for the official contest and the UCF Programming Team's practice on that same problem set.

Each scoreboard starts with a line containing two integers, n and m ($1 \leq n \leq 100{,}000$; $1 \leq m \leq 100{,}000$), representing the number of teams in the official contest and the number of teams from the UCF Programming Team in the practice, respectively. Following this will be n lines each describing a team's score from the official contest (in rank order). Following the official scoreboard will be m lines each describing a UCF team's score from their practice using the same problem set (also in rank order). A team's score consist of a single line starting with their name (a series of upper and lowercase letters between 1 and 10, inclusive, in length), followed by their number of problems solved, s ($0 \leq s \leq 10$), and their number of penalty points, p ($0 \leq p \leq 1{,}000{,}000$), each separated by a single space. All names are guaranteed unique from each other.

The Output:

For each competition, first output a line containing "`Competition #c:`" where c is the number of the competition in the input (starting with 1). On the following lines, output the combined scoreboard with each team in newly combined rank order. Each line of the scoreboard should list in order that team's rank, name, total problems solved and penalty points, each separated by a single space. Leave a blank line after the output for each competition.

Sample Input:

```
2
4 3
UCFOlympus 7 523
Stanford 5 627
GeorgiaTech 1 2000
UniversityOfFlorida 0 0
UCFBuyan 6 853
UCFZerzura 5 530
UCFAvalon 5 958
3 4
UCF 8 1207
CalBerkley 8 1263
MIT 6 645
Zerzura 7 966
Avalon 7 1221
Buyan 4 239
Valhalla 3 191
```

Sample Output:

```
Competition #1:
1 UCFOlympus 7 523
2 UCFBuyan 6 853
3 UCFZerzura 5 530
4 Stanford 5 627
5 UCFAvalon 5 958
6 GeorgiaTech 1 2000
7 UniversityOfFlorida 0 0

Competition #2:
1 UCF 8 1207
2 CalBerkley 8 1263
3 Zerzura 7 966
4 Avalon 7 1221
5 MIT 6 645
6 Buyan 4 239
7 Valhalla 3 191
```

The Woe of Painting

Filename: `painting`

Rob Boss hates painting. No matter how hard he tries, he can never completely cover a canvas in paint. It doesn't matter what kind of canvas he uses, those pesky corners always stay mockingly out of reach! This, of course, is due entirely to Rob's exclusive selection of circular brushes. When the Boss paints a canvas, he does so with a single brush of a certain radius to maintain the integrity of the work. Additionally, due to his professionalism, each of his brush strokes will remain wholely within the canvas, never crossing over an edge. Because of this, his circular brush will consistently become wedged in the corners of the canvas, leaving an area unpainted!

The last four episodes of Rob's public access painting show have ended with the Boss's leg sticking through the canvas and paint cans being flung about the studio. As a concerned viewer, you wonder if knowing the total area that would go unpainted on a given canvas, given a certain brush, might calm Rob enough to finish an episode.

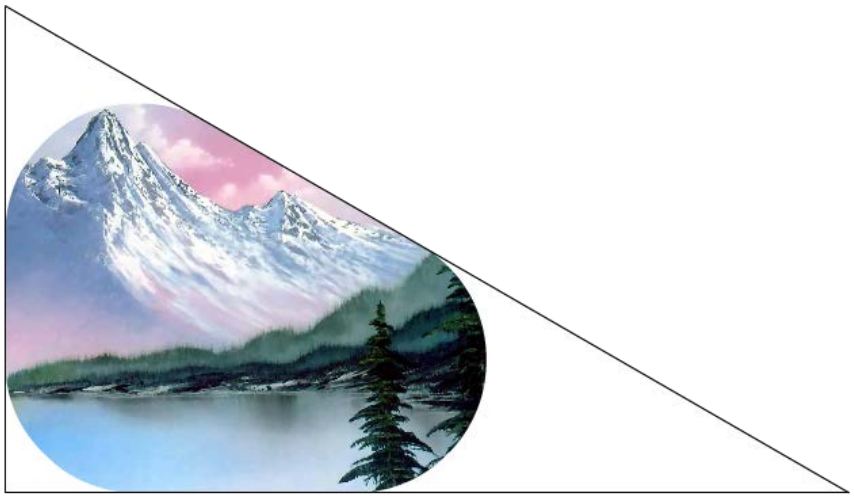

Figure 1: Second sample canvas and unpainted area

The Problem:

Given a canvas, specified as a list of angles, and the radius of Rob's brush, determine the total area of the canvas that cannot be painted. The given set of angles will always correspond to a simple convex polygon. It is guaranteed that Rob's brush can touch each side of the canvas at at least one point without going outside the canvas. If needed, use 3.141592653589793 for the value of π.

The Input:

The first line of the input file will begin with a single positive integer, *t*, representing the number of canvases the Boss will be painting this season. Following this will be *t* project descriptions. The first line of each project description will contain two positive integers, *n* ($3 \leq n \leq 100$) and *r* ($r \leq 1{,}000$), representing the number of corners on the canvas and the radius of the brush, in inches, respectively. On the following line will be *n* positive integers, each corresponding to the angle of a corner on the canvas, in degrees. Each angle will be between 1 and 179 degrees, inclusive. The angles for a canvas will always sum to (*n*-2)*180, but may appear within a project description in any order.

The Output:

For each canvas, output a single line in the form "`Canvas #`*`i`*`:` *`x`*" where *i* is the canvas number in the input (starting at 1) and *x* is the area of the canvas that cannot be painted in square inches, rounded to 4 decimal places. As an example of rounding, an area of 0.12345 would be rounded up to 0.1235, whereas an area of 0.12344 would be rounded down to 0.1234

Sample Input:

```
3
4 1
90 90 90 90
3 1
90 30 60
5 3
140 120 110 90 80
```

Sample Output:

```
Canvas #1: 0.8584
Canvas #2: 3.3225
Canvas #3: 6.2252
```

Professor Lateton and the Missed Appointment

Filename: `professor`

It was a month ago that Professor Lateton arrived at the curious village of Motchsteke to investigate the mysterious disappearance of a few local villagers. Of course, the letter informing the professor of these strange occurrences had arrived years prior, but he had other stuff to do and decided to get around to it later. Naturally, by the time the professor arrived in Motchsteke, the entire population had vanished. Unfortunately, the professor's investigation was only made worse from there. Over the past month he has been dragged from landmark to landmark by a series of increasingly irritating puzzles, an activity he finds tedious above all else. After a month of frustration and hair-pulling, the professor has nothing on his mind other than escaping this accursed village, mystery be damned! Sadly, the professor soon realizes this is easier said than done. Lost and exasperated, the professor decides to sit and wait for salvation, rather than risk stumbling upon another blasted puzzle.

It is not too long before a traveling salesman wanders into the curious village, looking to peddle his wares. When asked for directions, the salesman thinks for a moment before suddenly declaring that this very situation reminds him of a puzzle! The salesman enthusiastically describes the rules to an ancient game whilst Professor Lateton stomps around in a circle, waving his fists in the air and letting loose a stream of profanity.

The game the salesman describes is quite simple. To begin, a pile of n stones is placed on the ground. Two players will take turns removing at least one and up to m stones at a time from the pile. The last player to remove a stone loses the game! If the professor can beat the salesman at this game, the salesman will gladly provide directions to the nearest exit. Professor Lateton always gets to go first, but after a few dozen games he realizes that no matter what he does, the conniving salesman always wins! As tears begin streaming down his cheeks, he wonders, will he ever escape the puzzling village of Motchsteke?

The Problem:

Given a game in which the pile is composed of n stones and players may take at least one and up to m stones at a time, can Professor Lateton win the game if both players play optimally? A player playing optimally has insight into all possible future moves and will always play in such a way as to maximize their chances of winning. That is, an optimal player will never make a mistake that causes them to lose the game when they could have otherwise won.

The Input:

The first line of the input file will begin with a single positive integer, *t*, representing the number of games Professor Lateton will play. Following this will be *t* lines describing the games. Each game description will contain two integers *n* and *m* ($1 \leq m \leq n \leq 1{,}000$), representing the number of stones in the pile and the maximum number of stones a player can take at a time, respectively.

The Output:

For each game, output a single line in the form "`Game #`*`i`*`: `*`x`*" where *i* is the game number in the input (starting at 1) and *x* is either "`So long, Motchsteke!`" if the professor can win or "`Gentlemen don't cry.`" if there's no chance.

Sample Input:

```
3
12 3
7 2
6 2
```

Sample Output:

```
Game #1: So long, Motchsteke!
Game #2: Gentlemen don't cry.
Game #3: So long, Motchsteke!
```

Red Light

Filename: `redlight`

Aaron is extremely impatient and hates red lights, especially the last one on his way home from school, which is a very long light. He often turns right onto a slower road to avoid the red light only to see the light turn green a few seconds later. To avoid this infuriating situation and the resulting road rage, Aaron wants a program he can put on his phone to tell him if it's faster to turn or go straight.

The Problem:

Given a description of an intersection and how long Aaron has observed the light as red while approaching, determine if it is faster to take a right turn or go straight if the light takes as long as possible to turn green.

The Input:

The input will begin with a line containing a single positive integer, i, representing the number of intersections on which Aaron wants to run his program. Following this line, there will be i intersection descriptions. Each intersection description consists of two lines. The first line contains three integers, g ($1 \leq g \leq 1{,}000{,}000$), s ($1 \leq s \leq 1{,}000{,}000$) and r ($1 \leq r \leq 1{,}000{,}000$), representing the seconds it takes the traffic light to turn green after it becomes red, the seconds it takes to get home going straight after the light turns green, and the seconds it takes to get home turning right, respectively. The second line of each intersection description will contain a single integer, t ($0 \leq t < g$), representing the amount of time Aaron has observed the traffic light as red before he reached the intersection.

The Output:

For each intersection, output a line "`Intersection #`*x*`: `*a*" where x is the number of the intersection in the input (starting with 1) and a is the correct action for the intersection. If Aaron *saves* time in the worst case by waiting for the light to turn green and going straight the correct action should be "`Wait it out.`"; otherwise, the correct action is "`Right on red!`".

Sample Input:

```
4
200 400 500
90
200 400 500
110
600 300 800
599
600 300 800
0
```

Sample Output:

```
Intersection #1: Right on red!
Intersection #2: Wait it out.
Intersection #3: Wait it out.
Intersection #4: Right on red!
```

BrokenSpace

Filename: `space`

Help! John's space key is broken! Luckily, his shift key, as well as all of the letters and period key (which are the only other keys he uses), are working just fine. As such, to make it easier to read what he types, he is using camel case. Camel case means that the first letter of every word is capitalized and all other letters are lowercase. Can you write a program to convert John's camel case documents into proper spacing and capitalization?

It is guaranteed that the first letter of every word will be capitalized and all other letters within a word will be lower case. We define proper capitalization to mean that the first letter of every sentence is capitalized while all others are lowercase. The only exception to this is the word "I", which will always be capitalized. John always has at least one word per sentence and only uses periods to end sentences (so each sentence will end in a period and only a period; furthermore, periods will not appear elsewhere within a sentence). Proper spacing between words and sentences should be exactly one space.

The Problem:

Given John's document written in camel case, convert it to proper spacing and capitalization.

The Input:

The first line of the input will contains a single integer, *n*, representing the number of lines that John's document contains. Following this will be *n* lines of letters and periods (that do not contain spaces, obviously) written in camel case. Each line will be between 1 and 200 characters in length.

The Output:

For each line of John's document, output that line converted to proper spacing and capitalization, as defined above.

Sample Input:

```
3
Help.MySpaceKeyIsBroken.
PleaseHelp.IAmGoingToBreakMyShiftKeyNext.
GetToWork.
```

Sample Output:

```
Help. My space key is broken.
Please help. I am going to break my shift key next.
Get to work.
```

Squirrel Squad

Filename: `squirrel`

The following is classified. Most people know about the UCF police department. However, very few know that the squirrels of UCF make up their own department of special agents, who are sometimes called upon to handle some of the more challenging situations! When not working, these squirrels hang out in their many secret bases around campus. These bases are all interconnected by secret passages, which the squirrels have guaranteed are all equal in length.

An urgent situation has arisen that demands all of the squirrels' attention! They can pick any one base to activate the squirrel signal in, calling all squirrels to that base. However, they need your help to figure out which one is best!

The Problem:

Given a list of which bases are connected to which (by unit length passages), calculate which base to call all of the squirrels to that minimizes the farthest distance that any squirrel must travel to get there (assuming they take the shortest paths).

The Input:

To keep the information about their secret bases a secret, the squirrels will include some misinformation for you to calculate as well. The first line of input will contain a single positive integer, *n*, representing the number of maps to process (you'll have to process the fake ones and the real one just the same, they won't tell you which is the real one). Following this, *n* maps will follow.

Each map starts with a line with two integers, *b* ($1 \leq b \leq 100$) and *p* ($1 \leq p \leq 1{,}000$), representing the number of bases on the map and the number of passageways connecting them, respectively. Following are *p* lines, each describing one of the passageways. Each of these passageways are described simply with two distinct integers, *x* and *y* ($1 \leq x \leq b$; $1 \leq y \leq b$; $x \neq y$), indicating that there is a passageway directly connecting secret bases numbered *x* and *y* (which are simply numbered from 1 to *b* to keep their actual locations secret). It is guaranteed that a path (direct or indirect) exists between all pairs of bases.

The Output:

For each map given in the input, output a message:

`Map #`*`m`*`: Signal base `*`s`*`! It is at most `*`d`*` squirrel paths away!`

where *m* is the map number in the input (starting with 1), *s* is the base that the squirrel signal should be used in, and *d* is the maximum distance that any squirrel has to travel taking the shortest path to get there from any base. If there is more than one equivalently optimal answer, choose the one that has the smallest *s*.

Sample Input:

```
2
4 3
1 2
2 3
2 4
3 3
1 2
2 3
1 3
```

Sample Output:

```
Map #1: Signal base 2! It is at most 1 squirrel paths away!
Map #2: Signal base 1! It is at most 1 squirrel paths away!
```

Electronic Toilet Dilemma

Filename: `toilet`

The year is 2059 and now practically everything has been computerized and installed with work-optimizing intelligence capable of maintaining human-level conversation – even toilets. Recently however, it was deemed immoral to employ such intelligence to handle human waste due to the social stigma involved. That's why the current toilets at UCF are now retiring and being replaced by older, non-intelligent models.

The way the older non-intelligent toilets work is similar to the way they worked back in 2014: using sensors they wait until something is detected (hopefully a person) and then wait until that something leaves before finally flushing. Each toilet has two time thresholds: one for the time of continuous detection required to recognize an entity and another for the time of continuous non-detection to recognize emptiness.

For example, take a toilet that needs three seconds of detection to recognize an entity: if it detects for two seconds, doesn't detect anything on the third second, but detects on the fourth second, the entity would not be recognized since the three seconds of detection were interrupted. This ensures the toilet doesn't flush pre-maturely. The hardware for this simple functionality is in place for all the models, but the software to tell the toilet when to flush has been lost. UCF now needs you to write a program that will tell when the toilets should flush given the data from their hardware.

The Problem:

Given a string representing data from the toilet hardware and the times required for recognition, output at the end of which second(s) the toilet should flush.

The Input:

The input will begin with a line containing a single integer, n, representing the number of toilets to process. Following this, n toilet descriptions follow, one for each toilet. Each description begins with a line containing two integers, d and u ($1 \leq d \leq 100$; $1 \leq u \leq 100$), representing the number of continuous seconds of detection required to recognize an entity and the number of continuous seconds of non-detection required to recognize emptiness, respectively. On the next and final line of the description there is a string of length between 1 and 100 characters, inclusive, each of which is either '-' (representing one second of continuous non-detection) or '#' (representing one second of continuous detection) . The first character of the string represents the first second.

The Output:

For each toilet first output "`Toilet #`*x* `flush times:` " where *x* is the toilet number in the input (starting at 1). Then, on the same line output all times when the toilet should flush in seconds in increasing order (each separated by a single space). If there are no times when the toilet should flush, output "`None`" instead.

Sample Input:

```
3
2 3
--##-#-----#-###--
4 1
---####--#-##-##-###
3 3
###---#---####---#
```

Sample Output:

```
Toilet #1 flush times: 9
Toilet #2 flush times: None
Toilet #3 flush times: 6 17
```